
Abstract:

- Los contratos inteligentes son programas autoejecutables que se corren en la cadena de bloques y permiten que los pares hagan cumplir acuerdos sin la garantía de un tercero.
- El contrato inteligente en Ethereum es el elemento fundamental de las finanzas descentralizadas con miles de millones de dólares estadounidenses en valor. **Sin embargo, los contratos inteligentes están lejos de ser seguros y los ataques que explotan vulnerabilidades han provocado pérdidas valoradas en millones.**
- En este trabajo, se exploró el estado actual de la seguridad de los contratos inteligentes, las vulnerabilidades prevalentes y el soporte de herramientas de análisis de seguridad, mediante la revisión de los avances y la investigación publicada en los últimos cinco años.
- Se estudiaron 13 vulnerabilidades en los contratos inteligentes de Ethereum y sus contramedidas, e investigamos nueve herramientas de análisis de seguridad.

---

1. Introduccion

Las criptomonedas y las finanzas descentralizadas (DeFi) se caracterizan por el uso de blockchain para transferir fondos entre pares en redes sin intermediarios. **DeFi propone un modelo en el cual los participantes realizan acuerdos a través de la ejecución automatizada de contratos inteligentes basados en blockchain, y se basa en aplicaciones descentralizadas (DApps) que realizan funciones financieras en registros inmutables y públicamente visibles llamados blockchains.**

La mayoría de las DApps son contratos inteligentes que se ejecutan en una red blockchain y aprovechan sus protocolos de consenso para almacenar sus códigos fuente y aplicar cláusulas bajo reglas predefinidas para brindar servicios confiables.

Ethereum es la primera y la plataforma más prominente que admite la implementación de contratos inteligentes en un lenguaje de programación de alto nivel, como Solidity. **Ethereum es la segunda plataforma blockchain más grande con una capitalización de mercado de 317.3 mil millones de dólares estadounidenses en junio de 2021, y proporciona un entorno de ejecución para más del 95% de las aplicaciones DeFi. Bitcoin también admite el desarrollo y la ejecución de contratos inteligentes, pero el lenguaje de script que utiliza tiene limitaciones, como un soporte limitado para transacciones más allá de la verificación de firmas.** Ethereum es una plataforma de computación respaldada por la Máquina Virtual de Ethereum (EVM), que es una máquina de estado que crea un entorno seguro para ejecutar contratos inteligentes en el cual el estado se refiere al registro.

El contrato inteligente es un elemento fundamental de DeFi y, por lo tanto, su seguridad es de gran importancia. Los contratos inteligentes no se pueden cambiar después de su implementación, por lo que es necesario verificar su código en busca de posibles vulnerabilidades. **Los ataques exitosos a contratos inteligentes han causado pérdidas financieras significativas, como el conocido ataque a la Organización Autónoma Descentralizada (DAO) que evaporó millones de dólares y obligó a realizar una bifurcación en Ethereum.**

En este artículo, investigamos las vulnerabilidades relacionadas con los contratos inteligentes mediante la revisión de los trabajos más recientes publicados desde 2018 para complementar estudios anteriores y abordar las siguientes preguntas de investigación:

• RQ1: ¿Qué vulnerabilidades en los contratos inteligentes de Ethereum se han estudiado recientemente?

• RQ2: ¿Cuáles son las posibles contramedidas para mitigar las vulnerabilidades de los contratos inteligentes?

• RQ3: ¿Cuál es la precisión, eficiencia y limitaciones de las herramientas existentes de análisis de seguridad para contratos inteligentes?

---

2. Contexto

a. Cuentas de Ethereum

El elemento básico de Ethereum son las cuentas, también llamadas estado de cuenta, y cada cuenta tiene cuatro campos: nounce, balance, storage y code. **El nounce es un contador de transacciones, lo que significa que para cada nueva transacción enviada por esta cuenta, el nounce se incrementará en uno y se adjuntará en la estructura de datos de la transacción; el balance es la cantidad de Ether (la moneda utilizada en la plataforma Ethereum) que posee la cuenta; el almacenamiento es el espacio de memoria para el código y su ejecución; el código es donde se almacena el código del contrato inteligente**. Hay dos tipos de cuentas: cuentas externas y cuentas de contrato. La diferencia principal entre ambas es si el campo del código está vacío o no**. Las cuentas externas están controladas por pares de clave pública-privada (propiedad de los titulares de cuentas humanas), mientras que las cuentas de contrato están controladas por su código. Ambas cuentas se hashean y se almacenan en una estructura de datos llamada modified Merkle Patricia Tree , cuya raíz se hashea y se almacena en cada bloque**. Las cuentas externas pueden iniciar una acción que altera el estado de la Máquina Virtual de Ethereum (EVM), lo que se denomina transacción. Las transacciones se transmiten a toda la red. Un minero luego seleccionará y ejecutará las transacciones y propagará el cambio de estado resultante al resto de la red.

---

b. Ciclo de vida de un contrato inteligente

Se describen cuatro etapas diferentes del contrato inteligente: creación, despliegue, ejecución y finalización.

• **Creación**. En su etapa de creación, un EVM puede ser objetivo de varios lenguajes de alto nivel, como Solidity , Serpent o Bamboo . **El más comúnmente utilizado es Solidity, que es un lenguaje similar a JavaScript y completo en términos de Turing. El EVM no puede ejecutar directamente los códigos de Solidity, por lo que se compilan en opcodes (instrucciones de bajo nivel utilizadas por el EVM) y se codifican en bytecodes por razones de almacenamiento.**

• **Despliegue**. El protocolo de Ethereum describe dos tipos de transacciones: una que invoca llamadas de mensaje y otra que resulta en el despliegue de contratos**. El desarrollador inicia una transacción que contiene bytecodes (almacenados en un campo llamado "init" en la estructura de la transacción), y esta acción devuelve otro fragmento de código que se almacenará en el entorno de ejecución del EVM y se ejecutará más tarde.**

• **Ejecución**. Durante la etapa de ejecución, un contrato inteligente es un programa en ejecución, similar a un proceso o hilo en una computadora independiente. **Recibirá transacciones (del primer tipo) y los datos que se pasarán al contrato como parámetros. Luego, el EVM ejecuta las instrucciones una por una hasta que finalice o se alcance el límite de gas. Este proceso ocurre cuando se mina un nuevo bloque.**

• **Finalización.** Después de la ejecución, los estados se actualizan y se almacenan en las blockchains junto con las transacciones. Esto completa el ciclo de vida del contrato inteligente.

---

c. **Entorno de desarrollo de Ethereum**

- **Block: En Ethereum consta de tres partes: encabezado, lista de transacciones y lista de ommers. El encabezado contiene tres nodos raíz modificados de MPT (stateRoot, transactionRoot y receiptsRoot), junto con otra información sobre el bloque. Estos tres Modified MPT son el world state trie , transaction trie y receipts trie . Dentro de estos tries se almacenan todas las cuentas, su estado y transacciones. Existe otro Modified MPT, el Account storage contents trie, cuyo nodo raíz se almacena en el estado de cuenta del world state trie. Las dos listas siguientes son transacciones reales y ommers, que son seleccionados por el minero durante el proceso de minería. Los ommers son bloques que tienen un padre igual al padre del padre del presente bloque.**
- **World State Trie: Contiene un mapeo entre el estado de una cuenta y su dirección, lo que vincula todas las cuentas, incluidas las cuentas externas y los contratos, en este trie. Cada bloque tiene un único trie del estado mundial.**
- **Account storage contents tire: Almacena todos los datos de los contratos.**
- **Transaction trie: Almacena el hash de todas las transacciones incluidas en un bloque y no se modifica una vez que el bloque es minado.**
- **Receipts Trie: Almacena los resultados de la ejecución de transacciones, la cantidad acumulada de gas utilizado, los registros y el código de estado de la transacción. La serialización de esta información, combinada con claves, se almacena en el trie de recibos.**

---

3. Vulnerabilidades y contramedidas

Aunque la tecnología blockchain se considera generalmente segura, inmutable y anónima, no está exenta de problemas de seguridad, vulnerabilidades y ataques, especialmente en relación con los contratos inteligentes. Los ataques a DAO y el hackeo de Parity Wallet son ejemplos destacados de ataques que explotaron vulnerabilidades en los contratos inteligentes.

---

a. **Re-entry**

La reentrancia describe una situación en la que un contrato A llama a un contrato B, que a su vez podría llamar de vuelta a A y ejecutar la llamada de A nuevamente. El mecanismo de fallback de Solidity causa esta situación. La función fallback se ejecuta cuando las llamadas desde otros contratos no pueden encontrar una función coincidente. Cuando el llamante utiliza la función call sin proporcionar ninguna firma de función, se activará la función fallback del receptor. Esta función podría llamar a la función del llamante para volver a entrar en el llamante. **Este mecanismo puede provocar transferencias inesperadas e incontroladas de Ether en algunas circunstancias.**

Al principio, el atacante llama a la función withdraw() en el contrato A para solicitar un retiro de un Ether. El contrato A utiliza call para enviar el Ether y no adjunta ninguna firma de función. La función fallback del contrato atacante responde a esta llamada llamando nuevamente a la función withdraw de A. La segunda llamada se considera una llamada "anidada" dentro de la llamada de retiro anterior, ya que la anterior aún no ha finalizado. En este momento comienza una llamada recursiva. **Normalmente, el llamante recibiría un Ether y el saldo del llamante se deduciría con la ejecución de la instrucción balances[msg.sender] -= _amount;.** Sin embargo, en una situación de reentrancia, esta instrucción nunca se ejecutará debido a que la función call sobre esta línea llevará a llamar de forma recursiva a la función withdraw() hasta que el saldo total del contrato A sea inferior a un Ether o se alcance el límite de gas.

CONTRAMEDIDAS:

- Existen tres técnicas para prevenir la reentrancia. La primera es evitar el uso de la función call siempre que sea posible. En su lugar, se puede utilizar la función transfer para enviar Ether a otras cuentas. **Esta función solo envía 2300 gas con las llamadas externas, por lo que el contrato llamado no tendrá suficiente gas para volver a entrar en el contrato llamante.**
- La segunda técnica es agregar un "bloqueo", que generalmente es una variable de estado que indica el estado de las llamadas externas y permite la llamada externa solo si el estado es correcto
- La tercera técnica es realizar todos los cambios lógicos antes de ejecutar la función call. Por ejemplo, en el fragmento de código del Listado 1, se debe mover la línea balances[msg.sender] -= _amount antes de msg.sender.call{value: _amount}('').

---

b. Arithmetic Issues

En Solidity, el tipo de dato entero tiene un rango específico. Durante las operaciones aritméticas en variables, sus valores pueden superar los límites superiores o inferiores. Si eso sucede, el valor se ajustará al otro lado del límite.

Por ejemplo, si el valor real es mayor que el límite superior, el valor actual será el valor real menos el límite superior. En versiones anteriores de Solidity, esta anomalía no activa ninguna alerta. Los atacantes podrían aumentar o disminuir valores enteros específicos para engañar al contrato inteligente y lograr un comportamiento no deseado.

CONTRAMEDIDAS:

Dado que la vulnerabilidad aritmética se encuentra en Solidity, utilizar una biblioteca bien diseñada y auditada como SafeMath proporcionada por OpenZepplin en lugar de las operaciones aritméticas incorporadas ayudará a mitigar el riesgo.

---

c. **Delegatecall to Insecure Contracts**

Delegatecall es una función especial en Solidity. Es similar a la función call, pero con una diferencia importante. El delegatecall reutiliza el código del contrato llamado y lo ejecuta en el contexto del contrato llamador. Por ejemplo, msg.sender y msg.data en el contrato VulnerableContract serán la dirección y los datos del llamador, y las variables de estado utilizarán las del contrato llamador. Una vez que el contrato InsecureContract es llamado mediante delegatecall, la función doSomething se ejecuta en el contexto del llamador, lo que significa que la variable de estado owner del contrato VulernableContract tendrá una modificación inesperada. El bloque VulnerableContract no proporciona ninguna forma de cambiar su propiedad una vez que se ha construido**. Sin embargo, cuando el atacante llama a este contrato en un cierto punto de ejecución, hará un delegatecall al contrato InsecureContract, en el cual la variable de estado owner se modificará con el nuevo valor de msg.sender, que es la dirección del atacante. En otras palabras, delegatecall otorga a otros contratos, como InsecureContract, el permiso para modificar sus propias variables de estado, como address public owner. Esto se debe a que la función delegatecall mantiene el contexto.**

CONTRAMEDIDAS:

Dado que DELEGATECALL aprovecha el cambio de contexto de los contratos inteligentes, lo primero que se debe hacer es verificar el posible contexto de los contratos y sus bibliotecas de llamada (u otros contratos). Solidity también proporciona la palabra clave Library para restringir la alineación de las ranuras de variables de estado con las variables de estado del llamador. La biblioteca de Solidity no puede tener variables de estado propias, por lo que no habrá cambio de contexto ni modificación no autorizada de las variables de estado del llamador.

---

d. Selfdestruct

La operación selfdestruct en Solidity proporciona una forma de eliminar contratos de los bloques siguientes. Sin embargo, esta operación es peligrosa, ya que si no se actualiza la dirección del contrato en todos los remitentes, algunos seguirán enviando Ether al contrato destruido, lo que causará la pérdida de Ether. Además, este proceso enviará de forma forzada el Ether restante del contrato a una dirección designada. Adversarios maliciosos podrían aprovechar este comportamiento para enviar Ether de forma forzada a algunos contratos.

CONTRAMEDIDAS:
Para evitar una llamada de selfdestruct inesperada, se debe tener precaución al realizar llamadas externas. Al mismo tiempo, la lógica del contrato debe evitar depender del valor de this.balance, ya que puede ser modificado de forma forzada por atacantes.

---

e. Freezing Ether

Esta vulnerabilidad también es conocida como "greedy contract" o "locked money" . Un contrato inteligente está diseñado para poder recibir y enviar Ether. Una situación de congelamiento de Ether ocurre cuando un contrato solo puede recibir Ether pero no tiene la capacidad de enviar Ether. Un contrato puede ser codicioso y el Ether enviado a su dirección quedará bloqueado si el contrato no define ninguna función de retiro (withdraw).

CONTRAMEDIDAS:
Es una buena práctica evitar esta situación durante la fase de diseño del contrato inteligente. Aquellos contratos que pueden recibir Ethers deben tener funciones para retirar (withdraw) los Ethers. Esto permitirá que los usuarios puedan recuperar los fondos enviados al contrato y evitará que el Ether quede bloqueado en el contrato sin posibilidad de ser enviado a otra dirección.

---

f. CAUSAS

- **Visibilidad: La visibilidad de las transacciones en Ethereum puede comprometer la privacidad de los contratos inteligentes, lo que puede ser problemático en términos de confidencialidad.**
- **Opacidad: La falta de transparencia en los contratos inteligentes plantea preocupaciones, ya que muchos de ellos son opacos y manejan grandes sumas de dinero, lo que requiere una auditoría exhaustiva del código fuente.**
- **Inmutabilidad: La incapacidad de modificar los contratos inteligentes una vez desplegados puede perpetuar contratos problemáticos en la cadena, lo que requiere soluciones alternativas y plantea preocupaciones de seguridad adicionales.**
- **Ejecución Automatizada: Las interacciones entre contratos inteligentes pueden dar lugar a transferencias de flujo de control no controladas, lo que podría permitir a atacantes maliciosos acceder a funciones de alto privilegio.**
- **Minería: La elección del minero puede afectar el orden de ejecución de las transacciones, lo que introduce incertidumbre y puede tener un impacto significativo en las transacciones sensibles al orden.**
- **EVM: La utilización de oráculos en la Máquina Virtual de Ethereum (EVM) plantea desafíos en términos de integridad, precisión y consistencia de los datos provenientes del mundo real.**
- **Lenguaje de Programación Inmaduro: Solidity, el lenguaje de programación utilizado en Ethereum, aún se encuentra en evolución y presenta vulnerabilidades conocidas como el problema de reentrancia y el manejo inadecuado de excepciones.**

---

4. **Herramientas de Análisis de Seguridad**

En esa sección, se estudiaron las herramientas de análisis de seguridad de contratos inteligentes existentes y se compararon nueve herramientas recientemente propuestas que se lanzaron después de 2017. Estas herramientas aplican principalmente análisis estático o dinámico, aunque algunas incorporan enfoques de aprendizaje profundo o se centran en tipos específicos de vulnerabilidades. El estudio complementa investigaciones anteriores y se enfoca en herramientas propuestas después de 2018 que son específicas de Ethereum y son ampliamente utilizadas o mencionadas en múltiples estudios recientes. No se consideraron algunas herramientas que no cubren las vulnerabilidades presentadas o cuyos desarrolladores/autores no informan sobre su eficiencia o precisión.

---

smartCheck. Esta herramienta de análisis estático utiliza ANTLR para traducir el código fuente de Solidity en un árbol de análisis XML y lo verifica con patrones XPath. Detecta vulnerabilidades basadas en patrones predefinidos.

DefectChecker. Esta herramienta toma los bytecodes como entrada, los desensambla en opcodes, los divide en bloques básicos y ejecuta instrucciones de manera simbólica en cada bloque. Luego genera un grafo de flujo de control (CFG) y registra eventos de pila para detectar características y vulnerabilidades específicas.

contractWard. Utiliza aprendizaje supervisado para encontrar vulnerabilidades. Extrae características de bigramas de 1619 dimensiones de los opcodes utilizando un algoritmo n-gram y utiliza algoritmos de clasificación para detectar vulnerabilidades basadas en etiquetas previas.

NPChecker. Analiza el no determinismo en la ejecución de contratos inteligentes y realiza modelado sistemático para exponer diversos factores no determinísticos en el contexto de ejecución del contrato.

MadMax. Utiliza el descompilador Vandal para producir un código de tres direcciones y límites de funciones. Analiza el código de tres direcciones, reconoce conceptos como bucles y variables de inducción, analiza estructuras de memoria y datos dinámicos, e infiere vulnerabilidades relacionadas con el consumo de gas.

Osiris. Combina análisis simbólico y de contaminación para encontrar errores enteros en contratos inteligentes. Crea un grafo de flujo de control y ejecuciones simbólicas de cada camino en el grafo, realiza análisis de contaminación en la pila, memoria y almacenamiento, y detecta posibles errores enteros en las instrucciones ejecutadas.

contractFuzzer. Combina análisis estático de ABI y bytecodes con tecnología de fuzzing para explorar vulnerabilidades en contratos inteligentes. Crea una red de prueba de Ethereum para monitorear la ejecución de los contratos y genera casos de prueba basados en la información extraída de los ABI y bytecodes.

Sereum. Es una versión modificada del cliente Ethereum basada en Geth que se enfoca en vulnerabilidades de reentrancia. Agrega un motor de contaminación y un detector de ataques al cliente Geth para detectar instrucciones condicionales influenciadas por variables de almacenamiento y crear bloqueos que impiden actualizaciones adicionales en variables de estado que influyen en los flujos de control.

---

sFuzz. Es un fuzzer adaptativo que combina estrategias de AFL con estrategias adaptativas en Aleth. Utiliza un enfoque de fuzzing guiado por retroalimentación y monitorea la ejecución y eventos de pila para detectar vulnerabilidades.

---

5. Conclusiones

Se realizó un estudio de 13 vulnerabilidades en contratos inteligentes de Ethereum y sus contramedidas, y se examinaron nueve herramientas de análisis de seguridad.

Las inconsistencias en las definiciones de las diferentes vulnerabilidades en distintos trabajos de investigación dificultan la creación de una métrica o referencia unificada para la comparación entre las herramientas de análisis de seguridad.

Con convenciones de nombres consistentes y definiciones ampliamente aceptadas de las vulnerabilidades se facilitaría la investigación y las comparaciones.

Se necesitan más esfuerzos de investigación para ampliar este trabajo y se requieren nuevas técnicas de análisis de seguridad para mejorar aún más la seguridad de los contratos inteligentes.

Una posible extensión futura de este trabajo sería estudiar las vulnerabilidades relacionadas con plataformas blockchain distintas de Ethereum, como Hyperledger Fabric, Hyperledger Sawtooth, EOS, Tezos, Solana, Corda, NEO y Cardano.
