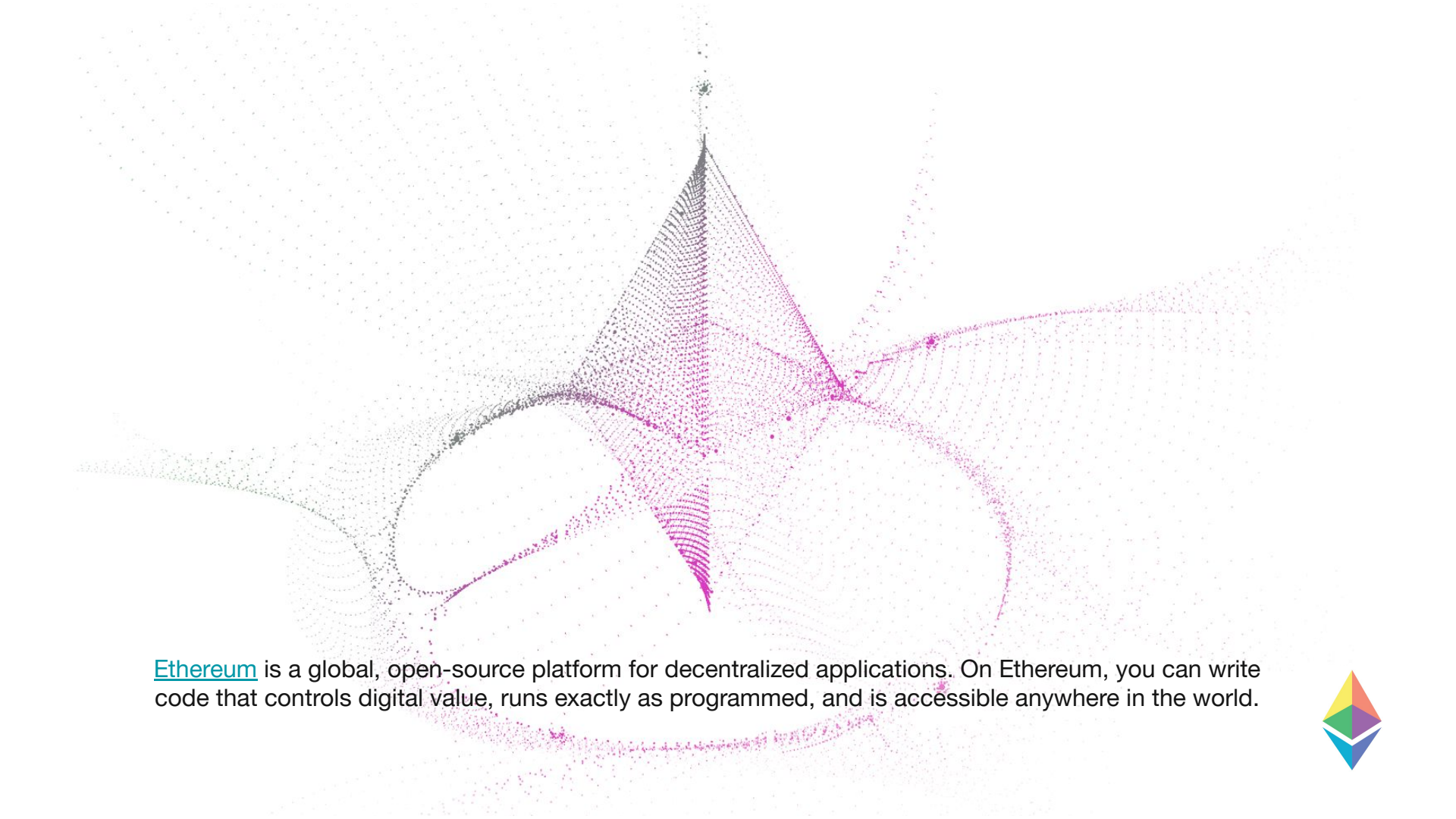


Ethereum Smart Contracts

Subtleties

A “smart contract” is simply a piece of code that is running on Ethereum. It’s called a “contract” because code that runs on Ethereum can control valuable things like ETH or other digital assets.





Ethereum is a global, open-source platform for decentralized applications. On Ethereum, you can write code that controls digital value, runs exactly as programmed, and is accessible anywhere in the world.





- La seguridad es una de las consideraciones más importantes al escribir contratos inteligentes:
 - **Los errores son costosos** (P. Ej. Ataque DAO).
 - **Los errores son fáciles de explotar.** Todos los contratos inteligentes son públicos, y cualquier usuario puede interactuar con ellos simplemente creando una transacción.
- Es fundamental seguir unas buenas prácticas de desarrollo y utilizar patrones de diseño que hayan sido probados.



Antes de Empezar - Programación Defensiva

La programación defensiva es un estilo de programación adecuado para el desarrollo de contratos inteligentes. Se basa en las siguientes prácticas:

- **Menos es Más:** Cuanto más simple es el código, menor es la posibilidad de que ocurra un error. Se debe prestar especial atención a contratos inteligentes con un gran número de líneas de código. **Más simple es más seguro.** La complejidad es el enemigo de la seguridad.
- **Reutilización de código:** El uso de librerías está altamente recomendado, **se trata de no reinventar la rueda.** Si en el desarrollo de un contrato se repite más de una vez el uso de una función, se debe traspasar a una librería y aprovechar su reutilización. Utilizar estándares o fragmentos de códigos ya probados es más seguro que cualquier código nuevo.
- **Calidad del código:** Cuando se desarrollan contratos inteligentes, se debe prestar especial atención a la calidad del código. Esto se debe a que **una vez desplegado el contrato, éste no admite cambios.** Además, si contiene errores, éstos pueden causar elevadas pérdidas de dinero. Se deben utilizar metodologías de ingeniería y desarrollo de software.



Antes de Empezar - Programación Defensiva

- **Legibilidad/Auditabilidad:** A la hora de desarrollar un Smart Contract se debe ser claro para facilitar la comprensión y la auditabilidad del mismo. Se recomienda utilizar el conocimiento de la comunidad, herramientas colaborativas y hacer público el trabajo que se está llevando a cabo para lograr un beneficio común. **Hay que tener en cuenta que el bytecode siempre será público y se pueden aplicar técnicas de ingeniería inversa para encontrar vulnerabilidades.**
 - Guía de las convenciones de estilo y nomenclatura: <https://solidity.readthedocs.io/en/latest/style-guide.html>
- **Testing:** Se deben realizar todos los test posibles sobre la funcionalidad del Smart Contract. Nunca se ha de suponer que los parámetros de entrada a una función son los correctos en cuanto a formato o longitud. Además, tampoco se debe dar por supuesto una ejecución no maliciosa del código desplegado. **Hay que tener en cuenta que el entorno donde se ejecutan los contratos es público.**



Reentrada

- **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 fallback**. Por lo tanto, cuando un contrato envía ether a esta dirección, invocará el código malicioso que contiene dicha función. Normalmente, el código malicioso ejecuta una función en el contrato vulnerable, realizando operaciones que el desarrollador de dicho contrato no espera. 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 lo "vuelve a ingresar" en el contrato que generó la primera transacción.

- **Ejemplo:** [EtherStore.sol](#), [AttackEtherStore.sol](#)

- **Técnicas Preventivas:**

- Usar la función **"transfer"** al enviar ether a contratos externos. La función **"transfer"** solo envía 2300 unidades de gas con la llamada externa, que no es suficiente para que la dirección / contrato de destino llame a otro contrato (es decir, vuelva a ingresar el contrato de envío).



Reentrada

- Garantizar que toda **la lógica que cambia las variables de estado ocurra antes de que se envíe ETH fuera del contrato** (o cualquier llamada externa). En EtherStore.sol, las líneas 45 y 48 deben colocarse antes de la línea 42. Es una buena práctica que cualquier código que realice llamadas externas a direcciones desconocidas sea la última operación en una función localizada o ejecución de código. Esto se conoce como el patrón [checks-effects-interactions](#).
- Usar un **mutex o lock**. Agregar una variable de estado que bloquea el contrato durante la ejecución del código, evitando llamadas reentrantes.
- **Ejemplo Aplicando Técnicas Preventivas:** [EtherStoreAPT.sol](#)
- **Ejemplo del mundo Real:** El ataque DAO (Organización Autónoma Descentralizada) fue uno de los principales ¿ataques? que ocurrieron en el desarrollo temprano de Ethereum. En ese momento, el contrato DAO tenía más de \$ 150 millones. La reentrada jugó un papel importante en el ataque, que finalmente condujo al Hard Fork que creó Ethereum Classic (ETC). [Analysis of the DAO exploit](#)



Over/Underflows

- **Vulnerabilidad:**

La máquina virtual de Ethereum (EVM) especifica **tipos de datos de tamaño fijo para enteros**. Esto significa que una variable entera sólo puede representar un cierto rango de números.

Por ejemplo, el tipo uint8 (entero sin signo de 8 bits) sólo puede almacenar números en el rango 0...255.

- Intentar almacenar 256 en una variable del tipo uint8 dará como resultado 0.
- Sumar 1 a una variable del tipo uint8 cuyo valor es 255 dará como resultado en número 0 (Overflow)
- Restar 1 a una variable del tipo uint8 cuyo valor es 0 dará como resultado el número 255 (Underflow).
- Sumar 256 a una variable del tipo uint8 cuyo valor es de X dejará a la variable sin cambios.

Por ello, **las variables de tipo entero pueden explotarse** si la entrada del usuario no se comprueba previamente o se realizan cálculos en operaciones que dan como resultado números que se encuentran fuera del rango del tipo de datos que los almacena.

- **Ejemplo:** [TimeLock.sol](#)

- Si un atacante se hiciera con las claves de un usuario, parece que debería esperar al menos una semana para poder desbloquear los fondos de la víctima por lo que este contrato parece una buena forma de protegerse de este tipo de robos.



Over/Underflows

- Sin embargo, si el atacante es listo podrá explotar la vulnerabilidad Arithmetic Overflow que tiene el contrato.
 - El atacante puede saber el locktime actual de la víctima (userLockTime) ya que la variable que lo almacena es pública (LockTime).
 - Después, puede llamar a la función raiseLockTime y pasar como argumento el número $2^{256} - \text{userLockTime}$. Este número se agregaría a userLockTime actual y provocaría un desbordamiento, estableciendo lockTime[msg.sender] a 0.
 - Por último, el atacante puede llamar a la función withdraw para obtener los fondos de la víctima.
- **Técnicas Preventivas:**
 - Uso de librerías que reemplacen la suma, resta y multiplicación de los operadores matemáticos estándar. OpenZeppelin creó la librería SafeMath para evitar vulnerabilidades de Over/Underflows.
 - <https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol>
- **Ejemplo Aplicando Técnicas Preventivas:** [SafeMath.sol](#), [TimeLockAPT.sol](#)
- **Ejemplo del mundo Real:** Implementación de la función batchTransfer() en un grupo de contratos del token ERC20 que contenía la vulnerabilidad de Overflow.
 - <https://medium.com/@peckshield/alert-new-batchoverflow-bug-in-multiple-erc20-smart-contracts-cve-2018-10299-511067db6536>



Visibilidad predeterminada

- **Vulnerabilidad:**

Las funciones en Solidity tienen cuatro especificadores de visibilidad que dictan cómo se pueden llamar.

- **External:** Las funciones externas son parte de la interfaz del contrato, lo que significa que **pueden llamarse desde otros contratos mediante transacciones**. Una función externa no se puede llamar internamente.
- **Public:** Las funciones públicas son parte de la interfaz del contrato y **pueden llamarse externamente o internamente** mediante mensajes.
- **Internal:** Las funciones internal sólo pueden llamarse internamente (desde el contrato actual o los contratos derivados de él).
- **Private:** Las funciones private sólo pueden llamarse desde el contrato en el que están definidas.

Las funciones predeterminadas, por defecto, son públicas, lo que permite a los usuarios llamarlas externamente.

El uso incorrecto de los especificadores de visibilidad puede conducir a algunas vulnerabilidades devastadoras en los contratos inteligentes.

Todo lo que está dentro de un Smart Contract es visible para todos los observadores externos a la blockchain. Hacer algo privado sólo evita que otros contratos lean o modifiquen la información, pero aún así será visible para todo el mundo fuera de la blockchain.



Visibilidad predeterminada

- Ejemplo: [AddressGame.sol](https://addressgame.sol)

Se trata de un Smart Contract diseñado para actuar como un juego de recompensas por adivinar direcciones que una vez truncadas a un uint8 sean mayores que la dirección del smart contract también truncada a un uint8. Una vez logrado, el Smart Contract llama a la función de retirada de Ganancias (sendWinnings) para que el usuario obtenga su recompensa.

Desafortunadamente la visibilidad de las funciones no se ha especificado. **La función sendWinnings es pública** (el valor predeterminado) y, por lo tanto, cualquier dirección puede llamar a esta función para robar la recompensa.

- **Técnicas Preventivas:**

- **Especificar siempre la visibilidad de todas las funciones** en un contrato, incluso si son intencionalmente públicas.
- **Utilizar versiones del compilador de Solidity iguales o superiores a la 0.5.0**, las cuales generan un error de compilación por no definir la visibilidad de las funciones. Las versiones de los compiladores iguales o superiores a la 0.4.17 muestran un warning.
- Aún así, habrá que tener especial cuidado en saber qué visibilidad aplicar a las funciones, ya que una mala definición de la misma, como hemos visto, puede tener consecuencias desastrosas.



Visibilidad predeterminada

- **Ejemplo Aplicando Técnicas Preventivas:** [AddressGameAPT.sol](#)
- **Ejemplo del mundo Real:** En **el primer hack Multisig de Parity**, se robaron alrededor de 31 millones de \$ en ETH principalmente de tres billeteras debido a que en varias funciones que gestionaban la propiedad de las billeteras no se especificó la visibilidad y fueron predeterminadas como public. Un atacante pudo explotar esta vulnerabilidad y transfirió la propiedad de varias billeteras a la dirección del atacante. Siendo el dueño el atacante, agotó las billeteras de todo su ETH
 - <https://www.freecodecamp.org/news/a-hacker-stole-31m-of-ether-how-it-happened-and-what-it-means-for-ethereum-9e5dc29e33ce/>



Valores de retorno no verificados utilizando CALL

- **Vulnerabilidad:**

Existen distintas formas de realizar llamadas externas en Solidity:

<https://solidity.readthedocs.io/en/latest/units-and-global-variables.html#members-of-address-types>

Las funciones `send()` o `call()` devuelven un valor booleano para indicar si la ejecución de la llamada se realizó correctamente. Esto quiere decir que ante la ejecución fallida de la llamada no se realiza un `revert()` como sucede con la función `transfer()`, sino que la transacción se ejecuta dando como resultado el valor `false`. La vulnerabilidad existe cuando el desarrollador del Smart Contract no tiene en cuenta este comportamiento y no hace una verificación del valor devuelto.

La vulnerabilidad puede provocar pérdida de Ether o incluso que un Smart Contract quede inutilizable.

- **Técnicas Preventivas:**

- La recomendación típica es usar la función `transfer()` para el traspaso de fondos.
 - Importante: <https://eips.ethereum.org/EIPS/eip-1884>
- Si se requiere utilizar la función `send()` o `call()` se debe verificar el valor de retorno de la llamada.
- Uso de patrones de retirada de fondos con el fin de aislar la lógica de la llamada externa del resto de la función.
 - <https://solidity.readthedocs.io/en/latest/common-patterns.html#withdrawal-from-contracts>



Valores de retorno no verificados utilizando CALL

- **Contrato a analizar:** [Sender.sol](#)

Smart Contract mediante el cual se realizan llamadas a funciones a partir de la función `send()`.

- **Contrato a analizar:** [Calling.sol](#)

Smart Contract mediante el cual se realizan llamadas externas a un sistema de votación.

- **Ejemplo del mundo Real:** Etherpot (lotería basada en un Smart Contract).

En la función `cash(...)` del contrato `lotto.sol` se puede ver como no se realiza una verificación del valor de retorno de la llamada al utilizar la función `send()`.

<https://github.com/etherpot/contract/blob/master/app/contracts/lotto.sol#L99>



Ether inesperado

- **Vulnerabilidad:**

Existe la posibilidad de enviar Ether de manera forzada a un Smart Contract sin utilizar ninguna función propia de dicho contrato y sin usar ninguna función `payable`.

- Con la función `selfdestruct`.
- Precargando el contrato de Ether a partir del conocimiento previo de la dirección del contrato.

El hecho de que se pueda explotar la vulnerabilidad se debe a un mal uso de `address(this).balance`.

- **Técnicas Preventivas:**

- La lógica del Smart Contract debe evitar, siempre que sea posible, depender de valores exactos del saldo que aloja el contrato. Si no se puede evitar, hay que tener en cuenta que al utilizar `address(this).balance` su aplicación debe hacer frente a saldos inesperados.
- Utilizar una variable definida que gestione el Ether depositado en el contrato, es decir, que se incremente a partir de las funciones `payable`. Esta variable no responderá ante Ether enviado de manera forzada.



Ether inesperado

- **Contrato vulnerable:** [Crowdfunding.sol](#)

Smart Contract mediante el cual los inversores aportan una cierta cantidad de Ether. Una vez se ha alcanzado el objetivo marcado, el contrato permite sacar la cantidad aportada a los inversores.

- **Contrato atacante:** [EtherSender.sol](#)

Smart Contract que mediante la función `selfdestruct(target)` envía la cantidad de Ether que se aloja en el mismo al contrato objetivo. Este contrato consta de dos métodos análogos para explotar la vulnerabilidad.

- **Contrato securizado:** [CrowdfundingSec.sol](#)

Smart Contract mediante el cual los inversores aportan una cierta cantidad de Ether. En este caso no se hace uso de `address(this).balance`, sino que se crea una nueva variable que gestiona el Ether aportado por los inversores.



Tx.origin

- **Vulnerabilidad:**

El uso de `tx.origin` para la autenticación en un Smart Contract provoca una vulnerabilidad, que en seguridad tradicional se podría asemejar al *phishing*.

La explotación de la vulnerabilidad pasa por hacer creer a la víctima que el contrato malicioso se trata de una EOA (*External Owned Account*) a la cual puede enviar cierta cantidad de Ether.

Un Smart Contract que utilice `tx.origin` para autorizar la ejecución de ciertas funciones es vulnerable. Para la explotación de esta vulnerabilidad se requiere de ingeniería social para engañar a la víctima.

- **Técnicas Preventivas:**

- No utilizar `tx.origin` para autorización de entrada en funciones.
- Utilizar en su lugar `msg.sender`.
- “Do NOT assume that `tx.origin` will continue to be usable or meaningful.” - Vitalik Buterin

- <https://ethereum.stackexchange.com/questions/196/how-do-i-make-my-dapp-serenity-proof/200#200>



Tx.origin

- **Contrato vulnerable:** [Victim.sol](#)

Smart Contract que permite obtener Ether a partir de la función `fallback` en el cual idealmente sólo el propietario del contrato es capaz de sacar los fondos alojados en el mismo.

- **Contrato atacante:** [MyWallet.sol](#)

- **Contrato atacante alternativo:** [MyWalletExternal.sol](#)

Smart Contract mediante el cual la víctima, al realizar una transacción hacia la dirección de este contrato, provocará que se invoque la función `fallback` llamando a la función de retirada de fondos de su contrato.

Al ser la víctima quien inicia la transacción, la dirección que inició la llamada de la función por primera vez es la víctima, por lo que `tx.origin` será la dirección de la víctima y la condición de entrada (autorización) será válida para la retirada de fondos, que serán recibidos por el contrato malicioso.



¡Gracias por venir!

Ya seas nuevo en todo esto, o hayas estado con nosotros desde Génesis, **muchas gracias por todo el apoyo mostrado y las contribuciones realizadas.**

¡Estamos emocionados de seguir construyendo el ecosistema de Ethereum juntos!

