Análisis de la seguridad de los smart contracts

Roi Rodríguez Huertas

Máster Universitario en Ciberseguridad y Privacidad
TFM en Sistemas de Blockchain - 2022-23 Sem.1

Profesor colaborador: **Friman Sanchez Castaño**

Resumen

La tecnología *Blockchain* lleva unos cuantos años entre nosotros y cada vez su utilidad es más clara para el público general. Uno de los puntos de inflexión vino con la introducción de los contratos inteligentes, "*smart contracts*" en inglés, que corren bajo esta red y que permiten la ejecución de aplicaciones de forma descentralizada sin la necesidad de un servidor central. Las características de los *smart contracts* los hacen propensos a fallos graves de seguridad con consecuencias desastrosas si no se siguen una serie de pautas y buenas prácticas.

Para ello se plantea un análisis en profundidad del estado de la seguridad de los *smart contracts*, revisando los fallos de seguridad más comunes, sus causas y soluciones. Además se proporcionarán ejemplos con código que ilustren recomendaciones y consejos para conseguir un *smart contract* seguro. Para ello se consultará la literatura disponible con el fin de estudiarla y condensar en este trabajo la información actualizada y más relevante.

Abstract

The Blockchain technology has been arround us a few years now. Their utility has been demostrated to the general public. One of the key changes has been the introduction of smart contracts, programs that work under the Blockchain by themselves without the need of a central server. The properties of smart contracts makes them susceptible to serious security problems with distastrous consequences if some rules and good practices are not respected.

This work provides an thorought analysis of the state of the security of smart contracts. The most common fails will be rewieved, among their causes and solutions. Furthermore examples will be provided that show recommendations to archieve a safe development of smart contracts. To archieve this, previous work will be explored, studied providing the reader the most useful and updated information.

Contenido

1. Introducción....6
1.1 Contexto y justificación del trabajo....6
1.2 Estado del arte....7
1.3 Objetivos....7
1.4 Metodología....8
1.5 Planificación....9
1.6 Impacto en ético, social y ambiental....12
2. Introducción a la Blockchain....15
2.1 Definición y Funcionamiento....15
2.2 Utilidad....15
2.3 Plataformas Blockchain....16
2.4 Plataformas compatibles con Smart Contracts....18
3. Smart contracts....20
3.1 Smart contracts en Ethereum....20
3.2 Programación....20
3.2.1 Entorno de desarrollo....21
3.2.2 Ejemplo de *smart contract*....21
3.2.3 Modificadores de visibilidad....22
3.2.4 Modificadores de función....22
3.2.5 Modificadores arbitrarios....23
3.2.6 Librerías externas y delegate call....24
3.3 Pruebas y despliegue....24
3.4 Generar una llamada a una función....25
3.5 Ejecución de *smart contracts*....27
3.6 Ethereum vs otras tecnologías....29
4. Entorno de trabajo....31
4.1 Equipo....31
4.2 Software....31
4.3 Redes utilizadas....33
4.4 Repositorio de código....33
5. Vulnerabilidades en los *smart contracts*....34
5.1 Desbordamiento aritmético (*Underflow*/*Overflow*)....35
5.1.1 La vulnerabilidad....35
5.1.2 Ocurrencias vulnerabilidad....36
5.1.3 Solución....38
5.2 Reentrada....40
5.2.1 La vulnerabilidad....40
5.2.2 Vulnerabilidades producidas....42
5.2.3 Soluciones....45
5.3 Control de Acceso I: modificadores visibilidad....46
5.3.1 La vulnerabilidad....46
5.3.2 Ocurrencias vulnerabilidad....46
5.3.3 Soluciones....48

5.4 Control de Acceso II: librerías y delegatecall....50
5.4.1 La vulnerabilidad....50
5.4.2 Ocurrencias vulnerabilidad....50
5.4.3 Soluciones....52
5.5 Uso incorrecto llamadas bajo nivel....54
5.5.1 La vulnerabilidad....54
5.5.2 Ocurrencias vulnerabilidad....54
5.5.3 Soluciones....55
5.6 Mala aleatoriedad....58
5.6.1 La vulnerabilidad....58
5.6.2 Ocurrencias vulnerabilidad....59
5.6.3 Soluciones....60
5.7 Denegación de servicio....61
5.7.1 La vulnerabilidad....61
5.7.2 Ocurrencias vulnerabilidad....62
5.7.3 Soluciones....62
5.8 Vulnerabilidades en el cliente....64
5.8.1 La vulnerabilidad....64
5.8.2 Ocurrencias vulnerabilidad....64
5.8.3 Posible ataque (*short address)*....67
5.8.4 Soluciones....68
5.9 Ether no esperado....70
5.9.1 La vulnerabilidad....70
5.9.2 Ocurrencias vulnerabilidad....70
5.9.3 Soluciones....71
5.10 Front running o condición de carrera....72
5.10.1 La vulnerabilidad....72
5.10.2 Ocurrencias vulnerabilidad....72
5.10.3 Soluciones....74
5.11 Manipulación de tiempo....76
5.11.1 La vulnerabilidad....76
5.11.2 Ocurrencias vulnerabilidad....76
5.11.3 Soluciones....77
5.12 Declaración incorrecta de variables....78
5.12.1 La vulnerabilidad....78
5.12.2 Ocurrencias vulnerabilidad....80
5.12.3 Soluciones....80
6. Desarrollo seguro de smart contracts....82
6.1 Ejemplos de programación segura....82
6.1.1 Versión actualizada del compilador....82
6.1.2 Llamadas externas – evitar ataques reentrada....83
6.1.3 Envío de Ether....85
6.1.4 Solucción a *front-running,* patrón "Commit-Reveal"....90
6.1.5 Bucles y borrado de arrays....93
6.1.6 Manejo de errores....95
6.2 Pruebas de *smart contracts*....99

6.2.1 Entorno de pruebas....99
6.2.2 Pruebas unitarias y de integración....100
6.2.3 Ejemplo prueba unitaria con Brownie....100
6.2.4 Red de pruebas públicas....102
6.3 Herramientas de prueba automatizadas....104
6.3.1 Remix....104
6.3.2 Mythril....105
6.3.3 Slither....106
6.3.4 Oyente....107
7. Conclusiones y trabajo futuro....108
7.1 Trabajo futuro....109
Anexo I: repositorio de código....110
Bibliografía....111

1. Introducción

1.1 Contexto y justificación del trabajo

Las tecnologías *Blockchain* causaron gran expectativa en el público general tras la el lanzamiento de Bitcoin en 2008. Sin embargo, no fue hasta el lanzamiento de Ethereum unos años más tarde en 2015 donde se revolucionó también el mundo de la computación con la llegada de los *smart contracts*.

Los *smart contracts* son programas con una serie de reglas inmutables que son ejecutados en una red distribuida sin la necesidad de un servidor central. La posibilidad de diseñar aplicaciones complejas que utilicen los *smart contracts* como alternativa a un servidor, ofrece un abanico nuevo de posibilidades a los desarrolladores.

Sin embargo algunas de las características de los *smart contracts* que los hacen atractivos, también los convierte en potencialmente peligrosos en caso de que la lógica programada en ellos contenga algún error o vulnerabilidad. En la mayor parte de las veces las *Blockchain* sobre la que funcionan estos programas son de acceso público, lo que permite que cualquier usuario con acceso a internet pueda interactuar con el *smart contract*. Otra de las principales características de los *smart contracts* es que una vez desplegados en la red, son inmutables y no se pueden modificar, lo que imposibilita corregir los errores. Además la mayor parte de los *smart contracts* están centrados en aplicaciones que implican transacciones monetarias, lo que las hace especialmente atractivas para hackers maliciosos.

Un símil para darnos cuenta de los retos de seguridad que supone un *smart contract*, sería como si una aplicación interna de una entidad bancaria destinada a manejar las transacciones de los clientes, expusiese de forma pública su interfaz de uso así como todos sus datos y además, no presentase ningún método directo para arreglar alguna vulnerabilidad si la hubiera. Parece inverosímil pero existen cientos sino miles de *smart contracts* funcionando en la actualidad con una funcionalidad muy similar.

Es por ello que la seguridad debe ser uno de focos principales durante el desarrollo de los *smart contracts*. Si no es así, ocurren vulnerabilidades y ataques como los que se registran todos los años con pérdidas multimillonarias tanto para los dueños como para los usuarios. Tan solo en el año 2021, más de 680 Millones de dólares fueron perdidos a causa de vulnerabilidades que provocaron el robo de los fondos o la pérdida de los mismos al quedar bloqueados en cuentas inaccesibles[1].

En este trabajo se pretende identificar qué vulnerabilidades que afectan a los *smart contracts,* el por qué, y como evitar que se produzcan. Para ello se

revisarán los ataques producidos en el pasado para descubrir cuales fueron los errores cometidos por los desarrolladores, y cuales fueron las características técnicas de los *smart contracts* que permitieron estos fallos. Con este conocimiento se propondrán un conjunto de medidas a aplicar que ayuden a los desarrolladores a evitar cometer estos fallos. Para ello se revisarán las herramientas disponibles que ayuden a identificar posibles vulnerabilidades durante el desarrollo, así como proponer algún caso de ejemplo que demuestre la implementación de estas técnicas.

1.2 Estado del arte

Se ha realizado una pequeña revisión de trabajos publicados en el área de la seguridad en los *smart contracts* para comprobar que nivel de conocimiento y de investigación existe en este momento. La seguridad en el campo de la informática está cada vez más presente dada su importancia, y aunque hay numerosas fuentes sobre la seguridad en los *smart contracts*, no es un campo tan explorado como otros más establecidos en la informática como puede ser la web.

Una de las fuentes más relevante a la hora de realizar este TFM será la publicación *An Analysis of Smart Contracts Security Threats Alongside Existing Solutions[2]*, que servirá como punto de partida para este TFM. En el artículo, se presentan tipos de vulnerabilidades, ataques más relevantes hasta el momento y herramientas para la seguridad de los smart contracts, temas que serán clave en este TFM. A pesar de un artículo reciente (2019), estamos en un área que está creciendo rápidamente por lo que es clave revisar los resultados y compararlos a los obtenidos en este TFM, ya que estado de la seguridad que puede diferir tras el paso de unos pocos años tras la evolución del lenguaje, descubrimiento de nuevas vulnerabilidades y desarrollo de nuevas herramientas de análisis.

Sin embargo esta no es la única fuente ya que hay numerosos estudios recientes explorando nuevas vías para analizar y mejorar la seguridad que serán utilizados en este trabajo.[3]–[7]

1.3 Objetivos

El objetivo principal de este trabajo será implementar una guía, que incluya casos prácticos, para mejorar la seguridad durante el desarrollo de *smart contracts*.

A continuación se enumeran los objetivos específicos necesarios a realizar en este TFM para alcanzar el objetivo principal:

- Descubrir que tipos de redes de bloques existen, cuales son las más utilizadas, y de estas cuales soportan el uso de contratos inteligentes.
- Analizar y entender el funcionamiento de un contrato inteligente, desde su programación hasta su ejecución.

- Identificar y describir de forma detallada qué tipos de vulnerabilidades existen que afecten a los *smart contracts*.
- Descubrir y analizar los ataques a *smart contracts* que hayan producido más impacto desde el nacimiento de la *Blockchain*.
- Conocer que recomendaciones y consideraciones han de tener los desarrolladores a la hora de producir y desplegar *smart contracts* de forma segura.
- Descubrir si existen alternativas para analizar de forma automatizada potenciales vulnerabilidades durante el desarrollo.
- Elaborar una guía, que incluya casos o código de ejemplo, para evitar producir estas vulnerabilidades y evitar sus consecuencias.
- Elaborar una *DApp* sencilla, que incluya consejos enumerados en la guía de mejores prácticas.
- Realizar una reflexión sobre el trabajo realizado y posibles trabajos futuros.

1.4 Metodología

Este TFM es un ejercicio de investigación. Como se ha detallado brevemente en el resumen, será necesario estudiar y consultar artículos y publicaciones sobre la seguridad en los *smart contracts*. Para ello se consultarán revistas y bases de datos reconocidas para de esta forma comprender el estado actual de la seguridad, y poder realizar un listado de las vulnerabilidades más importantes, así como detallar al lector que puntos son los más importantes a la hora del desarrollo y mantenimiento de un contrato inteligente.

El trabajo a realizar se divide en varias etapas necesarias para lograr los objetivos marcados:

Definición del plan de trabajo

En esta primera etapa se introduce cual es la finalidad del trabajo y se realiza un pequeño estudio del estado del arte. Se definen los objetivos a cumplir en el trabajo, la metodología a llevar a cabo, así como las tareas y la planificación de las mismas. Además se realiza un análisis del impacto ético, social y medioambiental.

Introducción a la Blockchain

En esta etapa se pretende introducir al lector a la tecnología *Blockchain*, repasando su funcionamiento, las características y los casos de uso. Además se enumerarán redes *Blockchain* existentes.

Introducción a los *smart contracts*

Se introducen los *smart contracts* de forma similar a la *Blockchain*, pero en este caso haciendo más inciso en la parte técnica en el funcionamiento así como en el desarrollo. Se introducirá el lenguaje de programación usado y los pasos necesarios en la creación de un *smart contract*.

Análisis de la seguridad de los *smart contracts*

Una vez introducidas las tecnologías, se hará un análisis exhaustivo de la seguridad de los *smart contracts*. Para ello se enumerarán las vulnerabilidades conocidas, agrupándolas en grupos según el tipo. Además se hará un análisis de los ataques pasados más relevantes.

Desarrollo seguro de los *smart contracts*

Ya conocidos los puntos débiles de los *smart contracts*, se detallaran buenas prácticas que ayuden al desarrollo de *smart contracts* seguros. Entre ellas se incluirán consejos durante el desarrollo, así como análisis de los *smart contracts* utilizando herramientas disponibles. Por último se realizará la creación de una DApp sencilla que refleje las buenas prácticas sugeridas.

Conclusiones y trabajos futuros

En esta última fase se presentan las conclusiones tras el trabajo realizado, resumiendo el estado de la seguridad actual de los *smart contracts*. Además se identificarán posibles puntos de trabajo futuros que no hayan sido explorados en este TFM.

1.5 Planificación

A continuación se presenta un listado de las tareas necesarias para cumplir los objetivos necesarios para completar este trabajo. Este listado de tareas así como la planificación correspondiente para ejecutar las mismas está sujeta a cambios según avance el trabajo realizado y la comprensión de la materia sea mayor, pudiendo incidir más en determinadas tareas que otras respecto al plan inicial, o incluso añadir nuevas tareas o prescindir de otras ya existentes.

Listado de tareas

- **Plan de trabajo**
  - Problema a resolver: detallar el problema a resolver en este TFM
  - Objetivos: Enumeración de los objetivos que se quieren alcanzar con la realización del TFM.
  - Metodología: La descripción de la metodología que se seguirá durante el desarrollo del TFM.
  - Tareas: Listado de tareas a realizar para lograr el objetivo del proyecto.
  - Planificación: Realizar un calendario de trabajo para las tareas.
  - Estado del arte: estudiar la bibliografía publicada sobre el tema que servirá como punto de partida del trabajo.

- Impacto: estudio del impacto ético social y medioambiental.
- Revisión: Revisión previa al a entrega de seguimiento.
- HITO: Entrega seguimiento Plan de trabajo PEC1

- **Introducción a la Blockchain**
  - Blockchain: definición de la *Blockchain*, su funcionamiento y usos.
  - Smart contracts: definición de *smart contract* y utilidad.
  - Blockchains más comunes: listado de *Blockchain* con más cuota de mercado que soporten *smart contracts*.
- **Smart contracts**
  - Tipos: que tipos de *smart contracts* contracts existen si hay varios.
  - Funcionamiento: como funciona un *smart contract* en detalle
  - Programación: como se desarrolla un *smart contract*
  - Despliegue: como se envía un *smart contract* a la *Blockchain* para su uso.
  - Otros: detalles relevantes no incluidos en otros apartados.
  - HITO: Entrega seguimiento PEC2
- **Vulnerabilidades en los *smart contracts***
  - Vulnerabilidades: Analizar que tipos de vulnerabilidad existen
  - Ataques: Analizar los ataques a *smart contracts* producidos hasta el momento
  - Soluciones: Describir algunas de las soluciones a las vulnerabilidades. Análisis más detallado en el siguiente apartado para casos más relevantes.
- **Desarrollo seguro de *smart contracts***
  - Soluciones: Detallar prácticas de desarrollo que eviten vulnerabilidades específicas a *smart contracts* y vulnerabilidades más comunes de programación.
    - Se un repositorio de código con ejemplos de contratos con malas prácticas y la implementación de las soluciones recomendadas.
  - Pruebas de *smart contracts*: Explorar las herramientas y procedimientos necesarios para realizar pruebas de forma correcta en los *smart contracts*.
  - Herramientas: Exploración de herramientas que permitan encontrar vulnerabilidades durante el desarrollo
  - HITO: Entrega seguimiento PEC3
- **Conclusiones**
  - Conclusiones: apartado para reflexionar sobre lo aprendido en este trabajo
  - Trabajos futuros: establecer posibles puntos para extender el mismo en un futuro.
- **Maquetación trabajo**
  - Plantilla: maquetación de los estilos para la plantilla del documento.
  - Integración: Integrar las entregas de seguimiento en el documento final.

- Revisión: comprobar que el documento sea correcto y coherente, respetando las normativas y recomendaciones aportadas por la UOC.
- HITO: Entrega seguimiento PEC4

- **Presentación vídeo**
  - Diagrama: Realizar un esquema de los puntos más importantes a tratar.
  - Diapositivas: Realizar presentación con los puntos elegidos en el diagrama, y resumiendo el trabajo, de forma clara y visible para el lector final.
  - Grabación: Grabación sobre la diapositiva.
  - Revisión: Revisión y corrección de la presentación en vídeo.
  - HITO: Entrega presentación en vídeo

Hitos del proyecto:

Se establecen diferentes entregas parciales del trabajo completado hasta el momento para realizar un mejor seguimiento. Las fechas de entrega con las correspondientes horas de trabajo en cada una de ellas son las siguientes:

- Entrega 1: 28/09 al 11/10 (50h)
- Entrega 2: 12/10 al 8/11 (75h)
- Entrega 3: 9/11 al 6/12 (75h)
- Entrega 4: 7/12 al 10/01 (75h)
- Entrega 5: 11/01 al 17/01 - Presentación vídeo (20h)
- Defensa TFM: Entre 23/01 al 27/01

Planificación (Diagrama de Gantt):
Se ha utilizado el software libre Planner[1], para la planificación del proyecto mediante un diagrama de Gantt. El calendario de trabajo se ha establecido en una jornada diaria de 4 horas entre lunes y domingo exceptuando festivos. Se han calculado el trabajo necesario para completar cada una de las tareas. Ver Figura 1.

Notar que en el diagrama de Gantt generado, por error de la aplicación no distingue entre duración y trabajo. Por ejemplo si indica 1 día de trabajo, es en realidad la duración de la tarea, ya que el trabajo será 4 horas, que es la duración de una jornada. El plan del proyecto que indica 11d y 2h, equivale a 46h de trabajo.

1 https://wiki.gnome.org/action/show/Apps/Planner (accedido 11 de octubre de 2022)

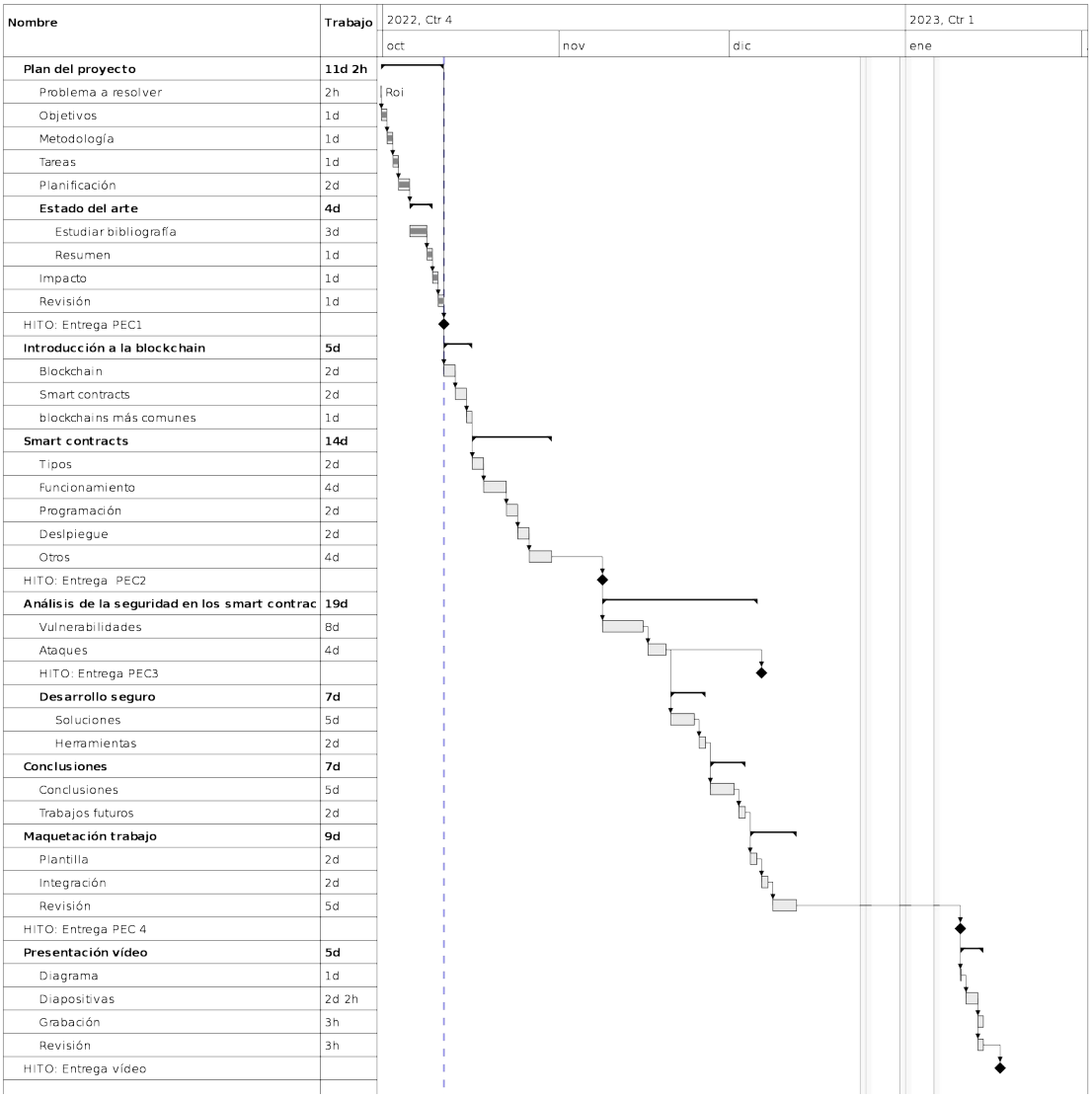

Figura 1: Diagrama de Gantt del proyecto

1.6 Impacto en ético, social y ambiental

La confección de un Trabajo de Fin de Máster conlleva la realización de numerosas tareas que pueden suponer algún tipo de impacto ético, social o ambiental. Se ha realizado un análisis para evaluar su importancia en el proyecto y conocer si se produce algún tipo de impacto positivo o negativo en alguna de las tres dimensiones.

Para ello he utilizado como referencia los ODS (Objetivos de Desarrollo Sostenible 2030, ONU[2]), los cuales proporcionan una serie de apartados a intentar cumplir para que el trabajo realizado sea sostenible.

2 https://www.un.org/sustainabledevelopment/ (accedido 8 de octubre de 2022)

En el apartado de conclusiones se hará una reflexión de si se han cumplido los objetivos marcados en esta sección.

Impacto ambiental:

Para la realización de este TFM se usará como principal herramienta un ordenador portátil o de sobremesa. Es un dispositivo que no realiza un consumo elevado de energía por lo que considero que el impacto en este aspecto es mínimo. Además no se utilizará transporte alguno gracias a las plataformas virtuales que proporciona la UOC, con el ahorro energético y en la huella de carbono que esto conlleva. El seguimiento del trabajo con el tutor se hará a través del Campus Virtual y del email.

Uno de los objetivos del TFM es dar a conocer a los desarrolladores de *smart contracts* una guía para mejorar la seguridad de los mismos, protegiendo de esta forma los recursos de la empresa.

Otros objetivos incluidos en los ODS considero que no se aplican a este proyecto como impacto en la vida acuática o terrestre.

Comportamiento ético y responsabilidad social

La ciberseguridad tiene un fuerte componente ético debido al uso que se puede dar del conocimiento en esta área. El descubrimiento de una pieza de software con una vulnerabilidad no conocida utilizado en un entorno empresarial o global posee gran valor dado el daño que puede causar esta información de caer en las manos equivocadas. Por otro lado este mismo conocimiento en buenas manos provocará que el error sea subsanado y un informe se haga público tras el hecho, de forma que los programadores y personal relacionado sea consciente del problema y se busquen soluciones para evitar volver a producir estas vulnerabilidades.

Es por ello que este trabajo tiene dos objetivos claros, identificar las vulnerabilidades presentes y pasadas en los *smart contracts* para dar información del porqué de su existencia, y por otra parte ofrecer una guía práctica que evita en la medida de lo posible que se produzcan. Si este trabajo cae en manos de una persona maliciosa no debería causar mayores estragos debido a que las vulnerabilidades aquí presentadas serán aquellas difundidas de forma pública, sin embargo para un programador este trabajo tiene la intención de servir como guía donde encontrar toda la información relevante para mejorar la seguridad en sus propios *smart contracts*. Por lo tanto considero que el impacto ético será muy positivo en este aspecto si se logran los objetivos de ese TFM.

Diversidad

El resultado de este TFM será de tipo técnico y no tendrá ningún impacto ni positivo ni negativo en aspectos de género, diversidad o derechos humanos, salvo el tipo de redacción escogida en el documento final que si puede tener cierto componente en aspectos de igualdad de género según

las expresiones utilizadas en el mismo. Para intentar reducir las desigualdades de género se han seguido ciertas pautas incluidas en el documento elaborado por el Servicio Lingüístico de la UOC, que mediante el uso de genéricos colectivos, palabras y expresiones despersonalizadas entre otros pretenden minimizar esta desigualdad en la lengua escrita.

Si que tendrá un impacto positivo en aspectos de seguridad de los datos y las TIC, por las mismas razones que se han dado en el apartado ético.

2. Introducción a la Blockchain

Para conocer a fondo los *smart contracts* y su seguridad es necesario primero entender el funcionamiento de la tecnología *Blockchain* sobre la que funcionan. En este apartado se hará una rápida revisión a las redes *Blockchain*, su funcionamiento y usos.

2.1 Definición y Funcionamiento

Una cadena de bloques, llamada *Blockchain* en inglés, es una base de datos descentralizada en forma de lista enlazada, la cual contiene una serie de bloques, los cuales están enlazados de forma segura utilizando criptografía formando una cadena. Todos los usuarios de una *Blockchain* dispondrán de una copia con la misma información, pudiendo añadir nuevos bloques a la misma, pero donde los bloques antiguos serán inmutables. Esta propiedad la hace ideal para registrar datos de forma transparente para todos los usuarios.

La primera propuesta de *Blockchain* nace del paper publicado por Satoshi Nakamoto, donde describe el funcionamiento de la moneda Bitcoin[8]. En Bitcoin, la Blockchain funciona como un libro de contabilidad, donde se registran de forma segura e inmutable, todas las transacciones realizadas hasta el momento.

2.2 Utilidad

Existen gran cantidad de redes *Blockchain* que cubren una variedad de casos de uso. Algunos de ellos son:

- Criptomonedas: Bitcoin fue la primera red que permitió realizar pagos Online de una moneda electrónica de forma descentralizada y sin la necesidad de un tercero, como una institución financiera. El término criptomoneda proviene de la naturaleza criptográfica de las redes *Blockchain*.

  Actualmente la mayoría de redes *Blockchain* giran entorno al concepto de las Criptomonedas, aunque después ofrezcan más aplicaciones.

- Bases de datos: la arquitectura de bloques de la *Blockchain* es ideal para almacenar ciertos tipos de bases de datos en las que sea necesario proporcionar integridad y no se requiera borrado de datos.

  Un ejemplo seria el almacenamiento de historiales médicos, permitiendo el acceso e intercambio de información de pacientes entre centros hospitalarios, doctores o farmacias de forma sencilla.

- Gestión de cadenas de suministro: las empresas del sector alimentario o similares pueden utilizar la tecnología *Blockchain* para rastrear los productos desde el momento que son recolectados hasta su venta al cliente final. De esta forma cada producto dispondrá de un certificado donde se pruebe por qué etapas ha pasado, facilitando realizar una trazabilidad del mismo tanto para la empresa, como para el cliente.

  Por ejemplo la empresa de supermercados Walmart ha empezado a utilizar esta tecnología desde 2017.[9]

- NFT y Tokens: Uno de los usos más mediáticos son los NTF, que proviene del inglés "Non-Fungible Tokens", es decir un token no fungible. Se tratan de tokens criptográficos que representan algo único, ya sea en el mundo real o virtual. La mayor diferencia con las Criptomonedas es que cada NFT es único e indivisible, pero si que son transferibles.[10]

- Contratos inteligentes y DApps: Los contratos inteligentes, que llamaremos a partir de ahora por su término en inglés *smart contracts*, son pequeños programas de ordenador que viven en la *blockchain* y pueden realizar acciones cuando se cumplan ciertas condiciones establecidas. Cuando un usuario interactúa con un contrato, acepta las normas establecidas en el mismo, las cuales son transparentes para todo el público.

  Las ventajas de un smart contract es que se ejecutan de forma transparente en la *blockchain* sin la necesidad de un tercero que compruebe que se cumplen las condiciones preestablecidas en el contrato. La mayoría de veces son inmutables, lo que los hace tan frágiles ante errores de programación como comprobaremos en este TFM.

- Con la aparición de los smart contracts también se introdujo el concepto de DApps o aplicaciones descentralizadas. Las DApps permiten crear una aplicación sin necesidad de un servidor centralizado, ya que será la propia blockchain la encargada de la lógica de la aplicación en el apartado del servidor.[11]

2.3 Plataformas Blockchain

Blockchains de primera generación

Después del nacimiento de Bitcoin en 2008 y su éxito posterior, aparecieron progresivamente nuevas plataformas blockchain que ofrecen un enfoque diferente a Bitcoin. Ya sea en el modo de gestionar las transacciones para hacerlas más eficientes y asequibles, por ejemplo Litecoin[12], o la solución propuesta por Monero[13], que añade una capa de privacidad de forma que no sea posible conocer el emisor real de una transacción.

Este tipo de redes de primera generación comparte una característica común y es que tan solo permiten realizar transacciones de pago con Criptomonedas.

Blockchains de segunda generación

En 2013 comenzó el desarrollo de Ethereum[14], posteriormente lanzado en 2015, y con ello las llamadas Blockchain 2.0. La novedad más llamativa que las diferencia de la anterior generación es la posibilidad de ejecutar contratos inteligentes y la creación de DApps, aunque no es la única. Ethereum también incluye sistemas de tokenización, que permiten representar activos o crear monedas dentro de la blockchain de Ethereum.

La implementación de los contratos inteligentes y el resto de funcionalidades es posible gracias a la inclusión de una máquina virtual que permite ejecutar aplicaciones que sean escritas en código Turing completo. Esto permite la creación de contratos arbitrarios para cualquier tipo de transacción o aplicación.

En definitiva con la aparición de las blockchains de segunda generación dejan de ser únicamente plataformas de pago, para en convertirse en una herramienta multifuncional con infinitas posibilidades.

Blockhains de tercera generación

Las blockchain de tercera generación vienen a solventar algunos de los problemas principales en las generaciones pasadas como la escalabilidad y el consumo de energía fruto del protocolo de consenso.

En el apartado de escalabilidad, Bitcoin permite actualmente una cantidad de 7 TPS(Transacciones por segundo) con un tiempo de confirmación de 60 minutos (el tiempo de generar 4 bloques). Comparado a un método de pago tradicional, la plataforma VISA afirma que procesa sobre 1700 TPS, esto es 242 veces más que Bitcoin. Ethereum permite 25 TPS, Litecoin 56 TPS, mientras que monedas más actuales como Cardano, permiten 250 TPS o Solana que afirma una capacidad de 29000 TPS.

Acerca del consumo de energía, muchas de las primeras redes utilizan el protocolo de consenso llamado PoW (Prueba de Trabajo, en inglés *Proof of Work*). El protocolo de consenso es necesario para determinar que nodo de la red creará el siguiente bloque. El protocolo PoW consiste en que todos los nodos de la red realicen cálculos de hash aleatorios hasta que encuentren uno compatible con los bloques creados anteriormente. Esto provoca que todos los nodos estén desperdiciando energía realizando cálculos de hash mediante GPUs o unidades de procesamiento paralelo. Se calcula que Bitcoin consume 204 TWh por año.

Una de las soluciones es emplear el protocolo PoS (Prueba de Participación, en inglés *Proof of Stake*). En este protocolo se elige al nodo que va a generar el siguiente bloque mediante una selección aleatoria entre todos

los disponibles, pero para que un nodo pueda participar debe cumplir una serie de requisitos como disponer de un número determinado de monedas reservados en la red, de forma que pueda perderlos si intenta crear bloques incorrectos. Ethereum realizó el cambio de PoW a PoS en Septiembre de 2022, lo que supuso una reducción del consumo que pasó de ser de 80 TWh/año a menos de 0.01TWh/año.[15]

2.4 Plataformas compatibles con Smart Contracts

Una vez conocidas las diferentes generaciones de plataformas *blockchain* existentes así como sus características principales, haremos una selección de las más relevantes que permitan la ejecución de Smart Contracts.

Para realizar la selección, se han utilizado dos criterios diferentes que ayuden a determinar el tamaño de la comunidad de usuarios. El primero es el valor de capitalización de mercado de la criptomoneda, tan solo aplicable a las blockchain de acceso público.

Un segundo criterio que toma como referencia el trabajo realizado en [16], donde se utiliza la plataforma StackOverflow para comprobar el número de preguntas realizadas por los usuarios bajo el "*tag*" de cada blockchain. Se entiende que a mayor número de preguntas, mayor será la comunidad de desarrolladores en esa plataforma, lo cual es un buen indicador para las plataformas privadas.

En la Tabla 1 se recogen los resultados obtenidos para cada criterio, incluyendo el puesto en CoinMarketCap y el número de *tags* y *tag* empleado para los resultados en StackOverflow. Además se incluyen otros detalles como el tipo de acceso, el tipo de lenguaje, el protocolo de consenso, así como los lenguajes de programación soportados por cada plataforma.

Se puede comprobar que Ethereum desde su nacimiento sigue siendo la plataforma más utilizada bajo los dos criterios, y que la segunda plataforma es Hyperledger Fabric, la cual tiene un gran uso en el sector privado. A esta última le siguen de cerca Corda y Solana.

Es por ello que para acotar un poco el trabajo nos centraremos principalmente en Ethereum, aunque cabe mencionar que otras plataformas comparte los mismos problemas y soluciones que los planteados en este trabajo.

Red	Coin Market Cap[3]	# Tags[3]	Tipo Acceso	Tipo Lenguaje	Tipo Consenso	Lenguajes Programación
Ethereum	2º	6310 [ethereum]	Público	Touring completo	PoS	Solidity, Vyper y otros
Cardano	8º	79 [cardano]	Público	Touring completo	PoS	Plutus, Marlowe, Glow
Solana	9º	1121 [solana]	Público	Touring completo	PoH (Proof of History) y PoS	Rust, C, C++
Cosmos	22º	46 [cosmos-sdk]	Público	Touring completo	PoS	Ethermint (Solidity comp), Secure EcmaScript (Similar JS), Pact
Near Protocol	29º	593 [nearprotocol]	Público	Touring completo	PoS	JS, Rust, AssemblyScript
Waves	89º	89 [wavesplatafor m]	Público	Touring incompleto	LPoS (Leased Proof of Stake)	RIDE
Nem	96º	14 [nem]	Público y Privado	Touring incompleto	PoI (Proof of Importance, variante de PoS)	llamadas API desde cualquier lenguaje
RootStock	315º	71 [rsk]	Público	Touring completo	PoW	Solidity
Hyperledger fabric		6131 [hyperledger-fabric]	Privada	Touring completo	PBFT (Practical Byzantine Fault Tolerance)	Java, Kotlin
Corda		2296 [corda]	Privada	Touring completo	Raft	Java, Kotlin

Tabla 1: Plataformas blockchain compatibles con smart contracts

3 Datos obtenidos el 4 de Noviembre de 2022

3. Smart contracts

Ya introducido el concepto de *blockchain* y las diferentes plataformas disponibles, en este apartado profundizaremos más en el funcionamiento de los *smart contracts*.

3.1 Smart contracts en Ethereum

Los *smart contracts* en Ethereum son un tipo de cuenta Ethereum. Es decir, tienen un saldo y pueden ser objetivo de transacciones en la red, aunque sin embargo, no están controlados por ningún usuario sino que funcionan de la manera que están programadas.

Cualquier usuario puede interactuar con un contrato inteligente disponible en la red mediante transacciones que ejecuten una de las funciones definidas en el mismo.

A pesar de ser autónomos y que cualquier usuario pueda interactuar con el *smart contract*, dependerá de la lógica implementada en el mismo las tareas que se pueden realizar en el. Por ejemplo, el creador del *smart contract* puede añadir lógica que le permita retirar saldo del contrato, así como restringir el uso del contrato a ciertos números de cuenta Ethereum.

Los *smart contracts* se ejecutan en la máquina virtual de Ethereum (EVM). La EVM está basada en una pila donde se almacenan los datos en palabras de 256bits en formato Big-Endian. Una de las características más relevantes es que la EVM es Turing completa, esto es, que puede codificar cualquier cálculo y lógica que pueda llevarse a cabo e implementar bucles.

3.2 Programación

En la mayoría de *blockchain* los *smart contract* se ejecutan en un tipo de máquina virtual. Dicha máquina posee un conjunto de instrucciones de los que se compone el lenguaje de bajo nivel. Los contratos en Ethereum se componen de una lista comprendida de estas instrucciones que se ejecutan una tras otra siguiendo la lógica definida en las mismas, llamado *bytecode*.

Sin embargo, como cualquier lenguaje moderno, se utiliza un lenguaje de alto nivel para facilitar la programación. En Ethereum, este lenguaje de alto nivel es Solidity, y tiene aspectos similares a C++ y además es orientado a objetos.

Este lenguaje al ser compilado se convierte en el *bytecode* interpretable por la EVM y además también será generado una interfaz binaria de aplicación (ABI). La ABI indica las funciones disponibles en el contrato, cómo realizar llamadas a estas funciones y también cómo decodificar el resultado de las llamadas.

Notar que el lenguaje *bytecode* de la EVM no soporta funciones o métodos directamente, sino que estos son introducidos por lenguajes de más alto nivel como Solidity, que con la ayuda de la interfaz ABI indican cómo llamar a estas funciones incluidas en el *bytcode*, tal y como se explica en el apartado 3.4.

3.2.1 Entorno de desarrollo

Es recomendable utilizar un entorno de desarrollo para la programación de contratos inteligentes. Para programar en Solidity existe un entorno online llamado Remix[4]. Remix contiene un editor de código, editor de scripts para despliegue, compilador y opciones de despliegue tanto en una red *blockchain* virtual creada en el propio navegador web, en una las redes de prueba de Ethereum, o incluso en la red oficial.

También existe la posibilidad de utilizar un entorno local de desarrollo como Truffle[5], que de forma similar a Remix contiene todas las herramientas necesarias de compilación, despliegue y posibilidad de crear redes *blockchain* locales para realizar pruebas.

3.2.2 Ejemplo de *smart contract*

A continuación se muestra un pequeño *smart contract* de ejemplo incluido en Remix por defecto. El *smart contract* permite almacenar un número y leer ese mismo número.

```
contract Storage {

    uint256 number;

    /**
     * @dev Store value in variable
     * @param num value to store
     */
    function store(uint256 num) public {
        number = num;
    }

    /**
     * @dev Return value
     * @return value of 'number'
     */
    function retrieve() public view returns (uint256){
        return number;
    }
}
```

Figura 2: Contrato de ejemplo "Storage"

4 https://remix.ethereum.org
5 https://trufflesuite.com/

En el contrato hay una variable llamada *number*, que almacena un entero de 256 bits, y dos funciones, *store*, que permite reemplazar el número almacenado, y *retrieve*, que permite obtener el número almacenado.

3.2.3 Modificadores de visibilidad

En verde se pueden observar los modificadores de las funciones. En este caso las dos funciones tienen un modificador de visibilidad público. Es obligatorio indicar la visibilidad de las funciones, mientras que las variables tendrán por defecto modificador *internal*.

Existen cuatro modificadores de visibilidad aplicables a funciones y variables de estado:

- ***public***: Las funciones y variables pueden ser accedidas por cualquier parte desde dentro y fuera del contrato.

  Si una variable tiene modificador public, Solidity crea un *getter* automáticamente. Por ejemplo la función *retrieve* sería innecesaria si se declara *number* como *public*.

- ***private***: las funciones y variables privadas solo pueden ser accedidas desde el propio contrato que las declara, y las funciones no pueden ser heredadas por otras funciones.

  Hay que tener en cuenta que declarar una función o variable privada no la hace invisible en la *blockchain*, tan solo inaccesible desde otro contrato.

- ***internal***: de forma similar a *private*, pero en este caso se permite la herencia de funciones o variables.

- ***external***: solo aplicable a funciones. Solo pueden llamarse fuera del contrato donde han sido declaradas y no de forma interna (desde otra función del propio contrato por ejemplo).

3.2.4 Modificadores de función

Además de los modificadores de visibilidad existen otros adicionales:

- ***view***: son funciones que no alteran el estado de la *blockchain*, aunque pueden leerlo.

- ***pure***: son funciones que tan solo dependen de los argumentos y en ningún caso interactúan con el estado de la *blockchain*. Estas funciones son útiles para unir código común a varias funciones y estructurar el código de forma más clara.

- ***payable***: necesario para funciones que reciban o envíen criptomonedas.

Dado que las funciones *view* y *pure* no modifican el estado de la *blockchain* al ejecutarse, no consumen "gas", es decir, no hay que pagar una comisión

en la red por ejecutarlas, ya que tan solo se ejecutarán en el nodo al que estemos conectados sin afectar a la red.

El resto de llamadas a funciones modifican el estado de la *blockchain*, lo que requiere que se realice una transacción para completar la llamada y de esta forma aplicar una comisión por instrucción ejecutada.

Por lo tanto diferenciaremos entre llamadas a funciones (sin coste) y transacciones (con coste).

3.2.5 Modificadores arbitrarios

Por último es posible declarar modificadores arbitrarios en el propio contrato. Estos modificadores consisten en código que se ejecutara antes o después de la función a modificar, normalmente funciones llamadas a la función *require* para comprobar datos antes de ejecutar la función.

La ventaja de los modificadores es poder aplicarlos en varias funciones al mismo tiempo, reduciendo la duplicidad de código.

En la Figura 3 se observa una modificación del contrato previo añadiendo dos modificadores "onlyowner" y "limite100". El primero comprueba que la cuenta que interactúa con el contrato es la definida, y el segundo comprueba que el valor máximo es de 100.

```
function store(uint256 num) public limite100(num) onlyowner {
    number = num;
}

address constant Owner = 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4;

modifier onlyowner {
    require(msg.sender == Owner, 'only owner allowed to modify the contract');
    _; //Aqui va la función a modificar
}

modifier limite100(uint256 num) {
    require(num<=100, 'valor maximo es 100');
    _; //Aqui va la función a modificar
}
```

Figura 3: Modificadores de función

A continuación en la Figura 4, se observa como quedaría la función "store" tras aplicar los modificadores. Con el carácter "_", se indica en que parte de la función se añade el modificador, si al final o al principio.

```
function store(uint256 num) public{
    require(num<=100, 'valor maximo es 100');
    require(msg.sender == Owner, 'only owner allowed to modify the contract');
    number = num;
}
```

Figura 4: Modificadores de función

3.2.6 Librerías externas y delegate call

En Ethereum los contratos compilados tienen un determinado tamaño según la complejidad del mismo. A mayor tamaño también lo es la comisión a pagar para desplegar el contrato. Una manera de reducir esta comisión, es la de combinar funcionalidad compartida en un solo lugar, llamado contrato librería.

Para usar la librería es necesario desplegar esta primero, y referenciar a la misma desde el contrato donde se quieran utilizar las funciones. Aunque otra opción también es incluir el código de la librería en el contrato desplegado.

En el mundo de la computación tradicional sería el equivalente de librerías dinámicas o estáticas. En las librerías estáticas el fin es reutilizar código ya escrito y revisado, reduciendo la posibilidad de añadir errores de programación, mientras que en las dinámicas tienen como añadido reducir el espacio en el despliegue, con el ahorro de comisiones de la red que conlleva si es un contrato que tiene como objetivo desplegarse múltiples veces. Por ejemplo un contracto que actúe como wallet para los usuarios.

3.3 Pruebas y despliegue

Una vez creado el contrato, será necesario probarlo exhaustivamente. Una de las técnicas a aplicar es la realización de pruebas funcionales que se centran sobretodo en las pruebas unitarias y pruebas de integración. Es recomendable escribir test automáticos a medida que avanza el desarrollo. Existen entornos de desarrollo que soportan la ejecución, algunos de ellos son Remix, Truffle, Hardhat y Brownie.

Además de ejecutar test permiten hacer un despliegue en una red virtual o en una de las redes de prueba de Ethereum para comprobar la funcionalidad, ya que hacerlo en la red principal supone un coste además del peligro de perder fondos si hay un error que subsanar.

Aunque existen patrones para actualizar los *smart contracts*, por defecto son inmutables, lo que también otorga garantías de su cumplimiento a los usuarios de los mismos.

Para el despliegue de un *smart contract* se utiliza normalmente un programa que contenga los pasos necesarios. Es típico programarlos en JavaScript

con ayuda de la librería Web3JS, que se encarga de interactuar con la cartera del usuario que despliega el contrato y con la red *blockchain*.

El despliegue consiste en enviar una transacción firmada como otra cualquiera, donde se incluye el *bytecode*, es decir el contrato compilado. Al ejecutarse correctamente la transacción se generará una cuenta de Ethereum correspondiente al contrato recién creado.

3.4 Generar una llamada a una función

La interacción con un contrato inteligente se suele hacer a través de una DApp, que no es otra cosa que una aplicación que interactúa con un *smart contract*. Normalmente están programados en lenguaje web, de forma similar al script de despliegue también se puede usar JS y Web3JS para interactuar con un contrato inteligente.

Sin embargo profundizaremos más para conocer cómo estas librerías realizan las peticiones al *smart contract*. Para entender este proceso, definiremos el siguiente contrato de ejemplo con dos sencillas funciones. En este caso son dos funciones *pure*, por lo que no interactúan con el estado de la red, aunque en caso de hacerlo el funcionamiento sería similar.

```
contract testcontract{

    function suma(uint256 x, uint256 y) public pure returns(uint256){
        return x+y;
    }
    function resta(uint256 x, uint256 y) public pure returns(uint256){
        return x-y;
    }
}
```

Figura 5: Contrato suma

El ABI generado por este contrato es el siguiente:

```
[{
      "inputs":
            [{"internalType":"uint256","name":"x","type":"uint256"},
             {"internalType":"uint256","name":"y","type":"uint256"}],
      "name":"resta",
      "outputs":[{"internalType":"uint256","name":"","type":"uint256"}],
      "stateMutability":"pure",
      "type":"function"
      },
 {
      "inputs":
            [{"internalType":"uint256","name":"x","type":"uint256"},
            {"internalType":"uint256","name":"y","type":"uint256"}],
      "name":"suma",
      "outputs":[{"internalType":"uint256","name":"","type":"uint256"}],
      "stateMutability":"pure",
      "type":"function"
}]
```

Código 1: ABI del contrato

Como ya se ha descrito antes, el ABI es la interfaz del contrato e incluye todo lo necesario para generar una llamada a una función y obtener su resultado sin necesidad de conocer el código fuente.

En este caso se describen las dos funciones, resta y suma, cada una con dos parámetros con nombre "x" e "y", del tipo *uint256*. Como salida devuelve otro entero sin signo *uint256*.

La transacción de llamada a una función contiene los siguientes datos:

- Id del método: se genera utilizando los primeros 4 bytes del hash Keccak256[6] de la signatura de la función.

  Por ejemplo, para la función suma, la signatura de la función tiene la siguiente forma "suma(uint256,uint256)", a esta cadena le aplicamos el hash Keccak256 y extrayendo los primeros 4 bytes se obtiene el valor 0xb1b45b1b.

- Parámetros: En caso de contener parámetros, se concatenarían después del Id del método en formato binario.

- Por ejemplo, si los parámetros son los números 15 y 25, se codificarían en formato hexadecimal con los valores 0xF y 0x19 respectivamente. A mayores se añade un *padding* de 0 hasta completar 256 bits en caso de los enteros. Otros tipos de datos tienen otras reglas aunque siempre son múltiplos de 32bytes.[7] Los dos parámetros quedarían de la siguiente forma:

```
0x000000000000000000000000000000000000000000000000000000000000000F
0x0000000000000000000000000000000000000000000000000000000000000019
```

6 https://cryptobook.nakov.com/cryptographic-hash-functions/secure-hash-algorithms
7 https://docs.soliditylang.org/en/develop/abi-spec.html]

Si juntamos el Id y los parámetros, la llamada a la función suma(15,25) sería la siguiente:

```
0xb1b45b1b0000000000000000000000000000000000000000000000000000000000000F0
0000000000000000000000000000000000000000000000000000000000000019
```

Por último el "*return value*" consiste en seguir el proceso inverso de codificar un parámetro. En este caso la función devuelve un entero sin signo con el valor:

```
"0x0000000000000000000000000000000000000000000000000000000000000028"
```

Lo que corresponde en decimal al número 40.

3.5 Ejecución de *smart contracts*

Una vez conocido los pasos necesarios para la creación de un *smart contract* y su despliegue, así de cómo interactuar con un contrato inteligente, se va a analizar el proceso de ejecución a nivel de red *blockchain*.

A continuación se muestra un pequeño esquema con la estructura de los participantes en una red *blockchain* (Figura 6). En este caso la red se compone de varios nodos conectados entre ellos Peer to Peer (P2P).

Si un cliente como el dispositivo mostrado desea interactuar con la red, ha de hacerlo a través de un nodo. Si además desea realizar transacciones es necesario su cartera y claves para poder firmar las mismas.

También se muestra otro diagrama en la Figura 7, que representa una llamada a una función que no modifica el estado de la *blockchain*. El usuario crea la transacción, la envía al nodo que la ejecuta directamente en la EVM. Si necesita obtener algún estado lo hace de la *blockchain* actual.

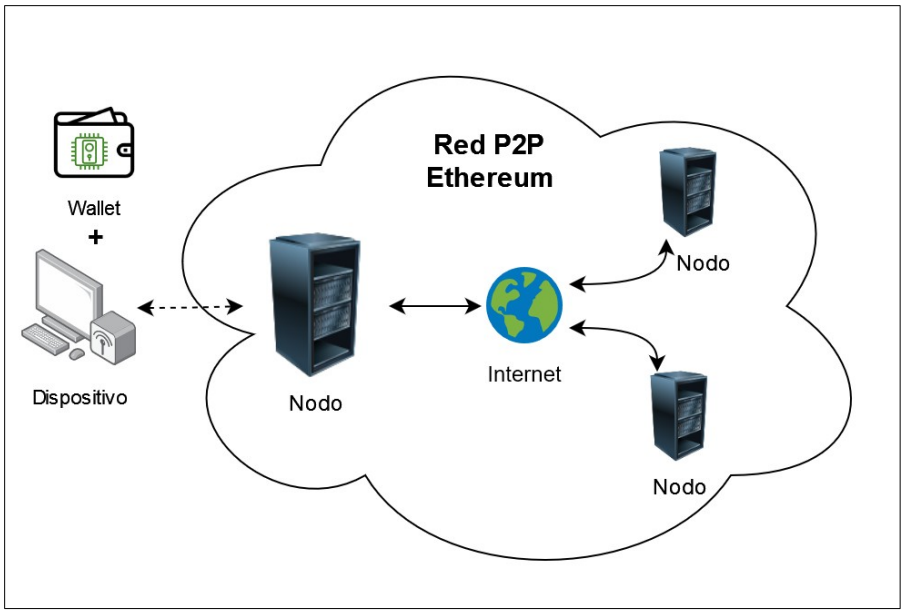

Figura 6: Estructura red Blockchain P2P

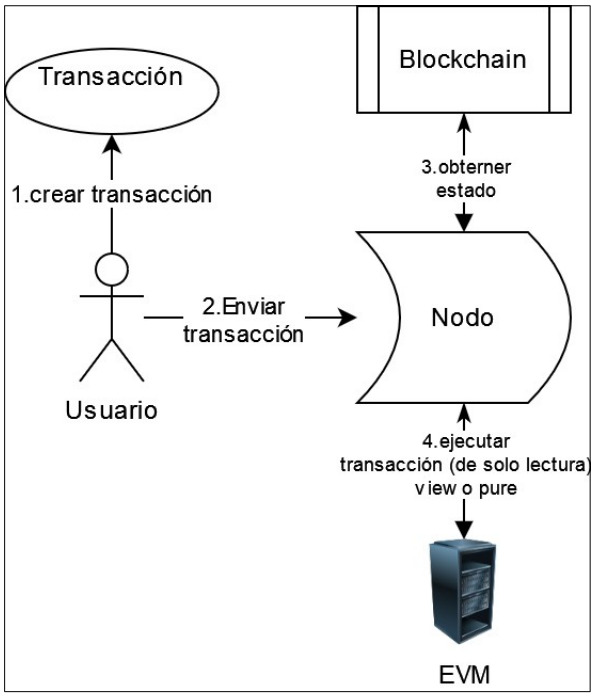

Figura 7: Llamada a función view o pure

Si se realiza una llamada que modifique el estado de la *blockchain*, es necesario crear una transacción firmada, que además supone un coste, en caso de Ethereum este coste se mide en "gas". Figura 8

De forma similar al caso anterior la petición se envía a un nodo cualquiera, pero en este caso al ser una transacción se almacena temporalmente en la "*mempool*", donde cualquier nodo validador pueda obtenerla. Cuando un

nodo cree un bloque, (en Ethereum desde la entrada del PoS esto se elige al azar), este será el encargado de procesar la petición en la EVM, y generar un bloque con el nuevo estado de la *blockchain*, el cual se concatena a los anteriores.

Posteriormente el resto de nodos en la red validan el nuevo bloque repitiendo los pasos que ha realizado el nodo validador, esto es, todos ejecutan el contrato y comprueban que el bloque obtenido es el mismo. En caso contrario se descartaría el nuevo bloque creado por ser inválido.

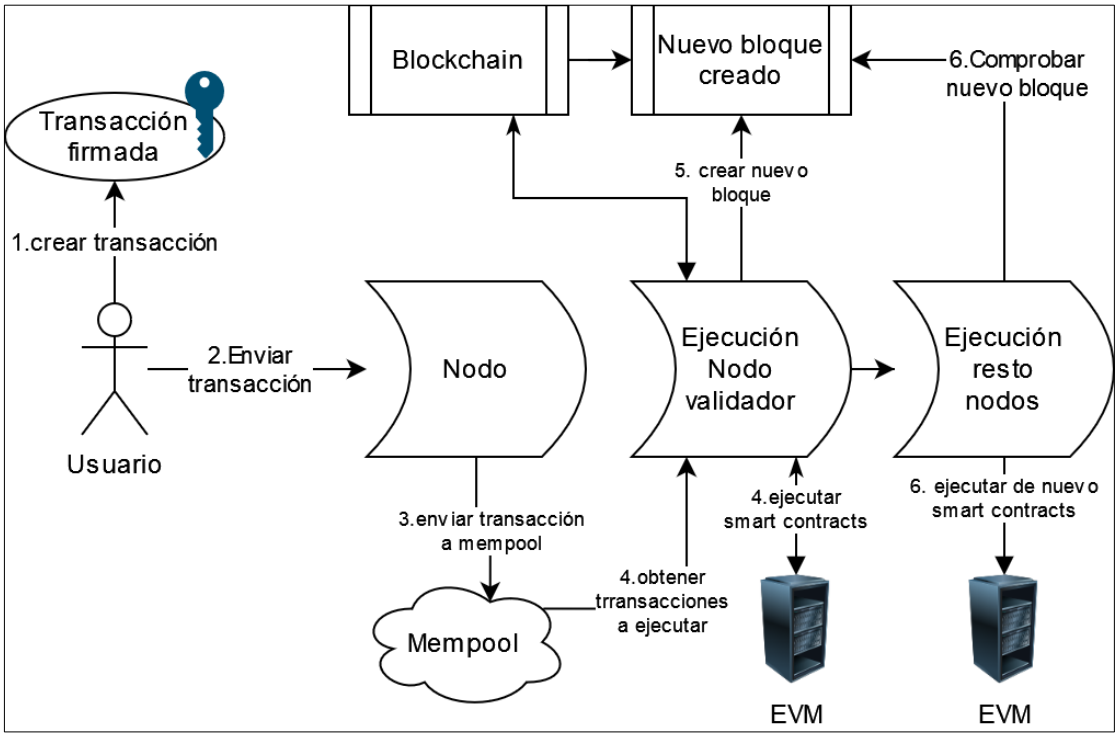

Figura 8: Transacción a smart contract (modifica estado de la Blockchain)

3.6 Ethereum vs otras tecnologías

A pesar de que el presente trabajo se centre en la seguridad de los *smart contracts* en Ethereum, también se comentarán diferencias existentes en alguna de las plataformas más populares.

Ethereum vs Solana

Solana es una *blockchain* más moderna que Ethereum que incorpora nuevas características. Una de ellas es la posibilidad de ejecutar varios *smart contracts* al mismo tiempo de forma paralela gracias a la implementación "Sealevel". Esta tecnología permite ejecutar múltiples contratos al mismo tiempo al agrupar transacciones que no modifiquen el estado de un mismo

contrato. De esta forma se aumenta la eficiencia de la red, sin crear condiciones de carrera.[8]

Contratos en Hyperledger Fabric

Hyperledger Fabric es una *blockchain* con permisos donde cada componente y usuario tiene una identidad, y existen políticas que definen el control de acceso y gobierno.

Como el acceso es privado, el vector de ataque de la red disminuye considerablemente a diferencia de otras redes públicas donde cualquier usuario puede interactuar de forma directa.

La combinación de un vector de ataque menor, junto a las políticas de control de acceso hacen que sea mucho más complejo realizar un ataque con éxito. Además al ser una red privada es mucho más sencillo actualizar el código de los *smart contracts* de encontrarse algún error en el código.

La seguridad sigue siendo un factor importante pero las consecuencias no son tan graves comparado a un *smart contract* en una red pública.

8 https://medium.com/solana-labs/sealevel-parallel-processing-thousands-of-smart-contracts-d814b378192

4. Entorno de trabajo

En este apartado se describirá brevemente el equipo y herramientas utilizadas.

4.1 Equipo

La realización de este trabajo se ha realizado en una computadora portátil. No se necesitan grandes requisitos técnicos para el análisis y desarrollo de *smart contracts*. Un equipo más potente puede ser útil a la hora de ejecutar test u otras herramientas en un menor tiempo, sin embargo la compilación es un proceso rápido al ser los *smart contracts* aplicaciones de poco tamaño y con un lenguaje de bajo nivel.

El portátil utilizado dispone de las siguientes características técnicas de hardware y sistema operativo:

- Procesador Intel I5-6200U (Finales 2015, 2 núcleos, 4 hilos 2,3Ghz)
- 8Gb RAM
- Sistema operativo Debian Sid
- Conexión a internet

4.2 Software

En este trabajo se han utilizado varias herramientas de software necesarias para realizar el desarrollo y pruebas sobre los *smart contracts*. En el apartado 6.2.1 Entorno de pruebas se realiza una breve descripción de algunos de los entornos de desarrollo (*frameworks* en inglés) disponibles.

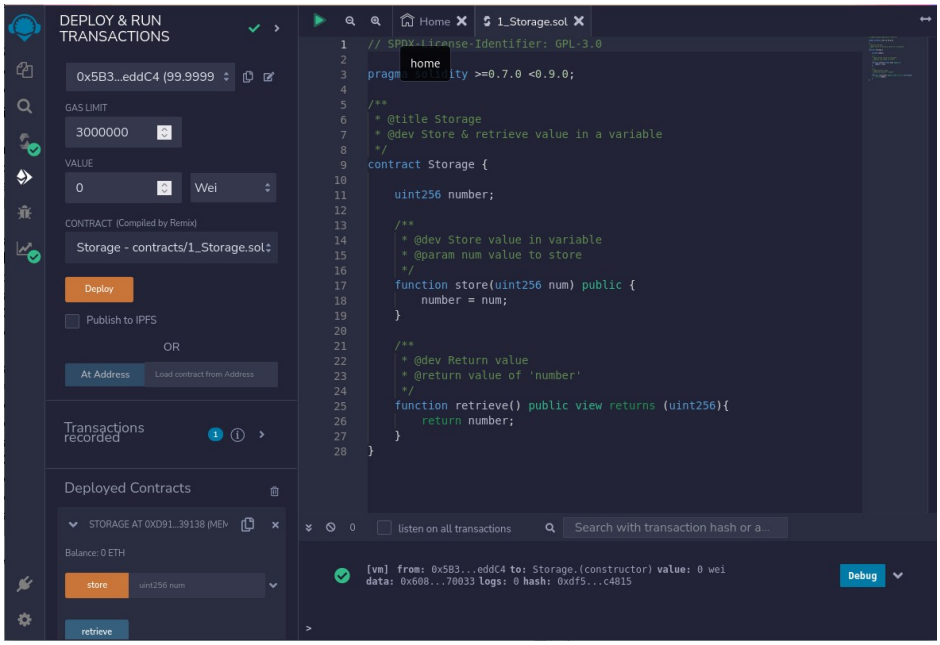

Figura 9: Remix IDE

En concreto se han utilizado dos *frameworks* diferentes según la necesidad.

El primero el IDE Remix[9] (Figura 9), el cual se ejecuta directamente desde el navegador y contiene todo lo necesario en una aplicación web para el desarrollo, compilación y despliegue en una red de pruebas virtual o incluso en las redes oficiales, así como la interación con el *smart contract* desplegado. Esto lo hace perfecto para desarrollar y probar pequeños ejemplos de código de forma sencilla y visual. La mayoría de los ejemplos del capítulo 5 son desarrollados en este *framework*.

Otro *framework* utilizado más completo pero también más complejo es "Brownie"[10]. También proporciona las herramientas de compilación, pruebas y despliegue, pero no incluye un editor ni interfaz gráfica. Su uso se realiza a través de comandos en la consola, o de script programados en Python que se encargan de realizar tareas como pruebas y despliegue de los *smart contracts*. Brownie no proporciona una red de pruebas virtual como Remix, por lo que es necesario utilizar la herramienta "Ganache"[11] si se desean realizar las pruebas sobre los contratos.

Como editor para utilizar junto a Brownie se ha elegido "Visual Studio Code" (Figura 10), por su interfaz limpia y la posibilidad de trabajar con varios lenguajes diferentes en un mismo editor, ya que fue necesario trabajar con los códigos en lenguaje Solidity, Python y JavaScript. Además permite la integración de control de cambios mediante git. Los *smart contracts* del capítulo 6 se han desarrollado en su totalidad en este *framework*.

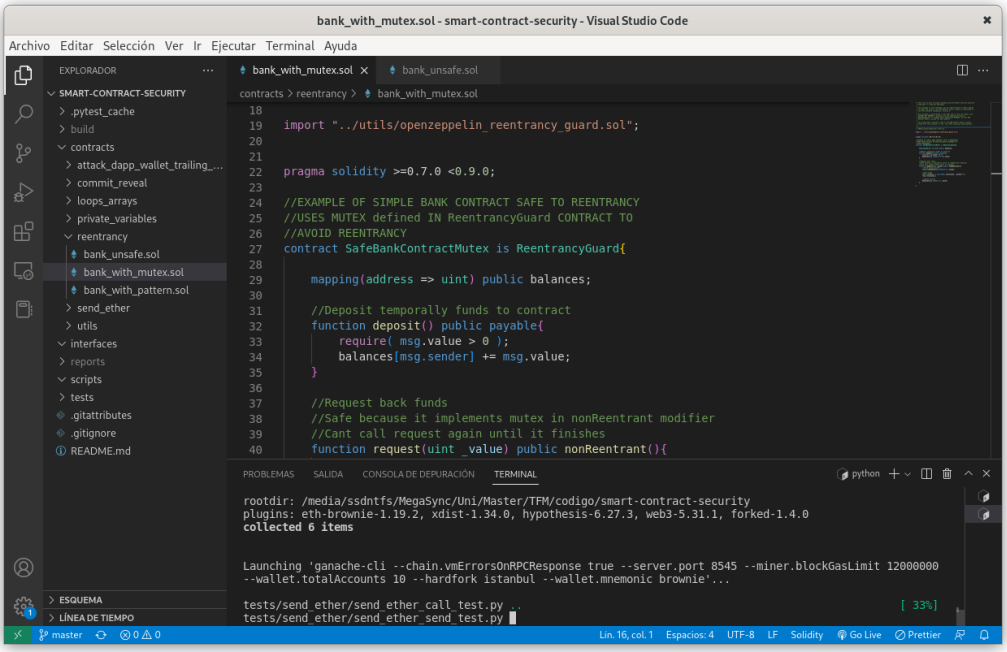

Figura 10: Visual Studio Code + Brownie

9 https://remix.ethereum.org
10 https://eth-brownie.readthedocs.io
11 https://code.visualstudio.com/

4.3 Redes utilizadas

Además de utilizar Remix y Ganache para generar una red virtual y probar los *smart contracts* de forma sencilla, también se han utilizado las redes oficiales de Ethereum. La red principal *(mainnet* en inglés), se ha utilizado únicamente para revisar transacciones ocurridas en el pasado y así entender de mejor forma las vulnerabilidades y ataques ya producidos.

Además se ha empleado la red de pruebas (*testnet* en inglés), en concreto "Sepolia", cuando era necesario entender o replicar alguna de las acciones durante el análisis de los smart contracts. Por ejemplo para comprender mejor el ataque de reentrada, o también el almacenamiento de variables privadas, ambos conceptos analizados en el siguiente capítulo.[12]

Para revisar las transacciones ocurridas tanto en la *testnet* como en la *mainnet* de Ethereum, se ha utilizado el explorador de bloques EtherScan[13]. Permite explorar cualquier bloque, cuenta o transacción realizada en el pasado en cualquiera de las diferentes *Blockchain* públicas de Ethereum.

4.4 Repositorio de código

Los códigos fuente de ejemplos completos se encuentran anexos a este trabajo y también subidos a un respositorio de GitHub (Ver Anexo I: repositorio de código)

12 Cuenta utilizada en las pruebas: 0x61737A6849401f376Ad918ab548d242c30877b2e
13 https://etherscan.io/

5. Vulnerabilidades en los *smart contracts*

En este capítulo se realiza un análisis de las diferentes vulnerabilidades que afectan a los *smart contracts,* en concreto aquellos desarrollados para Ethereum y utilizando el lenguaje de de programación Solidity,. Para ello se han consultado publicaciones[2], así como artículos web reconocidas para elaborar una lista de las más importantes [17]–[19].

Una de las grandes ventajas sobre el funcionamiento de la *Blockchain*, es que es posible acceder a todas las transacciones ocurridas en el pasado, incluidos ataques producidos, facilitando de esta forma su estudio. Como se indica en el capítulo anterior, se ha utilizado el explorador de bloques EtherScan para acceder a las transacciones pasadas.

Además se han recopilado una serie de posibles soluciones para cada caso, aunque en el capítulo 6, se incluyen ejemplos más detallados de algunas de estas soluciones que se aplican en los problemas más comunes que pueden comprometer la seguridad de los *smart contracts*.

Algunas de las vulnerabilidades presentadas son más complejas de cometer gracias a las actualizaciones de seguridad que presentan las herramientas de desarrollo como el compilador que advierten al desarrollador, pero es recomendable conocer las causas para no volver a realizar de nuevo los mismos errores cometidos en las primeras etapas de el desarrollo de *smart contracts*.

5.1 Desbordamiento aritmético (*Underflow/ Overflow*)

5.1.1 La vulnerabilidad

Un desbordamiento aritmético ocurre cuando se realiza una operación aritmética sobre una variable de tamaño fijo, en la que el resultado a almacenar quede fuera del rango del tipo de la variable. En inglés este concepto se denomina *overflow* si el desbordamiento ocurre al superar el rango por arriba, o *underflow* si se supera el rango mínimo.

En el lenguaje Solidity existen varios tipos de enteros, *int* para entero con signo y *uint* para enteros sin signo. Estos dos tipos de variables numéricas se pueden declarar con diferentes tamaños comprendidos entre los 8 bits, hasta los 256 bits, y cualquier tamaño intermedio en incrementos de 8 bits. Un entero con signo de 8 bits se declara como *int8,* mientras que el entero sin signo correspondiente sería *uint8*. Si no se declara un tamaño (*uint* o *int*), por defecto será de 256 bits.

El cálculo del rango para un tamaño de variable es muy sencillo de realizar. La cantidad de números que puede almacenar una variable de *b* bits es 2^b. Por lo tanto en los enteros sin signo el valor mínimo es 0, y el máximo será 2^b-1, ya que un valor lo ocupa el valor "0".

Por otra parte en los enteros la mitad de los valores será para los valores negativos, es decir 2^{b-1}, mientras que los positivos será lo mismo, menos 1 para almacenar el valor "0", es decir 2^{b-1}-1

En la siguiente tabla se hace un pequeño resumen del rango de alguno de los tamaños de enteros más utilizados, pero recordar que existe todo un abanico de rangos posibles en incrementos de 8 bits. (uint8, uint16, uint24, uint32, uint40, …, uint256)

bits	uint min	uint max	int min	int max
8	0	255	-128	127
16	0	65535	-32768	32767
32	0	4294967295	-2147483648	2147483647
64	0	$1{,}84*10^{19}$	$-9{,}22*10^{18}$	$9{,}22*10^{18}$
128	0	$3{,}40*10^{38}$	$-1{,}70*10^{38}$	$1{,}70*10^{38}$
256	0	$1{,}16*10^{77}$	$-5{,}79*10^{76}$	$5{,}79*10^{76}$

Tabla 2: Rango de las variables numéricas según su tamaño en bits

Por ejemplo, una variable del tipo *uint8* permite representar un rango entre 0 y 255 incluidos. Si esta variable tiene valor 250, y le realizamos una suma

con valor 10, el resultado sería 260, superando el rango y produciendo una situación de *overflow*. De forma contraria, si a la misma variable pero con valor 5, le realizamos una resta que supere el rango mínimo, p.ej., 5-10, el resultado sería un número negativo (-5), por lo que causaría una situación de *underflow*.

En versiones anteriores a v.0.8.0 de Solidity, al producirse una situación de underflow o overflow, tan solo se almacenan los bits más significativos, produciendo un efecto de envolvimiento alrededor del máximo (overflow) o del mínimo(underflow). Por lo que en el ejemplo anterior, los resultados obtenidos serían 4, y 251 respectivamente.

Para evitar las consecuencias de un error de programación, en las nuevas versiones el comportamiento por defecto si se detecta un desbordamiento aritmético, es la de realizar un "*revert*" de la transacción, desechando los resultados obtenidos antes de la llamada.

5.1.2 Ocurrencias vulnerabilidad

En abril de 2018, se detectaron dos ataques a varios contratos que implementaban el estándar ERC-20, entre ellos "BeautyChain" y "M2C Mesh Network". Estos contratos contenían una vulnerabilidad de *overflow* que permitió generar una gran cantidad de tokens sin coste para el atacante. [20], [21]

Esta vulnerabilidad se denominó "batchOverflow", por presentarse en la función "batchTransfer()" presentada a continuación:

```
source:
https://etherscan.io/token/0xc5d105e63711398af9bbff092d4b6769c82f793d#code

259  function batchTransfer(address[] _receivers, uint256 _value)
260          public
261          whenNotPaused
262          returns (bool) {
263
264   uint cnt = _receivers.length;
265   uint256 amount = uint256(cnt) * _value;
266   require(cnt > 0 && cnt <= 20);
267   require(_value > 0 && balances[msg.sender] >= amount);
268
269   balances[msg.sender] = balances[msg.sender].sub(amount);
270   for (uint i = 0; i < cnt; i++) {
271          balances[_receivers[i]] = balances[_receivers[i]].add(_value);
272          Transfer(msg.sender, _receivers[i], _value);
273   }
274   return true;
275  }
```

Código 2: Fragmento función vulnerable "BeautyChain"

La función permitía enviar una misma cantidad de tokens a varios destinatarios. En primer lugar se calculaba la cantidad total de tokens a

enviar, que corresponde con el número de destinatarios "_receivers.length", por la cantidad a enviar a cada uno de ellos:

```
265   uint256 amount = uint256(cnt) * _value;
```

Si se envía un valor lo suficientemente alto, se podria producir un overflow en la multiplicación, de forma que amount pase a tener un valor muy bajo, pasando así la comprobación de la línea 265.

De esta forma el atacante pudo enviar $5{,}79*10^{76}$ tokens a dos cuentas, cuando la cantidad de tokens máxima era de solo 7,000,000,000 y sin necesidad de poseer tokens en la cuenta de origen.[14]

La clave es que el número de tokens $5{,}79*10^{76}$ de la variable *_value* multiplicado por 2 tiene como resultado 0 si se produce un overflow en el tipo uint256. De esta forma la variable *amount* pasará a valer 0, permitiendo enviar esa cantidad sin que el emisor disponga de ningún token.

Otra vulnerabilidad detectada dos dias después fue proxyOverflow, muy similar a la anterior.

14 https://etherscan.io/tx/0xad89ff16fd1ebe3a0a7cf4ed282302c06626c1af33221ebe0d3a470aba4a660f

```
Source:
https://etherscan.io/token/0x3ac6cb00f5a44712022a51fbace4c7497f56ee31#code

205  function transferProxy(
206   address _from,
207   address _to,
208   uint256 _value,
209   uint256 _fee,
210   uint8 _v,
211   bytes32 _r,
212   bytes32 _s)
213   public transferAllowed(_from) returns (bool){
214
215   if(balances[_from] < _fee + _value) revert();
216
217   uint256 nonce = nonces[_from];
218   bytes32 h = keccak256(_from,_to,_value,_fee,nonce);
219   if(_from != ecrecover(h,_v,_r,_s)) revert();
220
221   if(balances[_to] + _value < balances[_to]
222         || balances[msg.sender] + _fee < balances[msg.sender]){
223         revert();
224   }
225   balances[_to] += _value;
226   Transfer(_from, _to, _value);
227
228   balances[msg.sender] += _fee;
229   Transfer(_from, msg.sender, _fee);
230
231   balances[_from] -= _value + _fee;
232   nonces[_from] = nonce + 1;
233   return true;
234  }
```

Código 3: Fragmento código vulnerable "M2C"

En este caso en la línea 215 se comprueba que el emisor posea más de la cantidad total que se quiere enviar. El problema es que la suma de _fee y de _value puede desbordar la variable uint256 si ambos valores son los suficientemente grandes.

Un atacante aprovechó este fallo para que la suma produjese como resultado 0 pasando de esta forma la comprobación de la línea 225, pero enviando una cantidad muy grande en las líneas 225, y 228.[15]

5.1.3 Solución

A partir de la versión v.0.8.0 de Solidity se introdujo de forma predeterminada, la comprobación de operaciones aritméticas que produzcan *overflow* o *underflow*, de forma que la transacción revierta de forma automática si se da una de las dos situaciones.[16] Esta comprobación añade un pequeño coste de gas y es posible desactivar este

15 https://etherscan.io/tx/0xa850ce29dac8cefd0cf3250486a8080ca32b0a94308b9ac3ac6639140e601344
16 https://docs.soliditylang.org/en/develop/types.html#integers

comportamiento mediante el identificador "*unchecked{ }*". Por lo tanto es muy importante revisar los cambios y recomendaciones introducidos en el compilador (Ver 6.1.1Versión actualizada del compilador).

A pesar de esta protección, el programador sigue siendo responsable de manejar estas situaciones de *underflow/overflow* de forma correcta. En ocasiones la acción por defecto de revertir la operación puede no ser la deseada y provocar que el contrato quede en una situación de bloqueo. (Ver 6.1.6Manejo de errores).

5.2 Reentrada

5.2.1 La vulnerabilidad

El descubrimiento de las vulnerabilidades de reentrada dieron lugar a uno de los ataques más trascendentes y polémicos en la historia de Ethereum: El DAO

Una reentrada ocurre cuando se origina una llamada desde un contrato externo a otro contrato, y se permita que el contrato externo realice nuevas llamadas al contrato antes de que la primera ejecución finalice.

Para que esto sea posible, el contrato vulnerable ha de realizar un envío de ether a otra cuenta mediante la función "call.value". A diferencia de utilizar las demás funciones disponibles como "transfer" o "send" que limitan el gas del envío a 21000, el uso de "call" no restringe la cantidad de gas a utilizar. El gas disponible será el restante de la transacción original.

Cuando un envío mediante "call" es dirigido a una cuenta Ethereum corriente, simplemente se transfiere la cantidad indicada sin mayor problema. Sin embargo, cuando este envío es dirigido a un smart contract, se invocará a la función *fallback* o *receive* del mismo. Esta función *fallback* puede contener código para realizar otra nueva llamada al contrato vulnerable, que se ejecutaría antes de que finalice la primera llamada. Si el uso de gas estuviese restringido a 21000, no sería suficiente para generar una nueva llamada.

Ejemplo de reentrada

A continuación se muestra de forma más claro este concepto con un ejemplo.

file: contracts/attack_reentrancy_example/shared_wallet_vulnerable.sol

```
1  contract SharedWallet{
2
3  mapping (address => uint) balances; //balance de cada usuario
4
5  (...)
6  function withdrawAll() public payable{
7      require(balances[msg.sender] > 0);
8
9      //Sent ether back to user
10     (bool status, ) = msg.sender.call{value: balances[msg.sender]}("");
11     require(status);
12
13     //Update user balance
14     balances[msg.sender] = 0;
15     }
16 }
17 (...)
```

Código 4: Fragmento contrato vulnerable

El contrato "*SharedWallet*", permite realizar depósitos y retiradas desde varias cuentas mientras se mantiene un registro de los balances de cada usuario. La función *withdrawAll()*, envía todos los fondos de la cuenta mediante la llamada "msg.sender.call", y después actualiza la variable que registra el balance. Como se realiza primero el envío mediante "call", y después se actualiza la variable con el balance, esta función es susceptible a un ataque de reentrada.

file: contracts/attack_reentrancy_example/shared_wallet_exploit.sol

```
1  contract Ataque{
2
3  function exploitDeposit() onlyowner public{
4      vulnContract.deposit{value:1000}();
5  }
6
7  function exploitWithdraw() onlyowner public{
8          vulnContract.withdrawAll();
9  }
10
11 fallback() external payable{
12     if (address(vulnContract).balance >= 1000){
13         vulnContract.withdrawAll();
14     }
15 }
```

Código 5: Fragmento contrato ataque

La llamada "msg.sender.call" si es realizada a un contrato inteligente, permitiría que esta en su función *fallback* realizase una llamada a *withdrawAll* de nuevo. Como el saldo restante de la cuenta no ha sido actualizado todavía, se realizaría un segundo envío de Ether. Este segundo envío también ejecutaría la función *fallback* de nuevo, lo que permite entrar en un bucle hasta vaciar los fondos del contrato "*SharedWallet*".

Estas acciones son las que se realizan en el contrato "A*taque*". En primer lugar mediante la función *exploitDeposit*, se realizaría un depósito, en este caso de valor 1000 wei, en nombre del contrato para disponer de saldo en el contrato "*SharedWallet*".

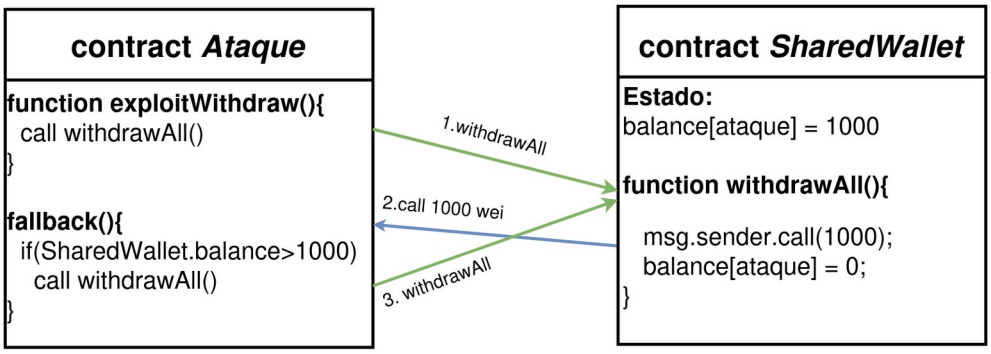

Figura 11: Ataque de reentrada

A continuación se iniciaría el ataque de reentrada de forma recursiva, mediante la función *exploitWithdraw,* que llama a SharedWallet.withdrawAll (1). Desde withdrawAll se realiza el envío de 1000wei hacia Ataque mediante "call"(2), que son recibidos en la función *fallback* del mismo.

Desde la función *fallback* es donde se controla el ataque. Antes de iniciar una nueva llamada a *withdrawAll*, se comprueba la cantidad de dinero disponible en el contrato *SharedWallet*, para evitar retirar más cantidad de la que dispone ya que se revertiría todo el ataque. Si dispone de fondos se realiza una nueva llamada a *withdrawAll* (3).

Los pasos (2) y (3) se repetirían en bucle hasta llegar al máximo de gas disponible, o hasta vaciar los fondos del contrato *SharedWallet*, vaciando el contrato *SharedWallet* con una cantidad de 1000wei (en este caso) por cada recursión realizada. Las cantidades pueden ser mayores, si se depositasen 1000 Ether se podrían obtener 1000 Ether por cada recursión.

5.2.2 Vulnerabilidades producidas

"El DAO" fue una asociación autónoma descentralizada creada por la empresa slock.it, que tenía como objetivo la financiación monetaria de empresas comerciales y sin ánimo de lucro. Esta funcionalidad se implementó en un contrato inteligente en Abril de 2016 tras uno de los mayores micromecenazgos o *crowdfoundings* hasta entonces.

Los ataques de reentrada eran desconocidos en Ethereum hasta el descubrimiento de un antipatrón de diseño en Junio de 2016 presente entre otros en el contrato DAO. Una vulnerabilidad fue encontrada y subsanada a tiempo en dicho contrato. [22]

Sin embargo una de las funciones también contenía una vulnerabilidad similar, *splitDAO,* que pasó inadvertida. Menos de una semana después de la publicación de slock.it, comenzó el ataque por el que se substrajeron 3.5 Millones de ETH, (50 Millones de US$ en 2016), que fueron transferidos a un DAO hijo, que solo controlaba el atacante.

El atacante no podría acceder a los fondos hasta pasado un periodo de tiempo marcado por el contrato DAO. Hasta entonces hubo gran debate hasta la realización de un *Hard-fork* de la red para restaurar los fondos. Muchos nodos no lo aplicaron y se creó la bifurcación de Ethereum (ETH), y Ethereum Classic (ETC) [23]

El fundamento de la vulnerabilidad es similar al explicado en el contrato de ejemplo, aunque en este caso es código es más complejo.

Por una parte la funcionalidad *splitDAO*, permitía dividir una parte del DAO principal a una DAO secundaria.[24] Para ello era necesario realizar una propuesta y esperar 7 días. Tras ese tiempo era posible llamar a la función *splitDAO,* que transfiere los tokens desde el DAO principal hasta el nuevo DAO creado.

Función splitDAO

```
 1  function splitDAO( uint _proposalID, address _newCurator
 2      ) noEther onlyTokenholders returns (bool _success) {
 3
 4    (...)
 5
 6    // Move ether and assign new Tokens
 7    uint fundsToBeMoved = (balances[msg.sender] *
                  p.splitData[0].splitBalance) /
p.splitData[0].totalSupply;
 8
 9    if (p.splitData[0].newDAO.createTokenProxy.value(fundsToBeMoved)
(msg.sender) == false)
10      throw;
11
12    (...)
13
14    // Burn DAO Tokens
15    Transfer(msg.sender, 0, balances[msg.sender]);
16    withdrawRewardFor(msg.sender); // be nice, and get his rewards
17    totalSupply -= balances[msg.sender];
18    balances[msg.sender] = 0;
19    paidOut[msg.sender] = 0;
20    return true;
21  }
```

Código 6: fragmento contrato DAO (función splitDAO)

En primer lugar se calculan en la variable *fundsToBeMoved* la cantidad de tokens que corresponden enviar para este usuario al nuevo DAO hijo.

Más adelante estos tokens son enviados mediante la función *withdrawRewardFor()*. Notar que son enviados antes de actualizar el saldo de balances[msg.sender] = 0.

```
 1  function withdrawRewardFor(address _account)
 2    noEther internal returns (bool _success) {
 3    if ((balanceOf(_account) * rewardAccount.accumulatedInput()) /
totalSupply < paidOut[_account])
 4      throw;
 5
 6    uint reward =
 7      (balanceOf(_account) * rewardAccount.accumulatedInput()) /
totalSupply - paidOut[_account];
 8    if (!rewardAccount.payOut(_account, reward)) // vulnerable
 9      throw;
10    paidOut[_account] += reward;
11    return true;
12  }
```

Código 7: fragmento DAO (función withdrawRewardFor)

En *withdrawRewardFor()* se calcula la cantidad a enviar en la variable *reward*. Esta cantidad siempre es la misma mientras no se actualice la variable *paidOut[account],* que no es actualizada hasta después de ser enviado los tokens mediante *payOut()*. De nuevo el problema es que en payOut se realiza el envío de dinero mediante *call,* lo que permite un ataque de reentrada antes de que se actualice la variable mencionada.

```
1  function payOut(address _recipient, uint _amount) returns (bool) {
2    if (msg.sender != owner || msg.value > 0 || (payOwnerOnly &&
_recipient != owner))
3        throw;
4    if (_recipient.call.value(_amount)()) { // vulnerable
5        PayOut(_recipient, _amount);
6        return true;
7    } else {
8        return false;
9  }
```

Código 8: fragmento DAO (función payOut)

Por último en la función payOut donde se realiza la llamada a *_recipient.call*, donde _recipient es la dirección de la cartera que contendrá el ataque, y que permite realizar una llamada recursiva a splitDAO de nuevo.

El stack de llamadas en el ataque de reentrada sería el siguiente:

- splitDAO()
  - withdrawRewardFor()
    - payOut()
      - _recipient.call() → fallback function del contrato de ataque
        - splitDAO()

- (…)

El atacante fue capaz de generar 30 recursiones en cada transacción realizada. El balance de la cuenta maliciosa en el contrato DAO original era de 258ETH, que multiplicados por 30 generan un total de 7780ETH obtenidos por transacción.

Al final de la transacción se ejecutaría la línea de splitDAO *balances[msg.sender] = 0,* por lo que el atacante no podría realizar un robo mayor a estos 7780ETH salvo repitiendo el ataque desde 0 y generando muchas propuestas de división del DAO con el tiempo de espera que conllevan.

Sin embargo el atacante fue capaz de elevar el x30 hasta un ataque prácticamente infinito, simplemente transfiriendo los fondos de su cuenta a otra cuenta utilizando la función *transfer* de DAO al comienzo del stack. De esta forma al finalizar el ataque con devolver los tokens de nuevo a la cuenta ya podría reiniciar el ataque.[25], [26]

5.2.3 Soluciones

Las vulnerabilidades de reentrada tan solo pueden ocurrir cuando se realiza una llamada externa, por lo que hay que tener precaución en su uso. Existen varias soluciones para evitar la vulnerabilidad:

- Utilizar patrón de diseño “Comprobar, Modificar, Interactuar”

  Se basa en que los cambios de estado del contrato se realicen antes de la llamada externa, y que después de esta no se puedan realizar más cambios.

  De esta forma si se realiza una reentrada, el estado del contrato ya habrá sido actualizado.

- Utilizar “mutex” que prohíban la reentrada hasta finalizar la función.

  Utilizan una variable para controlar que hasta que no finalice la ejecución entera de la función comprometida, esta no puede ser invocada de nuevo.

Estas dos soluciones se exploran en detalle en el apartado “6.1.2Llamadas externas – evitar ataques reentrada”. También es recomendable la lectura del apartado “6.1.2Llamadas externas – evitar ataques reentrada”, donde se dan recomendaciones adicionales a la hora de realizar envíos de Ethereum.

5.3 Control de Acceso I: modificadores visibilidad

5.3.1 La vulnerabilidad

Las vulnerabilidades de control de Acceso se encuentran en el primer lugar de la lista OWASP Top 10 2021[17]. Estas vulnerabilidades ocurren cuando no se utilizan correctamente las políticas de control de acceso disponibles para evitar que un usuario no puedan actuar fuera de los permisos intencionados.

En los Smart Contracts una vulnerabilidad de Control de Acceso se puede producir en muchos contextos, por ejemplo, permitir ejecutar varias veces funciones de un solo uso pensadas como inicializador del contrato o permitir ejecutar de forma pública funciones de forma de uso interno o sólo por el creador del contrato.

5.3.2 Ocurrencias vulnerabilidad

Un contrato con una vulnerabilidad de control de acceso fue Rubixi[27], un juego de esquema Ponzi donde los jugadores envían dinero con la expectativa de recibir más cantidad de Ether.

El creador del contrato tenia acceso privilegiado para modificar parámetros de ejecución, así como para canjear las comisiones producidas por el mismo.

Rubixi fue diseñado utilizando la versión 0.2.1 de Solidity, que tiene varias diferencias que disminuyen la seguridad respecto a versiones más actuales (versión actual de Solidity a 11/2022 v.0.8.7):

- La primera es que todas las funciones por defecto son públicas si no se indica un modificador de visibilidad. En la actualidad es obligatorio asignar el modificador de visibilidad, lo que disminuye la probabilidad de configurar erróneamente una función.
- La segunda es la definición del constructor del contrato. El constructor es una función opcional, sólo llamada de forma automática durante el despliegue del contrato, no pudiendo ser llamada en ningún otro momento, y en ella se define la lógica para configurar el estado inicial del mismo.

  En la actualidad la función constructor del contrato se define de la siguiente forma:

```
constructor(/*argumentos*/){…}
```

  A diferencia de la versión v0.2.1, donde se utilizaba el nombre del contrato para definir la función constructor. En este caso al ser Rubixi, debería haber sido:

17 https://owasp.org/Top10/A01_2021-Broken_Access_Control/

```
function Rubixi(/*argumentos*/){…}
```

Sin embargo, en el contrato en la red, la función pensada para ser constructor del mismo tiene la siguiente forma:

```
1  contract Rubixi{
2      address private creator;
3
4      (...)
5
6      function DynamicPyramid(){
7          creator = msg.sender;
8      }
9
10     modifier onlyowner {
11         if (msg.sender == creator) _
12     }
13
14     (...)
15 }
```

Código 9: fragmento contrato Rubixi

El desarrollador probablemente había llamado en un principio al contrato por el nombre DynamicPyramid, por lo que la función *DynamicPyramid()*, sería su constructor. Al cambiar el nombre del contrato y no la del constructor, lo que provocó es que esta función pasase a ser una función normal y corriente, y lo que es peor, de acceso público.

Por lo tanto, cualquier usuario podría llamar a la función *DynamicPyramid*() y convertirse en el creador del contrato. Gracias a esto, ese mismo usuario podría invocar a las funciones designadas por el modificador *onlyowner,* como por ejemplo, la función *collectAllFees()*, encargada de enviar al creador las "fees" generadas en el contrato.

```
1  //Fee functions for creator
2  function collectAllFees() onlyowner {
3      if (collectedFees == 0) throw;
4          creator.send(collectedFees);
5          collectedFees = 0;
6  }
```

Código 10: fragmento contrato Rubixi

Si se analizan las transacciones realizadas al contrato[18], se puede observar que el propio creador del contrato no se da cuenta en un principio del error. Intenta llamar a la función "collectAllFees()", pero sin embargo no recibe nada. Esto es porque la función pensada como constructor, no había sido llamada en ningún momento, y por tanto la variable *creator* no estaba inicializada.

18 https://etherscan.io/txs?a=0xe82719202e5965cf5d9b6673b7503a3b92de20be&p=10

Más tarde se da cuenta de su error, y llama a la función *DynamicPyramid* para convertirse en creador y obtener la recompensa. Sin embargo no es el único que puede llamar a esta función, cualquier usuario puede hacerlo y reclamar la recompensa en cualquier momento. Dada la inmutabilidad de los contratos de Ethereum salvo que se usen mecanismos para actualizar su funcionalidad, que no es el caso, múltiples usuarios intentaron llevarse las comisiones generadas por el contrato hasta que quedó en desuso por la comunidad.

5.3.3 Soluciones

Existen varias soluciones a la vulnerabilidad, algunas generales y otras específicas al ejemplo analizado:

Modificadores de funciones:

En las versiones actuales de Solidity es más difícil que ocurra una vulnerabilidad de control de acceso por despiste del programador. Desde la v.0.5.0, es obligatorio indicar modificador de visibilidad para todas las funciones del contrato.

Esto sin embargo no evita que el programador defina como como pública una función que no debería serlo, por lo que se ha de prestar especial atención a utilizar el modificador de visibilidad correcto.

Ver 5.1.1 Versión actualizada del compilador

Test de código:

La vulnerabilidad ocurrida en Rubixi sería fácilmente detectable con un test exhaustivo del contrato, ya que en su estado inicial la variable creador no estaría inicializada ya que no se invocó a la función pensada como constructor en ningún momento. En la llamada a la función de cobrar las comisiones por lo tanto no se generaría una transacción de su cobro y sería detectado el error.

Ver 5.2 Pruebas de smart contracts

Manejo de errores:

A mayores de los test también es recomendable manejar correctamente los errores en las llamadas al contrato. En la versión de Solidity utilizada por Rubixi todavía no estaba extendido el uso de *revert()* (en aquel entonces *throw()*) para causar una excepción en vez de devolver un resultado negativo o nulo. El uso de excepciones junto a los test advertirían rápidamente al programador de que había un error en la lógica del código.

Ver 5.1.6 Manejo de errores

Versión del compilador:

Por último es recomendable utilizar una versión actualizada del compilador. Como se ha indicado, con el paso del tiempo se han añadido diversas

características en Solidity que tienen como fin mejorar la seguridad del código y evitar errores de programación. Es igual de importante leer la documentación donde se indican apuntes y recomendaciones para programar de forma correcta y sin errores.

Ver 5.1.1 Versión actualizada del compilador

5.4 Control de Acceso II: librerías y delegatecall

5.4.1 La vulnerabilidad

Continuando con el apartado anterior, detallaremos otro tipo de vulnerabilidad de control de acceso. El fundamento es similar, que un usuario no privilegiado pueda ejecutar código que debería estar fuera de su alcance.

En este caso involucra un mal uso de librerías externas utilizadas de forma dinámica mediante *delegatecall*. *(Ver Librerías externas y delegate call)*

5.4.2 Ocurrencias vulnerabilidad

El contrato "Parity Multi-sig" permitía crear una cartera controlada por varios usuarios, pudiendo ser configurada para que cualquiera de los usuarios de la misma puedan enviar transacciones, o requerir un número mínimo de firmas para ello.

El contrato estaba dividido en dos partes con el fin de ahorrar espacio en el despliegue del mismo. El código común a todas ellas estaba en un contrato librería *"WalletLibrary"*, mientras que el contrato *"Wallet"*, es el que desplegaban los usuarios que querían crear su propia cartera Multi-sig.

Este contrato sufrió dos ataques por diferentes vulnerabilidades. La primera vulnerabilidad fue detectada en un ataque ocurrido el 19 de Julio de 2017 y fue posible por dos fallos de seguridad presentes uno en la librería "*WalletLibrary*" y otro en el contrato *"Wallet"*.

Parity Multi-sig bug 1:

```
1  // constructor - just pass on the owner array to the multiowned and
2  // the limit to daylimit
3  function initWallet(address[] _owners, uint _required, uint _daylimit) {
4      initDaylimit(_daylimit);
5      initMultiowned(_owners, _required);
6  }
```

Código 11: fragmento librería WalletLibrary

En la creación del contrato Wallet, se invoca a la función "*initWallet*" de la librería "WalletLibrary" para establecer los usuarios de la cartera, junto a otros parámetros como el número de firmas necesarias para enviar una transacción, y un límite diario de gasto.

Es necesario recordar que mediante el uso de "*delegatecall*", las variables modificadas en las funciones de la librería son las del propio contrato "Wallet". *(Ver Librerías externas y delegate call)*

La función *initWallet* estaba pensada para ser llamada una sola vez por el contrato que la utiliza, y sin embargo no se hace ninguna comprobación

para evitar que esto ocurra. Esto por si solo no permite ningún ataque si en el código del contrato que utilice la librería se realiza la llamada a esta función de forma controlada.

Sin embargo en el contrato "Wallet" esto no fue así. En la función *fallback* de este contrato se realizaban dos tareas:

```
1  function() payable {
2      // just being sent some cash?
3      if (msg.value > 0)
4          Deposit(msg.sender, msg.value);
5      else if (msg.data.length > 0)
6          _walletLibrary.delegatecall(msg.data);
7  }
```

Código 12: función payable del contrato Wallet

En la primera si el contrato recibía una cantidad de criptomonedas simplemente las almacenaba. Sin embargo si se invocaba a la función *fallback* sin ningún valor, se llamaba directamente a la librería, con los datos que haya indicado el usuario.

Esto presenta varios problemas:

La función fallback es pública, de forma que cualquier usuario y no solo los administradores del contrato pueden invocarla.

Al enviar directamente la entrada del usuario a la librería conlleva que el usuario pueda invocar a cualquier función de la librería a su antojo, incluyendo a la función *initWallet,* lo que sobrescribiría la lista de administradores de la cartera.

Esto fue justo lo que ocurrió, tres carteras multi-sig fueron atacadas substrayendo una cantidad mayor a 150'000ETH, el equivalente a (~30M USD) en aquel momento.

Parity Multi-sig bug 2:

A pesar que la vulnerabilidad fue subsanada la misma semana de ocurrir el ataque, en noviembre de ese mismo año, otra vulnerabilidad de control de acceso, esta vez presente tan solo en la librería, permitió que un usuario malintencionado destruyese la misma, dejando sin funcionalidad a los contratos que dependían de ella.

La el contrato librería "WalletLibrary"[28] estaba pensado para ser utilizado únicamente como una librería y no estaba contemplado que se le realizasen llamadas directamente. Las funciones de este contrato tan solo deberían actuar sobre las variables de otro contrato a través del uso de *delegateCall*.

Si alguien llama directamente a las funciones de la librería, las variables modificadas serán las de la librería, que fue lo que ocurrió en este caso. La librería contenía dos funciones que fueron las que desencadenaron su destrucción.

Un usuario realizó dos llamadas al contrato, la primera[19] a la función "initWallet", ya mostrada en la vulnerabilidad anterior. La librería no estaba inicializada, por lo que si un usuario llamaba a esa función, se convertiría en el dueño de la librería.

Esto posibilitó que ese mismo usuario, ahora dueño de la librería, pudiese llamar[20] a la función "kill", pensada para destruir una cartera multi-sig, pero en este caso actuaría sobre el contrato librería, destruyéndolo. Tras este hecho, todos los contratos Wallet Multisig que dependían de la librería quedaron inutilizados.

```
1  // kills the contract sending everything to `_to`.
2  function kill(address _to) onlymanyowners(sha3(msg.data)) external {
3     suicide(_to);
4  }
```

Código 13: función kill() WalletLibrary

La función "suicide", ahora renombrada como "*selfdestruct*", provoca la eliminación del *bytecode* del contrato y la retirada de fondos a una dirección indicada. [21]

5.4.3 Soluciones

En las dos vulnerabilidades hay varias acciones a llevar a cabo que pudieron evitarlas:

Acceso no controlado a "delegateCall"

Este fue el error más grave cometido por los desarrolladores. Cualquier persona podría llamar a esta función a través de la función *fallback* y lo que es aún peor, sin restricción de la función a llamar y sus argumentos.

La solución es sencilla, no incluir esta función en una función *fallback*, y de ser necesario, controlar que tan solo usuarios autorizados pueden realizar esta llamada.

Además es recomendable separar la lógica de recibir Ether, y de recibir una llamada que no se corresponda con ninguna función del contrato. A partir de Solidity v.0.6.0, se incluye la función "*receive*" y la función "*fallback*", cada una pensada para realizar estas dos tareas de forma exclusiva.

Inicialización múltiple

Otro error menos grave pero que ayudó a la existencia de la primera vulnerabilidad. En este caso se permitía inicializar la librería más de una vez,

19 https://etherscan.io/tx/0x05f71e1b2cb4f03e547739db15d080fd30c989eda04d37ce6264c5686e0722c9
20 https://etherscan.io/tx/0x47f7cff7a5e671884629c93b368cb18f58a993f4b19c2a53a8662e3f1482f690
21 https://docs.soliditylang.org/en/v0.8.17/introduction-to-smart-contracts.html?highlight=selfdestruct#deactivate-and-self-destruct

sin comprobar requisitos tan importantes como la de tener permisos de todos los usuarios de la cartera.

La solución más sencilla es permitir inicializar tan solo una vez mediante una comprobación al principio de la función, fácilmente implementable mediante un modificador.

Contratos no inicializados

De forma general es recomendable inicializar un contrato en su despliegue para evitar comportamientos no deseados. Si el contrato librería hubiera sido inicializado en su despliegue, no se podrían haber realizado las llamadas que provocaron su destrucción posterior.

Sin embargo un contrato librería se desarrolla de forma diferente en la actualidad, tal y como se indica en el siguiente punto.

Desarrollo de librerías y versión del compilador

En versiones anteriores a v.0.4.20 de Solidity, era posible realizar llamadas a una librería de forma directa como un contrato más. Esto provoco que fuese posible llamar a la función "*initWallet*" y a la función "kill". A partir de esa versión, se realiza una comprobación de la dirección del contrato sobre el que se está actuando, eliminando efectivamente las llamadas que modifiquen el estado de un contrato librería.

Una solución en aquel entonces hubiera sido inicializar la librería en su despliegue para que nadie pudiese realizar llamadas a las funciones de la misma.

En la actualidad, es importante utilizar una versión actualizada de Solidity, y tener en cuenta que un contrato librería se declara con la palabra clave "*library*" para diferenciarlo de un contrato común. Esto evita que un contrato pueda ser llamado directamente y consecuente eliminado mediante una llamada a "*selfdestruct*".

(Ver 5.1.1 Versión actualizada del compilador)

5.5 Uso incorrecto llamadas bajo nivel

5.5.1 La vulnerabilidad

Cuando se realizan llamadas externas en Solidity, ya sea para enviar Ether o para interactuar con una función de otro contrato, es importante interpretar correctamente la información recibida tras la llamada.

Para enviar Ether a otra cuenta se pueden emplear varias funciones. Entre ellas *transfer()*, *send() y call()*. Una de las diferencias más importantes es que *transfer()* revierte la transacción automáticamente si hay un problema durante el envío, mientras que *send()* y *call()* tan solo devuelven un valor booleano False.

Un error común, sobretodo en las primeras etapas de Solidity, era no comprobar este valor booleano, ya que el desarrollador imaginaba que la operación revertería automáticamente de haber un error.

Esto es especialmente importante al realizar envíos de Ether a contratos inteligentes, los cuales pueden utilizar más de 2300 de gas para recibir, y por lo tanto la función *send()* fallaría devolviendo *False*.

5.5.2 Ocurrencias vulnerabilidad

"King of the Ether" fue un simple juego de tipo Ponzi como muchos otros en los inicios de Ethereum. El funcionamiento era simple, existía puesto de trono en el contrato, que pertenecía a última persona que pagase la oferta que existía por el mismo. El precio a pagar para acceder al trono aumentaba por un porcentaje (33%), cada vez que había un nuevo postor.

Esta cantidad se pagaba al antiguo Monarca, por lo que este siempre recuperaba su desembolso inicial más un porcentaje, salvo en el caso que ningún sucesor pagase el precio en 14 días, periodo tras el cual el contrato se reseteaba a un precio de 0.5ETH.

El contrato en su versión 0.4 contenía una vulnerabilidad al enviar la compensación al último monarca mediante la función *send()* sin comprobar el valor de retorno.

```
// Claim the throne for the given name by paying the currentClaimFee.
function claimThrone(string name) {

  uint valuePaid = msg.value;

  // If they paid too little, reject claim and refund their money.
  if (valuePaid < currentClaimPrice) {
      msg.sender.send(valuePaid);
      return;
  }

   // If they paid too much, continue with claim but refund the excess.
   if (valuePaid > currentClaimPrice) {
      uint excessPaid = valuePaid - currentClaimPrice;
      msg.sender.send(excessPaid);
      valuePaid = valuePaid - excessPaid;
   }
```

Código 14: fragmento función vulnerable en "King of Ether"

En tres instancias del contrato existía este error. Las dos primeras si el nuevo solicitante pagaba demasiado poco dinero devolviéndoselo, o si pagaba demasiado, por lo que se le devolvía el valor en exceso.

Y también al pagar la compensación al antiguo monarca:

```
  if (currentMonarch.etherAddress != wizardAddress) {
      currentMonarch.etherAddress.send(compensation);
  }
```

Código 15: fragmento función vulnerable en "King of Ether"

Lo que ocurrió es que un monarca utilizaba como cuenta un contrato inteligente, que a la hora de recibir dinero utilizaba más de 2300 de valor de gas, por lo que la función send() no se completaba y devolvía false.

Sin embargo esto no se comprobaba y por lo tanto el contrato quedó en un estado inesperado, al tener Ether en exceso y donde el último monarca no fue compensado correctamente.

Finalmente el dinero fue reembolsado de forma manual por el creador.

5.5.3 Soluciones

Existen varias soluciones y consejos para evitar este error:

Comprobar return value de operaciones *send()* y *call()*

La solución más directa es comprobar el valor de retorno al utilizar estas dos funciones cuando se realiza un envío de Ether ya que al dar error devuelven False en vez de revertir la operación.

En la actualidad, Solidity advierte cuando no se utiliza la variable de retorno procedente de una llamada externa.

Realizar un revert en vez de devolver dinero

En la primera ocurrencia en el código de "*King of the Ether*" de send, se utiliza para devolver la cantidad ingresada si es menor al valor necesario para completar la operación.

Una mejor práctica es realizar un *revert* si se detecta que los parámetros de entrada no son los correctos o esperados, evitando así introducir vulnerabilidades al no producirse ningún cambio en el estado del contrato.

Envío seguro de Ether

Aunque se utilice la función *transfer()* que revierte la transacción en caso de error*, o send()* comprobando los valores de retorno, si la cuenta de destino es un contrato inteligente que requiera más de 2300 de gas para recibir dinero en su función *receive() o fallback()*, se produciría un bloqueo en el estado del contrato tal y como está programado.

En el ejemplo, el nuevo monarca es el que inicia la transacción para enviar los fondos al antiguo monarca. Por lo tanto si la transacción para enviar los fondos provoca un *revert*, el nuevo monarca no podrá atribuirse el trono de ninguna manera.

Una mejor solución es un patrón de retirada, donde la proclamación del nuevo monarca provoque que se añada al estado del contrato que el antiguo monarca pueda retirar cierta cantidad. De esta forma, el antiguo monarca será el que bloquee su retirada y no el contrato entero si utiliza un contrato que requiera un costo de gas demasiado alto.[22]

22 https://docs.soliditylang.org/en/v0.8.15/common-patterns.html#withdrawal-from-contracts

```solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract WithdrawalContract {
    address public richest;
    uint public mostSent;

    //ARRAY CON LAS RETIRADAS PENDIENTES
    mapping (address => uint) pendingWithdrawals;

    /// The amount of Ether sent was not higher than
    /// the currently highest amount.
    error NotEnoughEther();

    constructor() payable {
        richest = msg.sender;
        mostSent = msg.value;
    }

    function becomeRichest() public payable {
        if (msg.value <= mostSent) revert NotEnoughEther();
        pendingWithdrawals[richest] += msg.value;
        richest = msg.sender;
        mostSent = msg.value;
    }

    //RETIRADA MANUAL POR EL ANTIGUO MONARCA
    function withdraw() public {
        uint amount = pendingWithdrawals[msg.sender];
        // Remember to zero the pending refund before
        // sending to prevent re-entrancy attacks
        pendingWithdrawals[msg.sender] = 0;
        payable(msg.sender).transfer(amount);
    }
}
```

Código 16: ejemplo contrato "King of Ether" con patrón de retirada

En el apartado 6.1.3Envío de Ether se recogen consejos para realizar el envío seguro de Ether, profundizando en el uso de *send(), transfer() y call()* así como otros ejemplos del uso del "Patrón de retirada".

5.6 Mala aleatoriedad

5.6.1 La vulnerabilidad

La red Ethereum es determinista y como tal conlleva grandes dificultades a la hora de desarrollar un generador de números pseudo-aleatorios (PRNG, *Pseudo-Random Number Generator* en inglés).

Esto supone un problema a los desarrolladores que intentan crear su propio generador utilizando variables que en un principio pueden parecer una buena fuente de entropía pero que luego no lo son. A continuación se muestran diferentes categorías de origen de datos que producen números aleatorios susceptibles de ser vulnerables. [29]

Generadores basados en variables del bloque

Hay ciertas variables del bloque que pueden ser usadas incorrectamente como fuente de entropía para el generador:

- *block.coinbase*: dirección del minero que ha minado el bloque actual
- *block.difficulty:* dificultad de la red para encontrar el bloque
- *block.gaslimit:* límite máximo de gas a utilizar en el bloque
- *block.number:* número del bloque en la blockchain
- *block.timestamp:* hora en la que se ha minado el bloque

El primer problema con estas variables es que pueden ser manipuladas por el minero del bloque, si este tiene un incentivo para hacerlo.

Por ejemplo si hay un contrato donde la persona que adivine un número se lleve una recompensa económica, el minero puede modificar a su antojo las variables para conocer el número a generar de antemano.

Además todos los contratos del bloque tendrán las mismas variables. Un contrato de un atacante con el mismo generador producirá el mismo resultado y podría realizar una llamada interna al contrato vulnerable.

Generadores basados en el hash de un bloque anterior

Los bloques en la blockchain se identifican por el hash de verificación. Este dato es accesible mediante la función *blockhash(uint blockNumber)*. Aunque pueda parecer un origen aceptable de entropía, tampoco lo es como se especifica en la documentación de Solidity[23].

En primer lugar solo se pueden acceder a el hash de los 256 bloques más recientes. El *blockhash* del bloque actual o más antiguo de 256 bloques,

23 https://docs.soliditylang.org/en/v0.8.17/units-and-global-variables.html#block-and-transaction-properties

devuelven como resultado 0, no pudiendo ser usado entonces como generador.

El uso de *blockhash*(block.blocknumber-1) tiene el mismo problema indicado anteriormente. Otro contrato atacante con el mismo generador puede realizar una llamada interna en el mismo bloque.

Generadores basados en el hash de un bloque futuro

Una posible alternativa es utilizar el *blockhash* de un bloque futuro. En primer lugar se almacena en una variable el número de bloque actual. En el momento de la transacción el hash del bloque actual no se conoce. Pero en el bloque siguiente ya se podría acceder a este valor.

Sin embargo, hay que tener en cuenta que sigue estando la restricción de acceder a el hash de bloques más antiguos de 256. En ese caso el resultado obtenido será siempre 0.

Generadores basados en el hash de un bloque anterior y una semilla

Otras combinaciones utilizan el hash de un bloque anterior y una semilla guardada en una variable privada. El problema que a pesar de que otros contratos no pueden acceder a esta variable de forma directa, los datos en la red *blockchain* son públicos y accediendo al storage de la *blockchain* directamente se pueden obtener el valor de las variables sean privadas o no.

Generadores vulnerables a "front-running"

Estas vulnerabilidad es analizada en detalle en el apartado 5.10. Si se utiliza un oráculo externo para generar un número aleatorio para decidir un ganador, un atacante puede estar analizando las transacciones de la *mempool*.

Cuando observe la transacción del oráculo, el atacante puede generar una transacción pagando más gas para tener más prioridad sobre la del oráculo, haciendo que se ejecute antes sabiendo el próximo número ganador.

5.6.2 Ocurrencias vulnerabilidad

Una de las ocurrencias de esta vulnerabilidad más sonadas fue el contrato SmartBillions Lotery, el cual fue puesto a prueba por la comunidad antes de su lanzamiento oficial en un "hackatlón"[24].

El contrato funcionaba con un generador basado en un bloque futuro, y contenía la vulnerabilidad descrita en ese apartado. Si un usuario proporcionaba una apuesta del número "0", y esperaba 256 bloques para obtener el número ganador, este será siempre "0" al no poder obtener el contrato el hash real.

24 https://www.reddit.com/r/ethereum/comments/74d3dc/smartbillions_lottery_contract_just_got_hacked/

5.6.3 Soluciones

Las soluciones para el desarrollo de un PRNG seguro son complejas y suelen requerir un proceso de varias transacciones.

Oráculos externos

Una posible solución es el empleo de un oráculo externo que proporcione la entropía necesaria para el contrato de forma externa a la *blockchain*. Hay que prestar atención a los problemas de "front-running", que pueden ser mitigados añadiendo una ventana de apuesta y otra ventana donde solo el oráculo pueda interactuar con el contrato.

Como desventaja de esta solución es que utiliza un servicio centralizado, que podría ser manipulado y se desconoce su funcionamiento real.

Signidice

Signidice[25] es un algoritmo que puede ser utilizado para ciertos juegos que funcionan en la blockchain, donde el resultado de cada ronda depende únicamente en el resultado del PRNG y opcionalmente en un número elegido por el jugador. No es un algoritmo válido cuando interactúan varios jugadores sobre un mismo resultado, p.ej un contrato lotería.

Está basado en el intercambio de firmas criptográficas entre el jugador y el contrato.

Patrón "commit-reveal"

Una solución para cuando se necesita un PRNG que funcione con varios usuarios se puede utilizar una implementación de un patrón "commit-reveal". (Ver 6.1.4Solucción a front-running, patrón "Commit-Reveal"")

Este patrón introduce dos ventanas, la primera donde los usuarios envían un secreto encriptado al contrato, y una segunda ventana donde los usuarios revelan los secretos y el contrato comprueba que son correctos. Cuando todos los usuarios son verificados, se emplea los secretos para generar el número aleatorio.

Esto sin embargo supone un problema cuando algún usuario descubre que el número a generar si descubre su semilla no será ventajoso y decide no mostrarla. (Ver siguiente apartado 5.7Denegación de servicio). Hay varias soluciones a este caso, una de ellas es requerir dejar en fianza cierta cantidad de Ether para participar, que será devuelta si se envía la semilla posteriormente, sino el usuario perderá la fianza. Una implementación de esta solución se encuentra en el contrato RanDAO[26]

25 https://github.com/gluk256/misc/blob/master/rng4ethereum/signidice.md
26 https://github.com/randao/randao

5.7 Denegación de servicio

5.7.1 La vulnerabilidad

La categoría de vulnerabilidades de Denegación de Servicio es muy general y amplia, y consiste en ataques donde el contrato puede quedar en un estado inoperable. Debido a la naturaleza de la blockchain, esto conlleva a que los fondos en el contrato queden muchas veces atrapados para siempre.

Alguno casos en los que puede ocurrir:

Bucles actuando sobre *mappings* o arrays

Este problema ocurre cuando un contrato interactúa sobre un *mapping* o array, que pueda ser manipulado externamente por los usuarios. Si el contrato necesita realizar un bucle sobre todo el array o el *mapping* y estos son lo suficientemente grandes, puede ocurrir que no sea posible completar la operación por falta de gas.

Esto dejaría el contrato en un estado de bloqueo ya que todas las transacciones revertirían de forma automática por falta de gas.

Operaciones del creador

Un patrón donde los creadores del contrato tengan ciertos privilegios a la hora de ejecutar transacciones, por ejemplo que se requiera que ejecuten una función para permitir la interacción de usuarios con partes del contrato, puede causar problemas si el usuario privilegiado deja de operar en la red o pierde sus claves.

Destrucción de librerías externas

En la vulnerabilidad analizada en el apartado 5.4, la destrucción de una librería externa provocó la inutilización de un contrato que dependía de la misma.

Continuación del contrato dependiente de llamadas externas

Cuando la continuación de las operaciones de un contrato depende de la ejecución correcta de una llamada externa, por ejemplo el envío de Ether a una cuenta, existe la posibilidad de producir una denegación de servicio en el contrato si se rechazan estas llamadas

Por ejemplo, si la cuenta de destino es un contrato y en su función *fallback* donde recibe Ether rechaza la transacción o consume más gas del disponible, ya sea de forma intencionada o no.

Usuarios no cooperativos

Los contratos que dependan de la acción de varios usuarios para progresar, pueden verse comprometidos en un estado de bloqueo si alguno de ellos

no realiza la suya. Un ejemplo de contrato vulnerable sería uno diseñado para generar números aleatorios a partir de una semilla de varios usuarios usando el patrón "Commit-Reveal". Si uno de ellos no envía la semilla en la etapa de revelado, todo el proceso queda inutilizado.

5.7.2 Ocurrencias vulnerabilidad

Además de la vulnerabilidad del apartado 4.4 relativa a la destrucción de una de las librerías que utilizaba el contrato "Multi-sig Wallet", que provocó su bloqueo, existen numerosas ocurrencias de Denegación de Servicio.

Una de ellas fue en el juego de Ethereum "GovernMental". En este caso para continuar la ejecución del contrato correctamente era necesario borrar dos arrays con el siguiente código:

```
1  creditorAddresses = new address[](0);
2  creditorAmounts = new uint[](0);
```

Este código generaba un bucle que recorría el antiguo array para borrar elemento a elemento. Como los dos arrays eran lo suficientemente largos, requería un coste de gas de 5'057'945, cuando en aquel entonces gas máximo por bloque era de 4'712'388, (Bloque 1405425[27], 26/04/2016).

El gas máximo por bloque es un valor dinámico que puede variar según el porcentaje de gas usado en los bloques anteriores, en un porcentaje cercano al 0.09% de bloque en bloque. Una solución es enviar varias transacciones seguidas, de forma que al utilizar mucho porcentaje del gas disponible en cada bloque, el límite se iría subiendo progresivamente y como sugirió el propio Vitálik Buterin[28].

Tres meses más tarde un usuario aprovechó que el debido al estado de la red, el gas máximo tenía un valor cercano al necesario para desbloquear el contrato. El usuario envió varias transacciones que consumiesen el gas para seguir subiendo este límite, hasta que finalmente pudo completar correctamente la transacción[29].

5.7.3 Soluciones

Debido a la diversidad de errores de programación que pueden dejar el contrato en un estado de bloqueo es dificil dar una solución concisa al problema solo se enumerarán ciertas soluciones a los problemas enumerados

Controlar borrado de arrays

27 https://etherscan.io/tx/0x820828793ab78902212abf997dd9731ef55f1cf7f8c75747076f7565d2d76742
28 https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck/
29 https://etherscan.io/tx/0x0d80d67202bd9cb6773df8dd2020e7190a1b0793e8ec4fc105257e8128f0506b

En el caso de necesitar borrar arrays donde los usuarios tienen cierto control en el tamaño de los mismos, es importante comprobar el tamaño para no generar una transacción que requiera más gas del disponible en un bloque.

Una solución es no borrar directamente no borrar el array, y tener un contador de los elementos del array por separado. Otra solución es dividir el trabajo en tareas más pequeñas más manejables.

(Ver 6.1.5Bucles y borrado de arrays)

Realizar un test exhaustivo del contrato

Si el conjunto de pruebas a realizar es lo suficientemente completo, es posible detectar estos errores en la etapa de pruebas. Es importante probar el contrato con condiciones reales o incluso mayores a las esperadas para detectar posibles problemas.

(Ver 6.2Pruebas de smart contracts)

Añadir función de emergencia

Existe la posibilidad de añadir una función de emergencia, por ejemplo la instrucción *selfdestruct* que destruiría el contrato y enviaría los fondos a una cuenta indicada.

El problema que supone esta solución es que el creador pueda invocar a esta función en cualquier momento y pueda escapar con los fondos de los usuarios.

Participantes no cooperativos

En el caso de participantes no cooperativos hay varias estrategias que se pueden implementar para disminuir la posibilidad de que ocurra un ataque de este tipo.

- Añadir un incentivo económico:

  Los usuarios que participen deberán dejar en fianza una cantidad que perderán si no realizan las acciones requeridas. Adiccionalmente se puede incluir una propina si las acciones son realizadas de forma correcta. Esta solución es implementada por el contrato RandDAO (ver 5.6Mala aleatoriedad)

- Añadir un límite de tiempo:

  Pasado un cierto timestamp en el futuro o un número de bloque, el proceso quedará reseteado de forma automática para evitar bloquear el contrato. Esta medida se puede combinar con la anterior para que no sea rentable realizar un ataque de Denegación de servicio si este no envía su transacción.

5.8 Vulnerabilidades en el cliente

5.8.1 La vulnerabilidad

Debido a la dificultad para un usuario de interactuar directamente con un *smart contract* desplegado, son necesarias aplicaciones cliente que interactúen con el *smart contract* mediante una interfaz amigable. Es tan importante que tanto el *smart contract* como la aplicación con la que interactúa el usuario sean seguros.

Gracias a la interfaz ABI, aplicaciones externas pueden hacer uso de los contratos en la red Ethereum, creando transacciones de forma dinámica utilizando los métodos disponibles en los *smart contracts*. (*Ver apartado 3.4 Generar una llamada a una función)*.

El problema viene cuando la aplicación externa crea de forma incorrecta estas transacciones, lo que resulta en una entrada errónea en el contrato que puede producir comportamientos inesperados.

Notar que esta vulnerabilidad afecta a los contratos inteligentes pero se debe a un uso incorrecto de los mismos en una aplicación externa. Aunque también es posible añadir comprobaciones en el propio contrato para intentar detectar llamadas incorrectas.

5.8.2 Ocurrencias vulnerabilidad

A continuación se analizan dos vulnerabilidades de diferente naturaleza pero producidas en el cliente que interactúa con un *smart contract*. La primera vulnerabilidad afecta a los responsables de la aplicación cliente, mientras que la segunda afecta a los usuarios que interactuaban con la aplicación.

Short address attack

A finales de Marzo de 2017, se detectó una transacción extraña que comprendía el contrato de un token ERC20 llamado GNT [30]. Esta transacción[30] fue emitida por una página de compraventa de criptomonedas (en inglés *exchange*), en la que se invocaba al método *transfer* pero con unos parámetros incorrectos, que permitían que un posible atacante se aprovechase para vaciar las cuentas del *exchange*.

Para invocar a la función *function transfer(address _to, uint256 _value) returns (bool)*, hace falta indicar tres datos:

- Id del método: en este caso 0xa9059cbb
- Parámetro "_to": dato del tipo address de 32bytes, que contiene la dirección de destino

30 https://etherscan.io/tx/0x0213fb70e8174c5cbd9233a8e95905462cd7f1b498c12ff5e8ec071f4cc99347

- Parámetro "_value": dato del tipo uint256, donde se indica la cantidad de tokens a enviar.

El problema estaba en el tratamiento del parámetro "_to". Las direcciones Ethereum tienen un tamaño de 20 bytes. Pero el tamaño de la dirección introducida por el usuario no era comprobada, pudiendo ser menor de estos 20 bytes. Cuando se incluía una dirección que contenía ceros al inicio, por ejemplo la siguiente dirección de 17bytes se puede escribir con los ceros a la izquierda o sin ellos, siendo las dos equivalentes:

```
0xabcdabcdabcdabcdabcdabcdabcdabcdab (versión 17bytes)
0x000000abcdabcdabcdabcdabcdabcdabcdabcdab (versión 20bytes)
```

La aplicación no comprobaba si la dirección introducida por el usuario incluía los ceros a la izquierda o no, y simplemente suponía que la cadena introducida era de 20bytes.

El problema que un parametro del tipo *address* es necesario incluirlo como una cadena de 32bytes, y la aplicación incluía automáticamente 12bytes suponiendo que la cadena era de 20bytes.

```
0x000000000000000000000000abcdabcdabcdabcdabcdabcdabcdabcdab
(versión 17bytes) 17+12 = 29bytes INCORRECTO

0x000000000000000000000000000000abcdabcdabcdabcdabcdabcdabcdabcdab (versión
20bytes) 20+12 = 32bytes CORRECTO
```

Veamos que pasa si intentamos realizar esta transacción con el parámetro *address* generado de forma incorrecta, para tranferier una cantidad de 65535 tokens, es decir 0xFFFF.

La transacción generada será la siguiente:

```
0xa9059cbb000000000000000000000000abcdabcdabcdabcdabcdabcdabcdabcdab0000000
0000000000000000000000000000000000000000000000000FFFF

(Id método, parametro address _to, parámetro uint256 _value)
```

Sin embargo, la EVM interpretará los parámetros de la siguiente forma:

```
0xa9059cbb000000000000000000000000abcdabcdabcdabcdabcdabcdabcdabcdab0000000
0000000000000000000000000000000000000000000000000FFFF000000

(Id método, parametro address _to, parámetro uint256 _value)
```

Hay dos grandes diferencias, la primera es la dirección de destino, que en vez de ser la que se pretendía:

```
0xabcdabcdabcdabcdabcdabcdabcdabcdab, o

0x000000abcdabcdabcdabcdabcdabcdabcdabcdab
```

pasa ahora a ser:

```
0xabcdabcdabcdabcdabcdabcdabcdabcdab000000
```

Por otra parte, la cantidad en vez de ser 0xFFFF, pasa a ser FFFF000000, es decir 1.099.494.850.560 en vez de 65.535, el equivalente a desplazar la cantidad a la izquierda el número de bytes que faltaban de la dirección.

Aunque los parámetros enviados a la EVM tengan una longitud más corta que la esperada (la EVM espera 6 bytes más como entrada), si no son indicados por defecto los interpreta como 0x0.

Esto da como resultado que un usuario que quiera realizar una transacción legitima a su cuenta utilizando el *exchange* con la implementación incorrecta de la llamada, acabe enviando una cantidad mayor de tokens a una dirección completamente diferente, vaciando asi los depósitos del *exchange*.

Ataque BadgerDAO

La siguiente vulnerabilidad a diferencia de la anterior afectó a los usuarios de la plataforma BadgerDAO en Noviembre de 2021 [31], [32]. La plataforma BadgerDAO servía como puente para que los usuarios de Bictoin pudiesen tener acceso a el mundo de las finanzas descentralizadas (DeFi) que funciona sobre Ethereum.

En este caso la página web que servía como Dapp para interactuar con el *smart contract* de la plataforma fue comprometida. Los atacantes obtuvieron acceso al servidor de la página web e inyectaron código JavaScript que alteraba las transacciones que deberían firmar los usuarios para utilizar la aplicación, cambiando la dirección del smart contract sobre el que actuaba.

Cuando un usuario pretendia autorizar una cantidad de dinero para ser gastada en el contrato de BadgerDAO, los atacantes modificaron la petición para que el destino de esta autorización fuese un contrato controlado por ellos.

Esta transacción modificada deberán ser firmadas por el usuario utilizando su wallet. En este punto el usuario podría advertir que no está realizando la petición que deseaba. Sin embargo debido a la ausencia de una descripción detallada de la transacción, ni advertencia de interactuar con una nueva dirección desconocidal, así como la constumbre del usuario de aceptar las transacciones sin prestar su total atención, fueron lo que hicieron posible este ataque.

Los atacantes esperaron a que un cliente víctima con un saldo mayor a $50k interactuase con la aplicación para modificar la transacción. Si el usuario aceptaba esta transacción modificada, todo el saldo de su cuenta estaría comprometido, pudiendo ser vaciado por los atacantes.

El ataque fue satisfactorio y los atacantes fueron capaces de robar más de 120 millones de dólares.

5.8.3 Posible ataque (*short address)*

En el ejemplo dado, las monedas acababan enviadas a una dirección diferente a la indicada, quedando así los fondos perdidos. Sin embargo un atacante podria generar mediante fuerza bruta una dirección de cartera que contenga un número determinado de 0 al inicio de la dirección. Si estos ceros son omitidos al introducir la dirección en la aplicación cliente, la cantidad se desplazará a la izquierda. Además la dirección final será la correcta al "tomar prestado" los primeros bytes de la cantidad introducida, que serán 0.

Cuantos más 0 en la dirección, crecerá exponencialente el número de intentos que necesitará el atacante en hacer fuerza bruta para generar la cartera, pero también lo hará la cantidad de dinero transferida. La cantidad de intentos será 16^z, donde *z* será el número de 0 a contener al final de la dirección.

Con un sencillo programa en python es posible encontrar una cartera con 4 "0" al final de su dirección pública. Serán necesarios una media de 16^4=65536 intentos. La cantidad transferida en el ataque también se multiplicará por ese valor.

Este es el resultado de ejecutar este pequeño script:

```
file: contracts/attack_dapp_wallet_trailing_zeros/gen_wallet_zeros.py

Number of zeros: 4
Expected # of tries: 65536
# of tries: 1000        expected # of tries: 65536
# of tries: 2000        expected # of tries: 65536
# of tries: 3000        expected # of tries: 65536
# of tries: 4000        expected # of tries: 65536
# of tries: 5000        expected # of tries: 65536
# of tries: 6000        expected # of tries: 65536
# of tries: 7000        expected # of tries: 65536
# of tries: 8000        expected # of tries: 65536
# of tries: 9000        expected # of tries: 65536
# of tries: 10000       expected # of tries: 65536
# of tries: 11000       expected # of tries: 65536
# of tries: 12000       expected # of tries: 65536
# of tries: 13000       expected # of tries: 65536
# of tries: 14000       expected # of tries: 65536
# of tries: 15000       expected # of tries: 65536
Public addr: 0x61d41177ad6199E632DBf48006a2D87307520000
Private key:
0xf97d6343914187fd32832ce92254327b4fcd21415a1054c61790bea9f041d325
```

Código 17: salida del script Python "gen_wallet_zeros.py"

Se ha encontrado una dirección que contenga 4 zeros al final:

```
0x61d41177ad6199E632DBf48006a2D87307520000
```

Ahora un atacante podría utilizarla para realizar el ataque y transferirse fondos multiplicados por un valor de 0x10000 o de 65536.

El atacante introduciría como destino en el exchange la dirección generada pero sin incluir los 4 ceros del final, y como cantidad por ejemplo 255 (0xFF)

```
0x61d41177ad6199E632DBf48006a2D8730752
```

La transacción generada sería la siguiente:

```
0xa9059cbb000000000000000000000000061d41177ad6199E632DBf48006a2D873075200000000
00000000000000000000000000000000000000000000000000000000000000FF

(Id método, parametro address _to, parámetro uint256 _value)
```

Sin embargo será interpretada por la EVM de la siguiente forma:

```
0xa9059cbb000000000000000000000000061d41177ad6199E632DBf48006a2D873075200000000
00000000000000000000000000000000000000000000000000000000000000FF0000

(Id método, parametro address _to, parámetro uint256 _value)
```

La dirección real de destino será la creada por el atacante, es decir:

```
0x61d41177ad6199E632DBf48006a2D87307520000
```

Mientras que la cantidad pasaría de ser 255 (0xFF), a ser más de 16 Millones (0xFF0000), ya que el valor se desplaza tantos bytes como los que faltaban en la dirección (2bytes).

5.8.4 Soluciones

A diferencia de otras vulnerabilidades analizadas, esta no se produce en el propio contrato sino en un cliente del mismo. Las soluciones planteadas serán aplicadas mayormente en el cliente:

Verificar las transacciones a firmar (Usuarios)

Los usuarios deberán prestar atención a las transacciones que firman en su *wallet* personal. Esto supone un problema para los usuarios con menos conocimiento técnicos, por lo que los desarrolladores de las *wallet* deberían incidir en las consecuencias de cada transacción.

Una solución es acostumbrar a los usuarios a limitar las cantidades de aprobación para interactuar con un *smart contract*. En muchos casos los usuarios aprueban una cantidad ilimitada para ser gastada a modo de reducir la frecuencia de las peticiones de aprobación, y reducir el gasto en comisiones de la red. Sin embargo en un caso como el de BadgerDAO, están expuestos a perder todo su saldo.

Otra solución es utilizar una cuenta secundaria de Ethereum con un saldo menor con el fin de reducir el daño si se interactúa con un contrato no deseado.

Verificar entradas en el contrato

Una posibilidad es intentar verificar la entrada del usuario en el contrato. Sin embargo muchas veces es imposible diferenciar si la entrada recibida es la pretendida o no como en la vulnerabilidad analizada.

En otras ocasiones como parámetros es fácil añadir una comprobación para que los valores de entrada se encuentren dentro de un rango.

Hay sin embargo un detalle en el orden de los parámetros en el contrato. Como el *padding* solo ocurre al final de la entrada, el orden de los parámetros puede potencialmente mitigar alguna forma de este ataque.

Verificar entrada en el cliente

Desde el cliente es más importante todavía verificar la entrada del usuario ya que es más fácil alertar al mismo sin ocurrir en ningún gasto de transacción.

Tener certeza de codificar correctamente la transacción

Una solución es decodificar de nuevo los parámetros del usuario y compararlos a los originales para así asegurarse se ha codificado correctamente.

Valores de gas de transacción

Se pueden comprobar si valores como gas utilizado, o la dirección de destino de la transacción se corresponde con lo esperado.

5.9 Ether no esperado

5.9.1 La vulnerabilidad

Un patrón de programación defensiva es la validación del estado del contrato mediante comprobación de inmutabilidad. En esta técnica se definen un conjunto de parámetros inmutables que no deberían cambiar según se ejecuta el contrato.

Sin embargo esta técnica puede ser implementada erróneamente cuando el desarrollador realiza un uso incorrecto de la variable *this.balance,* si asume que el contrato solo puede recibir Ether a través de las funciones establecidas en el contrato. Esta falsa suposición puede dar a cabo resultados inesperados.

Cuando Ether es enviado a un contrato, debe ser a través de las funciones predeterminadas *fallback* o *receive* o a través de una función con el modificador *payable*. Sin embargo hay dos excepciones a esta regla:

Selfdestruct

Cuando en un contrato se invoca la función *selfdestruct(address _to)*, los fondos del contrato son enviados a la cuenta indicada. En el caso que la cuenta de destino sea un contrato, este recibirá los fondos sin ejecutar ninguna función del mismo. Incluso aquellos contratos que no contengan funciones del tipo *payable*.

Ether enviado antes de la creación del contrato

Existe otra forma de enviar fondos a un contrato sin ejecutar su código, y es enviar el Ether antes de que se despliegue el contrato. Las direcciones del contrato dependen únicamente de la dirección de la cuenta de despliegue, y su *nonce*. Por lo tanto es posible precalcular las posibles direcciones de los contratos desplegados por una cuenta y enviar el Ether antes de que la misma realice el despliegue.

Este enfoque es más complejo de llevar a cabo ya que es difícil conocer cuando la cuenta va a desplegar un contrato.

5.9.2 Ocurrencias vulnerabilidad

No se han divulgado vulnerabilidades de este tipo producidas en el mundo real, aunque esto no significa que no se hayan producido.

Una prueba de concepto de esta vulnerabilidad se presenta en el documento “Ethereum Smart Contract Security Best Practices”[33]

Source:
https://consensys.github.io/smart-contract-best-practices/attacks/force-feeding/

```
1  pragma solidity ^0.8.13;
2
3  contract Vulnerable {
4      receive() external payable {
5          revert();
6      }
7
8      function somethingBad() external {
9          require(address(this).balance > 0);
10         // Do something bad
11     }
12 }
```

Código 18: ejemplo de contrato vulnerable a envío forzado de Ether (source: consensys.github.io)

En este ejemplo, la ejecución de la función *somethingBad()* está restringida a que el contrato disponga de saldo. En teoría no es posible porque el contrato revierte las transferencias en su función "receive()".

Sin embargo si un atacante crea un contrato con Ether, y lo destruye mediante "*selfdestruct*()", de forma que el destinatario sea el contrato "Vulnerable", la transferencia se completará sin provocar un "revert()", y se podrá ejecutar la función "*somethingBad*()".

5.9.3 Soluciones

Esta vulnerabilidad ocurre cuando se realiza un uso incorrecto de la variable *this.balance* en conjunto a funciones que prohíban o registren la cantidad de Ether del contrato.

Es importante tener en cuenta que el contrato puede recibir Ether sin necesidad de ejecutar ninguna función por la técnicas presentadas, incluso cuando *fallback()* o *receive()* rechacen transferencias.

5.10 Front running o condición de carrera

5.10.1 La vulnerabilidad

El "front-running" o inversión ventajista es una práctica ilegal en los mercados de bolsa, donde un corredor de bolsa que actúe en nombre de terceros y sabe que órdenes de compraventa se van a producir que pueden alterar el precio del mercado, y utilizan esa información para su propio beneficio.

En el ámbito de las criptomonedas, una situación similar puede ocurrir cuando un usuario envía una transacción a la "*mempool*" para ser posteriormente procesada por un nodo para incluirla en la blockchain. En ese periodo de espera, un tercero puede enviar una transacción pagando una comisión de red mayor, para tener ventaja y que su transacción se tramite antes que la del usuario original.

Esta vulnerabilidad puede ser dependiendo del contrato por terceros usuarios malintencionados, o por los propios mineros del bloque. Cuando un *smart contract* es susceptible a un ataque por terceros usuarios es más grave que cuando es realizado por un minero, ya que es menor la probabilidad de que un minero en concreto sea el elegido para minar un bloque en concreto.

La ocurrencia más común de esta vulnerabilidad es en contratos que actúen como servicios de intercambio de criptomonedas descentralizados, aunque otros contratos de otra naturaleza pueden ser también vulnerables. Por ejemplo en un contrato donde el usuario que envíe una solución a un problema criptográfico gane una cantidad de criptomonedas.

Si un usuario envía una solución, otro diferente puede enviar la misma transacción pero con una comisión mayor para que su transacción sea incluida antes que la original y así robar el premio.

5.10.2 Ocurrencias vulnerabilidad

Se van a detallar dos vulnerabilidades diferentes de "front-running". La primera más tradicional sobre un contrato *exchange* descentralizado, y la segunda más específica a el orden de las transacciones en un smart contract.

Exchange descentralizado: Bancor

Una ocurrencia de esta vulnerabilidad se ha dado en el smart contract "Bancor", el cual implementa un protocolo para el intercambio de principalmente tokens ERC-20.

El contrato ajusta el precio de intercambio de cada Token según la cantidad de órdenes de compra y de venta. Si se introduce una orden en la "mempool" lo suficientemente grande, se va a producir un cambio en el

precio de los tokens. Esto da pie a que si un usuario monitora la "*mempool*" en busca de este tipo de transacciones, será capaz de enviar una transacción con mayor comisión para que se ejecute primero y proporcionarle un beneficio económico.

Si el precio de un token va a subir de valor, se puede realizar una orden de compra, esperar a que la orden del tercero se ejecute (lo que subirá el precio), y ejecutar posteriormente otra orden de venta a un precio mayor.

El perjudicado en este caso será el usuario de la orden original. Ya que en el caso de una compra, el usuario comprará el token a un precio mayor al estipulado, al producirse otra compra justo antes de la suya.

Un grupo de investigadores puso a prueba este concepto mediante un script de Python, que realizaba estas tareas de forma automatizada, probando el concepto. [34]

Bancor aplicó ciertas medidas para contener esta vulnerabilidad. Una de ellas establecer un parámetro en la transacción, que la cancele si el el precio del token ha variado. Como problema tiene que el atacante conocerá ya las intenciones del comprador, que posiblemente realice la orden de nuevo en un futuro cercano.

Otra solución sugerida fue establecer un precio máximo de gas por transacción hacia el contrato, rechazando transacciones que superen este límite. De esta forma, otros usuarios no puede aumentar el gas para dar prioridad a la misma. Esta solución reduciría el vector de ataque, tan solo los mineros podrían reordenar las transacciones al crear un nuevo bloque.

Estándar ERC-20

Esta vulnerabilidad en el estándar ERC20[31] para crear Tokens dentro de la red Ethereum, permite realizar una ataque de front-running en la función *approve()*.

Esta función *approve()* permite autorizar el gasto de una cantidad indicada de tokens a un tercero utilizando la cuenta del propio usuario que autoriza. El resultado de llamar a esta función es actualizar la cantidad que el usuario puede gastar. Por ejemplo, si Alice es la dueña de la cuenta y había autorizado a Bob a gastar 100 tokens en un pasado. Si Alice desea autorizar una cantidad mayor, por ejemplo 150 tokens, o incluso revocar esta autorización, Bob podría realizar un ataque de front-running para extraer más tokens de los que Alice tenía pensado autorizar.

En el caso de que Alice revoque los tokens, Bob podría realizar una operación *transfer* de los tokens a su cuenta antes de que se ejecute la transacción de Alice. De forma que la revocación se realizaría demasiado tarde.

31 https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md

De manera similar, en el caso de que la cantidad sea modificada, por ejemplo aumentada de 100 a 150 tokens, Bob podría realizar un ataque que le permita retirar los 100 tokens antes de que se ejecute la operación approve, y a continuación retirar los 150 tokens después de la ejecución. Gastando de esta forma 250 tokens en vez de 150 que Alice planeaba autorizar.

En este ultimo caso una posible mitigación consistiría en realizar un approve con cantidad 0, y otro con la cantidad deseada. De esta forma si Alice detecta que Bob retiró los 100 tokens, ya no autorizará a retirar los 150 tokens a continuación.

5.10.3 Soluciones

Diferentes tipos de "front-running" tienen diferentes soluciones según el contexto en el que ocurran:

Operaciones atómicas "comparar y modificar":

En el caso de la vulnerabilidad en la función *approve* de ERC20, existe una solución que consiste en añadir un parámetro más para añadir el valor actual de la variable a modificar (la cantidad que puede retirar la cuenta indicada). Esta solución también fue aplicada en el contrato Bancor, si el precio del token variaba, se cancelaría la orden.

Al ejecutar Alice la transacción, indicará que la cantidad anterior aprobada era de 100 Tokens, si en el momento de ejecutarse esta función, la cantidad no coincide, porque Bob ha realizado un ataque de front-running, la transacción de Alice será revertida y no se aprobarán nuevos fondos.

Admitir un precio máximo de gas

En contratos como Bancor, una solución para evitar un ataque de front-running por parte de los usuarios, es añadir un precio máximo de gas en la transacción.

En la actualidad el precio base a gastar por transacción está regulado por la red Ethereum y se puede acceder en la variable "block.basefee", mientras que "tx.gasprice" contiene el precio de gas pagado por el emisor. La diferencia de estas dos variables corresponde a la donación o incentivo que se llevarán los mineros para darle prioridad a la misma. Por lo tanto es fácil limitar este incentivo, por ejemplo, a un máximo de 10 gwei:

```
1  uint tipFee = tx.gasprice – block.basefee;
2  require( tipFee <= 10 gwei )
```

Patrón "commit-reveal"

Este método más robusto ya defiende contra mineros y usuarios por igual. Consiste en dividir las transacciones en dos etapas diferentes. En la primera "commit", se enviará una información de la transacción de forma encriptada,

por ejemplo mediante un hash de la misma. En la segunda "reveal", se envía la comprobación de la información en claro.

Se puede requerir esperar una cantidad de bloques entre el "commit" y el "reveal", de forma que sea imposible para un "front-runner" generar la segunda transacción a tiempo. Cuando el usuario revele los datos en la segunda operación, el atacante deberá esperar esta cantidad de bloques para poder enviar su segunda transacción para que sea válida.

Otra variación sería enviar una cantidad de Ether arbitraria en la primera transacción, mientras que en la segunda se indicaría al contrato la cantidad real que se quiere gastar, devolviéndose el Ether no empleado.

(Ver 6.1.4Solucción a front-running, patrón "Commit-Reveal")

5.11 Manipulación de tiempo

5.11.1 La vulnerabilidad

En un contrato de Ethereum, no hay forma de conocer la hora de forma precisa. La aproximación más cercana es utilizar la variable *block.timestamp* o su alias *now*. Esta variable proporciona la hora en la que se creó el bloque, el problema que es una variable que puede ser manipulada por los mineros si tienen incentivo suficiente de hacerlo

En la creación de un nuevo bloque, el único requisito de Ethereum para la validación del sello de tiempo "*timestamp*", es que el nuevo sello tenga un valor mayor o igual al anterior, es decir, no se pueden crear bloques con un *timestamp* en el pasado. Sin embargo no hay limitación de cuanto en el futuro puede ser un *timestamp* de un nuevo bloque.

Aunque no exista esta limitación del *timestamp* en el estándar Ethereum, en la práctica si que existe un límite implementado en los protocolos de validación. Antes del "merge", cuando los bloques eran minados mediante PoW y el tiempo entre bloque y bloque podía variar según la "suerte" de los mineros, este límite era de 15 minutos, aunque la media real entre bloque y bloque debería ser aproximadamente 13 segundos.

En la actualidad con la llegada de "PoS", los bloques son minados en una media de 12.06 segundos, aunque ahora la variabilidad es muy reducida. El límite de tiempo en los protocolo de validación se ha reducido a los 15 segundos.

Por lo tanto, si se crea un bloque con un *timestamp* en el futuro sobrepasando este límite, será marcado como inválido por la red hasta que se cree otro bloque con un *timestamp* correcto, donde entonces el bloque inválido pasará a a ser descartado definitivamente.

En resumen, en la actualidad los mineros tienen menor margen de maniobra para perpetrar este ataque, pero es necesario tener en cuenta el origen de la variable *block.timestamp* a la hora de desarrollar contratos donde la exactitud temporal sea un apartado crítico para el correcto funcionamiento del mismo.

Como nota adicional, es importante no utilizar la variable *timestamp* como fuente de entropía para generar números aleatorios. (Ver 5.6)

5.11.2 Ocurrencias vulnerabilidad

El juego Ponzi governmental analizado en el apartado de Denegación de Servicio (Ver 5.7), también era potencialmente vulnerable a un posible ataque de manipulación de tiempo, aunque en práctica sería complicado llevarlo a cabo, ya que el juego se reseteaba cuando pasasen más de 12 horas.

Una manipulación tan grande de tiempo provocaría un estancamiento en la red, al no poder minarse nuevos bloques con un *timestamp* correcto en el pasado, y tendrían que usar un *timestamp* incorrecto para poder ser válidos.

5.11.3 Soluciones

Variable *block.number*

```
1  private uint timerBlockNumber;
2
3  //Inicio del contador
4  function public startTimer(uint _secondsTimer, uint _blockTimeMean){
5      uint blocksDiff = _secondsTimer / _blockTimeMean;
6      timerBlockNumber = block.number + blocksDiff;
7  }
8
9  //Devuelve true si han pasado los bloques necesarios
10 function public checkTimer() returns(bool){
11     if ( block.number >= timeBlockNumber ){
12          return True;
13     }
14     return False;
15
16 }
```

Código 19: ejemplo uso de block.number en vez de block.timestamp

Se podría utilizar la variable *block.number* que devuelve la altura del bloque actual en la cadena de la red para realizar validaciones de tiempo. Actualmente se sabe que cada bloque tarda de media 12.06 segundos. si se necesita realizar una espera de 120 segundos, es tan sencillo como dividir los segundos entre el tiempo de bloque, para conocer los bloques que se deberán esperar:

Esta solución tiene ciertas desventajas, la principal que el contador perderá precisión, ya que el tiempo por bloque sufre pequeñas variaciones, y puede cambiar si la red sufre una actualización (por eso es un parámetro de entrada en el código sugerido).

Sin embargo si no se requiere de la precisión es una práctica recomendable utilizar el número de bloque para cambios importantes de estado dentro de un contrato. Esta solución se emplea a la hora de lanzar actualizaciones en la red Ethereum, donde el momento de realizar el cambio viene determinado por un número de bloque en contrapartida de utilizar una fecha y hora.

5.12 Declaración incorrecta de variables

5.12.1 La vulnerabilidad

Cuando se declaran variables en Solidity, es necesario especificar un modificador de visibilidad, y en algunos casos también es necesario especificar el lugar donde se almacenará dicha variable.

Uso incorrecto variables privadas

En el primer caso pueden ocurrir vulnerabilidades si el desarrollador tiene la falsa creencia de que declarar una variable como privada la hace inaccesible al resto de los usuarios. Esto es falso y lo único que evita es que otro contrato pueda acceder a esta variable mediante a este.

Un usuario común pueda inferir la información almacenada en una variable privada de dos maneras:

La primera recreando la transacción emitida originalmente en la *blockchain* de forma pública, que causó la modificación de la variable.

La segunda más sencilla, es acceder al almacenamiento "*storage*" del contrato, donde se almacenan todas las variables públicas y privadas. Dado que estas se incluyen en el orden en el que están declaradas, es fácil reconocer que posición en el *storage* ocupa cada variable.[32]

El siguiente script permite el acceso al *storage* del contrato Rubixi analizado previamente.

file: contracts/private_variables/access_storage.js

```
let urlRCP = "https://rpc.ankr.com/eth"
let rubixiAddr = "0xe82719202e5965cf5d9b6673b7503a3b92de20be"

const Web3 = require('web3');
const provider = new Web3.providers.HttpProvider(urlRCP);
const web3 = new Web3(provider);

async function getStorage(address, slots){
    for( let i=0; i<slots; i++ ){
        result = await web3.eth.getStorageAt(address,i);
        console.log("["+i+"]: "+result);
    }
}

getStorage(rubixiAddr,7)
```

Código 20: script para acceder al "storage" de un contrato en la "mainnet"

32 https://docs.soliditylang.org/en/v0.8.17/internals/layout_in_storage.html

```
OUTPUT
$ node contracts/private_variables/access_storage.js
[0]: 0x0000000000000000000000000000000000000000000000000000029adbd7732a56b7e
[1]: 0x0000000000000000000000000000000000000000000000000000000000000000
[2]: 0x0000000000000000000000000000000000000000000000000000000000000000
[3]: 0x000000000000000000000000000000000000000000000000000000000000012c
[4]: 0x0000000000000000000000000000000000000000000000000000000000000025
[5]: 0xdf81b3347711cb1176f49027dcfbd303fa5545d681f24c6b64f252f8523b3ced
[6]: 0x0000000000000000000000000000000000000000000000000000000000000096
```

Código 21: Salida del script para conocer el "storage" del contrato "Rubixi"

Cada uno de los datos en los *slots* se corresponde con las variables declarada en el contrato:

```
 1  uint private balance;              //SLOT[0]
 2  uint private collectedFees;        //SLOT[1]
 3  uint private feePercent;           //SLOT[2]
 4  uint private pyramidMultiplier;    //SLOT[3]
 5  uint private payoutOrder;          //SLOT[4]
 6  address private creator;           //SLOT[5]
 7
 8  struct Participant {
 9          address etherAddress;
10          uint payout;
11  }
12
13  Participant[] private participants; //SLOT[6]
```

Código 22: variables declaradas en el contrato Rubixi

El caso de los arrays dinámicos y *mappings* es algo más complejo. En este caso el dato almacenado en el *slot* 6, indica el tamaño del array dinámico. En esta caso hay 0x96= 150 participantes.

Para acceder a un elemento de un array dinámico se aplica la siguiente fórmula para conocer su *slot*:

```
slot_pos= keccak256( slot_array) + array_pos * slot_size
```

Para el array participants, el *slot_size* sería 2, ya que cada struct Participant ocupa dos *slots* (address y uint). La variable *slot_array* sería *6,*ya que el array ocupa el slot 6.

Conociendo que hay 150 participantes, el último participante se encontrará en el slot:

```
keccak256(6)+(149*2)= keccak256(6)+298
```

Punteros de memoria no inicializados

Otra vulnerabilidad posible a la hora de declarar variables dentro de una función es definir incorrectamente su localización. Hay tres posibilidades:

- *calldata:* solo es posible para argumentos de funciones del tipo external, y quiere decir que se accederá a ellas desde el propio calldata de la llamada.
- *memory:* almacenada de forma temporal en la memoria de la EVM.
- *storage:* almacenada de forma permanente en el almacenamiento del contrato.

En versiones pasadas de Solidity, las variable locales eran almacenadas en *memory o storage* de forma predeterminada según su tipo. Esto provocaba que una variable del tipo *storage* no inicializada, pudiese apuntar a una variable global del contrato, y con ello provocando una vulnerabilidad que puede ser intencionada (si el desarrollador la realiza de forma consciente), o desintencionada.

En la actualidad el compilador requiere indicar el lugar de memoria donde se almacena una variable, además de advertir si se corre el riesgo de sobreescribir una variable global si no se inicializa la variable.

5.12.2 Ocurrencias vulnerabilidad

No se conocen casos de vulnerabilidades producidas de alguno de los dos casos presentados.

Se presenta a modo de ejemplo, el siguiente fragmento de contrato que emplea una variable privada como fuente de entropía de forma incorrecta. Como es posible conocer el valor de esta variable, es fácil conocer el número que generará la función y convertirse en ganador.

Source: https://dasp.co/#item-6a

```
1  uint256 private seed;
2
3  function play() public payable {
4    require(msg.value >= 1 ether);
5    iteration++;
6    uint randomNumber = uint(keccak256(seed + iteration));
7    if (randomNumber % 2 == 0) {
8          msg.sender.transfer(this.balance);
9    }
10 }
```

Código 23: ejemplo del uso incorrecto de una variable privada
source: dasp.co

5.12.3 Soluciones

Una vez conocido el funcionamiento de la memoria en la EVM, es sencillo aplicar los conocimientos para no cometer estas vulnerabilidades:

- No suponer que las variables declaradas como privadas son inaccesibles al resto de usuarios

- Elegir correctamente el lugar de almacenamiento de las variables e inicializarlas. Este error es más difícil de cometer en la actualidad gracias a las múltiples advertencias del compilador y la necesidad de declarar explícitamente el lugar de almacenamiento de la variable.

6. Desarrollo seguro de smart contracts

Después de analizar las vulnerabilidades y sus causas más comunes, se propone una guía con buenas prácticas y consejos para mejorar la calidad del código y reducir las posibilidades de cometer errores de programación que produzcan vulnerabilidades.

Además se revisarán métodos de prueba del código a través de test manuales y test automatizados con el apoyo de herramientas específicas.

6.1 Ejemplos de programación segura

En este apartado se revisarán algunas de las soluciones para las vulnerabilidades con una explicación más detallada, con ejemplos de su implementación.

Los ejemplos de código se incluirán en el repositorio de código anexo.

6.1.1 Versión actualizada del compilador

Solidity es el compilador para *smart contracts* de Ethereum más popular. En sus diferentes versiones ha añadido progresivamente funcionalidad para mejorar la seguridad de los *smart contracts*, así como corregir errores del compilador presentes.

Es por ello importante utilizar una versión actualizada y revisar las últimas recomendaciones de uso y seguridad en la documentación oficial[33]. Algunos de los cambios más relevantes que ayudaron a mejorar la seguridad son los siguientes:

- Protección ante desbordamientos aritméticos (*underflow* y *overflow*)
    - En la v.0.8.0 se introdujo el chequeo de los desbordamientos en las operaciones aritméticas. Si se produce un *underflow* o un *overflow*, la operación revertirá, en vez de "envolver" el resultado.

      Previo a este cambio, era necesario incluir librerías o realizar comprobaciones en cada operación.
- Punteros de memoria no inicializados
    - En la v.0.5.0, se convirtió en obligatorio especificar el lugar de almacenamiento para variables no inicializadas. Si no se indica, provocará un error de compilación. Además se advierte al usuario de la posibilidad de sobreescribir variables del "storage" del contrato.
- Visibilidad variables y funciones por defecto
    - Desde la v.0.5.0, es obligatorio indicar modificador de visibilidad para todas las funciones del contrato.

33 https://docs.soliditylang.org/en/latest/

- Declaración de constructores
  - Desde la v.0.4.22, el constructor de un contrato se declara mediante el identificador "*constructor(args)*". Previamente se declaraba con el nombre del contrato: "*function ContractName(args)*", lo que podía provocar vulnerabilidades si el nombre del contrato cambiaba y no se realizaba el cambio en el constructor.
- Restricción de interacción con librerías
  - Antes de v.0.4.20, era posible realizar llamadas a una librería de forma directa como un contrato más. A partir de esa versión, se realiza una comprobación de la dirección del contrato sobre el que se está actuando, eliminando efectivamente las llamadas que modifiquen el estado de un contrato librería.

6.1.2 Llamadas externas – evitar ataques reentrada

Las vulnerabilidades de reentrada se pueden dar al realizar una llamada externa, ya sea invocando una función de un contrato externo, o enviar Ether a una dirección mediante la función *call()*.

Hay dos recomendaciones solventarlo completamente: implementar el patrón "comprobar, modificar, interactuar", o implementar un cierre de exclusión (en inglés "*mutex*"), que sirva como bloqueo a la reentrada.

A continuación se muestra un ejemplo de un contrato simple que actúa como banco y es vulnerable a un ataque de reentrada:

```
file: contracts/reentrancy/bank_unsafe.sol

20  //Request back funds
21  //NOT SAFE TO REENTRANCY
22  //NO MUTEX IN USE
23  //UPDATES STATE AFTER MAKING EXTERNAL CALL
24  function request(uint _value) public {
25      //Check enough balance
26      require(balances[msg.sender] >= _value);
27
28      //Send funds VULNERABLE TO REENTRANCY
29      (bool success,) = msg.sender.call{value: _value}("");
30      require(success);
31
32      //update balance UPDATED AFTER CALL, VULNERABLE TO REENTRANCY
33      balances[msg.sender] -= _value;
34  }
```

Código 24: fragmento contrato vulnerable a reentrada

El balance del usuario no se actualiza hasta completar la llamada *call()*, lo que permite el ataque de reentrada.

Patrón "Comprobar-Modificar-Interactuar"

Este patrón tiene como objetivo evitar ataques de reentrada y consiste en dividir las acciones realizadas en una función en tres grupos y en este orden:

- **Comprobar**: si se necesitan realizar comprobaciones se deben realizar al principio de la función.
- **Modificar**: una vez realizadas las comprobaciones se podrá modificar el estado del contrato.
- **Interactuar**: Si se necesitan realizar llamadas externas, siempre serán al finalizar las comprobaciones y sobretodo las modificaciones oportunas en el estado del contrato.

Después de las llamadas externas no se debería volver a modificar de nuevo el estado del contrato, aunque si que se puede comprobar el valor devuelto en la llamada externa y revertir toda la transacción si no es correcto.

Un error común es esperar al resultado de las llamadas externas para modificar el estado del contrato, lo que posibilita un ataque de reentrada.

En el siguiente fragmento de código, se implementa este patrón en el contrato anterior. En concreto se hace inciso en la división de estas tres etapas: comprobaciones, modificaciones e interacciones. Después de la interacción externa no se permite cambios en el estado del contrato, aunque si se puede comprobar si la llamada fue completada correctamente:

file: contracts/reentrancy/bank_with_pattern.sol

```
18  //Request back funds
19  //SAFE TO REENTRANCY
20  //USES PATTERN CHECK, EFFECTS, INTERACT
21  function request(uint _value) public {
22      //1-CHECK: Check enough balance
23      require(balances[msg.sender] >= _value);
24
25      //2-EFFECTS: update balance
26      balances[msg.sender] -= _value;
27
28      //3-INTERACT: Send funds
29      //Note: no more state changes after this
30      (bool success,) = msg.sender.call{value: _value}("");
31      require(success);
32  }
```

Código 25: fragmento contrato con solución a reentrada (patrón)

En funciones más complejas donde se divida la lógica en varias funciones internas, como en el caso del contrato DAO, es más importante vigilar que se sigue este patrón en el conjunto de las operaciones. Después de realizar una llamada externa, no se deberá modificar el estado del contrato desde la

función que realiza la llamada, ni desde ninguna función padre del mismo contrato.

"Mutex" de reentrada

Otra solución no excluyente de la anterior, es implementar un "*mutex*" que impida ejecutar la función de nuevo hasta que se complete en su totalidad. La principal desventaja es que si se implementa de forma incorrecta, se corre el riesgo de bloquear el acceso a la función a proteger para siempre.

Un ejemplo de esta solución es el contrato "*ReentrancyGuard*" desarrollado por OpenZeppelin[34]. En el se implementa el modificador *nonReentrant()* que utiliza una variable como mutex para impedir la reentrada. Hasta que no finalice la función por completo, no se podrá ejecutar esta función de nuevo u otra función con este modificador.

En el siguiente fragmento, se implementa esta solución en el contrato de ejemplo. Notar la presencia del modificador *nonReentrant()*, que comprueba y configura el mutex antes de ejecutar la función, y lo resetea al finalizar.

```
file: contracts/reentrancy/bank_with_pattern.sol

21  //Request back funds
22  //Safe because it implements mutex in nonReentrant modifier
23  //Cant call request again until it finishes
24  function request(uint _value) public nonReentrant(){
25      //Check enough balance
26      require(balances[msg.sender] >= _value);
27
28      //Send funds:
29      (bool success,) = msg.sender.call{value: _value}("");
30      require(success);
31
32      //update balance
33      balances[msg.sender] -= _value;
34  }
```

Código 26: fragmento contrato con solución a reentrada (mutex)

6.1.3 Envío de Ether

Enviar criptomonedas desde un *smart contract* es una de las acciones que más impacto pueden tener si se realiza incorrectamente. Hay que tener varios aspectos en cuenta sobre el funcionamiento de las funciones para el envío de *Ether* para no cometer errores.

Métodos disponibles en Solidity

Solidity proporciona tres métodos para transferir *Ether* a una dirección.

34 https://docs.openzeppelin.com/contracts/4.x/api/security#ReentrancyGuard

	send()	*transfer()*	*call()*
Resultado si falla la transferencia	-Devuelve False -No hay revert()	-revert transacción	-Devuelve False -No hay revert()
Gas permitido	2300	2300	Limitado al de la transacción original, salvo indicado un valor

Tabla 3: métodos envío Ether

Las funciones *send()* y *transfer()* se introdujeron tras el ataque al DAO para evitar cometer vulnerabilidades de reentrada al limitar el gas. La desventaja principal es que si el destinatario es un *smart contract,* la transferencia puede fallar por esta limitación de gas.

La función *call()* permite más libertad al programador, controlando la acción a realizar si falla la transferencia y permite definir la cantidad de gas. Si se utiliza en conjunto con el "patrón Comprobar-Modificar-Interactuar"[35] y con el "patrón de retirada", se pueden implementar *smart contracts* seguros a ataques.

Un ejemplo simple de uso de las tres funciones se incluye en el repositorio de código: "contracts/send_ether/send_ether.sol".

Patrón de retirada

El patrón de retirada pospone el envió de Ether para que el usuario destinatario sea el que realice una retirada en vez de enviar el Ether directamente a causa de una acción en la misma función. Esto disminuye la posibilidad de introducir problemas de bloqueo de estado del contrato, así como evitar ataques de reentrada si se aplica conjuntamente el patrón "comprobar modificar interactuar"

35 https://docs.soliditylang.org/en/v0.8.17/security-considerations.html#use-the-checks-effects-interactions-pattern

A continuación se muestra un ejemplo sencillo de este patrón.

file: contracts/send_ether/withdraw_pattern_simple.sol

```
 5  contract WithdrawPatternSimple{
 6
 7      mapping(address => uint) public balances;
 8
 9      // Add balance to account
10      function donateTo(address _to) public payable{
11          balances[_to] += msg.value;
12      }
13
14      //withdraw available funds
15      //USE CHECK EFECTS INTERACTION pattern
16      function withdraw() public{
17          //1: checks
18          require(balances[msg.sender]>0,"No funds to withdraw");
19
20          //2: effects - modify state of contract
21          uint userBalance = balances[msg.sender];
22          balances[msg.sender] = 0;
23
24          //3: interaction - external calls, no effects after this
25          (bool ok,) = payable(msg.sender).call{value: userBalance}("");
26          require(ok);
27      }
28
29
30      function getBalanceOf(address addr) public view returns(uint){
31          return balances[addr];
32      }
33  }
```

Código 27: ejemplo contrato simple con patrón de retirada

En vez de realizar una transferencia directamente en la función *donateToo(),* se añade esta cantidad a un mapping que almacena el saldo de cada dirección. Posteriormente, será el usuario que desee retirar el dinero, el que llame a la función *withdraw()* para obtenerlo. Al usar *call()* es importante aplicar el patrón "comprobar-modificar-interactuar" para evitar ataques de reentrada.

- ***Ejemplo contrato subasta sin patrón de retirada:***

```
file: contracts/send_ether/auction_unsafe.sol

24  //Place new bid
25  //-value sent should be higher by at least 0.5eth than previous bid
26  //-previous bidder gets bid back
27  function placeBid() public payable{
28      require(timestamAuctionCloses < block.timestamp, "Auction
closed");
29      require(msg.value > highestBid + 0.5 ether, "bid should at least
0.5eth higher than previous");
30
31      //Return previous bid
32      uint amount = highestBid;
33      highestBid = 0; //set to 0 to avoid doble spending by reentrancy
34      (bool success,) = highestBidder.call{value: amount}("");
35      require(success, "Error sending back funds");
36
37      //Store new bid
38      highestBid = msg.value;
39      highestBidder = msg.sender;
40  }
```

Código 28: contrato subasta sin patrón de retirada (vulnerable)

A mayores se presentan dos ejemplos más completos de un contrato de subasta. En el primero no se aplica el patrón de retirada mientras que en el segundo si.

En este ejemplo, a pesar de reenviar todo el gas disponible, y configurar las variables de forma correcta para evitar un ataque de reentrada, si el último pujador fue un *smart contract* que no admite recibir transferencias (no contiene las funciones *receive* o *fallback)*, ningún nuevo postor podrá completar su puja, ya que revertirá toda la transacción.

De esta forma ningún usuario podrá superar la oferta emitida por el *smart contract*.

- ***Ejemplo subasta con patrón de retirada:***

file: contracts/send_ether/auction_withdraw.sol

```
28  //Place new bid
29  //-value sent should be higher by at least 0.5eth than previous bid
30  //-previous bidder gets bid back
31  function placeBid() public payable{
32      require(timestamAuctionCloses < block.timestamp, "Auction
closed");
33      require(msg.value > highestBid + 0.5 ether, "new bid be
higher");
34
35      //Allow previous bidder to withdraw his last bid
36      balances[highestBidder] += highestBid;
37
38      //Store new bid
39      highestBid = msg.value;
40      highestBidder = msg.sender;
41  }

(...)

55  function withdrawBids() public{
56      //1: checks
57      require( balances[msg.sender] > 0 , "Sender has no balance left
to withdraw");
58
59      //2: effects
60      uint amount = balances[msg.sender];
61      balances[msg.sender] = 0;
62
63      //3: interactions
64      (bool success,) = msg.sender.call{value: amount}("");
65      require(success, "Error sending funds");
66  }
```

Código 29: ejemplo contrato subasta con patrón de retirada (seguro)

Este ejemplo implementa el patrón de retirada en conjunto al patrón "comprobar, modificar, interactuar". Al realizar una puja correcta, simplemente se añade la cantidad del último pujador a un *mapping* que lo asocia con su dirección.

Este patrón de retirada tiene varias ventajas:

- reduce la complejidad de la función: se reduce la cantidad de acciones a realizar en una misma función, simplificando el código. Antes la función se encargaba de configurar la nueva puja y realizar la devolución. Ahora solamente configura la nueva puja.
- imposibilita ataques de reentrada: en la función *placeBid()* no se realizan llamadas externa, lo que imposibilita un ataque de reentrada en esa función si se comete un error de programación.
- Imposibilita ataques de DoS: al separar la acción de retirada en una función aparte que debe llamar el último pujador, se retira la dependencia de completar la devolución para que continúe la

ejecución del contrato, evitando así ataques de DoS como el explicado anteriormente.

En la función *withdrawBids()* reiterar que se aplica el patrón "comprobar, modificar, interactuar" para evitar la reentrada.

6.1.4 Solucción a *front-running,* patrón "Commit-Reveal"

El patrón "Commit-Reveal" permite solucionar algunos de los problemas que se generan en una red de computación descentralizada como el *front-running* o la generación de números aleatorios.

Para la generación de números aleatorios ya existe una implementación completa de una solución para generar números aleatorios entre varias partes, RanDAO, que utiliza este patrón "Commit-Reveal". (Ver 5.6Mala aleatoriedad).

En el caso del *front-running* lo que se desea conseguir con este patrón es realizar una acción en un contrato, retrasando la revelación de un dato clave para que nadie se pueda adelantar a nuestra petición. (Ver 5.10Front running o condición de carrera).

A continuación se muestra un ejemplo de un contrato que permite realizar una subasta a ciegas sin que los usuarios que participen puedan conocer las pujas de sus rivales hasta la finalización de la subasta.

En vez de realizar una puja directamente indicando la cantidad a pagar, el usuario deberá hacerlo en dos ventanas: "commit" y "reveal".

Ventana "commit"

En la la primera etapa, "commit", la idea es que cada usuario realice un pago en concepto de fianza, que deberá ser igual o mayor a la puja que quiere realizar. La función es de esta fianza es evitar que un mismo usuario pueda realizar varias pujas y solo desvelar la que le interese. La desventaja es que los demás usuarios podrán conocer el valor mínimo de la puja, pero no el máximo. Otras alternativas sería establecer un precio fijo de participación, siempre que no sea si es muy pequeño en comparación al del objeto a subastar, para que no sea rentable generar varias pujas con diferentes valores.

En ventana, en vez de incluir el valor de la puja a realizar, incluirán un "hash" del valor de la puja. Para evitar que otros usuarios puedan descifrar este valor mediante fuerza bruta, en el hash se incluye la dirección del usuario y un "salt" a elección del usuario.

En el código de ejemplo el "hash" se calcula mediante la función Keccak-256 de los siguientes argumentos en binario:

- dirección del usuario
- salt a elección del usuario

- valor de la puja a realizar

Este valor se podría calcular mediante una función de Solidity. En este caso es una función *pure* que no genera una transacción a la red, pero el nodo donde se ejecute podrá conocer el valor de los argumentos. Por ello es recomendable ejecutar este código en una máquina local.

file: contracts/commit_reveal/blind_auction.sol

```
12  function calculateHash(address _sender, uint256 _salt, uint256 _value)
external pure returns(bytes32){
13      return keccak256(abi.encodePacked(_sender,_salt,_value));
14  }
```

Código 30: función para calcular el hash de la etapa "commit"

Otra alternativa sería realizar el cálculo en otro idioma de programación. Se propone el siguiente script Python:

file: contracts/commit_reveal/blind_auction_hash.py

```
 9  #ASK for address, salt and value
10  address = input("\nInsert wallet address-> ")
11  salt = int(input("\nInsert salt value-> "))
12  value = int(input("\nInsert value to send-> "))

18  #convert values to bytes
19  b_address = bytes.fromhex(address[2:])
20  b_salt = salt.to_bytes(32, byteorder = "big")
21  b_value = value.to_bytes(32, byteorder = "big")
22
23  #calculate keccak
24  encoded = b_address + b_salt + b_value
25  hash = keccak(encoded)
26
27  #return hash
28  hash_tohex = binascii.hexlify(hash).decode("ascii")
29  print("\nCALCULATED KECCAK HASH:")
30  print(" 0x"+hash_tohex.upper())
```

Código 31: script para calcular el hash de la etapa "commit"

Una vez calculado el hash, el usuario podrá invocar la siguiente función del contrato que almacenará su petición de puja.

file: contracts/commit_reveal/blind_auction.sol

```
62  //commitment = keccak("address"+"(uint256)seed"+"*uint_256)bid_value")
63  //you should send more eth than bid_value for the bid to be valid
64  //if you make an invalid bid, you'll lose it
65  function sendBidCommit(bytes32 _commitment) external payable{
66      checkUpdateAuctionState(AUCTION_COMMIT);
67      require(msg.value > 0);
68
69      users[msg.sender] = Participant(_commitment, msg.value, false, 0);
70
71      //owner can take this commitment lif user doesnt reveal correctly
72      valueUncheckedBids += msg.value;
73  }
```

Código 32: contrato subasta: función enviar commit

La función comprueba que la subasta esté en la ventana de peticiones, y almacena en un *mapping* la fianza y el hash de la puja.

Ventana "Reveal"

Cuando la ventana de peticiones se cierre, se abrirá otra ventana de revelación, "Reveal", donde los usuarios que hayan participado deberán revelar el valor real de la puja así como el "seed" elegido, y el contrato realizará el cálculo para cerciorarse de que es correcto y por tanto la puja válida.

Esta ventana deberá tener un tiempo lo suficientemente alto para que los usuarios puedan revelar a tiempo su puja, ya que si no lo hacen perderán su fianza. Un valor recomendado podría ser 1 día, como el tiempo de bloque actualmente es de ~12segundos, esta ventana debería durar al menos 7200 bloques.

file: contracts/commit_reveal/blind_auction.sol

```
75  //reveal bid
76  //required for the bid to be valid
77  function sendBidReveal(uint _seed, uint _bidValue) external{
78      checkUpdateAuctionState(AUCTION_REVEAL);
79
80      bytes32 _commitment = keccak256(
            abi.encodePacked(msg.sender , _seed, _bidValue));
81
82      Participant storage p = users[msg.sender];
83      require(p.commitment == _commitment, "Incorrect commitment");
84      require(_bidValue < p.commitValue, "Not enought commitment value");
85
86      p.revealed = true;
87      p.bidValue = _bidValue;
88
89      checkUpdateHighestBidder(_bidValue);
90
91      //owner can no longer withdraw this commitment
92      valueUncheckedBids -= p.commitValue;
93  }
```

Código 33: contrato subasta: función "reveal"

En esta función se realiza el cálculo de hash de las tres variables (*address*, *seed* y *bidValue)*, y se compara contra el valor enviado en la ventana de "Commit". Además se comprueba que el valor real de la puja sea menor al valor de la fianza pagada.

Acciones finales tras "Commit-Reveal"

file: contracts/commit_reveal/blind_auction.sol

```
 95  function getRefund() external{
 96      //CHECK
 97      checkUpdateAuctionState(AUCTION_FINISHED);
 98      Participant storage p = users[msg.sender];
 99      require(p.revealed); //user didn't reveal in time
100
101      //EFFECTS
102      uint val = p.commitValue;
103
104      //winner can only withdraw excess of commit value
105      if(msg.sender == highestBidder){
106          val -= p.bidValue;
107      }
108
109      //erase variables to get gas back
110      //same user cant call again this function successfully
111      p.commitment = 0;
112      p.commitValue = 0;
113      p.revealed = false;
114      p.bidValue = 0;
115
116      //INTERACT
117      (bool success, ) = msg.sender.call{value: val}("");
118      require(success);
119  }
```

Código 34: contrato subasta: obtener devolución

Finalizada esta ventana de "reveal", los usuarios que no hayan revelado su puja no podrán hacerlo ya y perderán su fianza. Los usuarios que lo hayan hecho y no sean los ganadores podrán recuperar su fianza, mientras que el ganador de la puja solo podrá recuperar la parte excedente de su fianza.

6.1.5 Bucles y borrado de arrays

Aunque Ethereum soporte una EVM turing completa, es importante tner en cuenta las limitaciones a la hora de crear bucles. Debido a la limitación del gas por bloque, no es posible crear funciones que ejecuten instrucciones en bucle de forma indefinida.

El uso de bucles está desaconsejado debido a esta limitación, sin embargo no siempre es posible evitarlo. Si se crean bucles que dependan directamente de las entradas de los usuarios se corre el riesgo de crear un bucle lo suficientemente grande que agote el gas disponible en el bloque y que no se pueda completar. (Ver ejemplo vulnerabilidad 4.7 Denegación de servicio).

El mismo error puede ser causado cuando se intenta borrar un array dinámico con una de las siguientes expresiones:

```
array = new uint[](0);
delete array;
```

La causa es que el código generado consiste en un bucle que borra elemento a elemento. Lo cual si el array es lo suficientemente largo, el gas necesario para borrarlo será insuficiente.

Una solución es crear funciones que eliminen de forma parcial el array, por ejemplo mediante la instrucción *array.pop(),* que elimina el último elemento.

```
file: contracts/loops_arrays/array_delete_elements.sol

39  //Deletes last _n elements
40  function deleteElements(uint _n) public{
41      require(_n <= dataArray.length);
42
43      for(; _n>0 ; _n--){
44          dataArray.pop();
45      }
46  }
```

Código 35: solución borrado arrays: borrado parcial

Sin embargo una mejor solución es directamente no borrar el array. Se puede utilizar una variable adicional para controlar el número de elementos guardados en el mismo. El array original seguirá conteniendo información pero a ojos del programador, esta información no se accederá porque se controla la longitud del array mediante la variable creada.

File: contracts/loops_arrays/array_use_variable.sol

```
23  uint[] dataArray;
24  uint lenght = 0; //controls lenght of array
25
26  function getData(uint _pos) public returns(uint){
27      require(_pos < lenght);
28
29      return dataArray[_pos];
30  }
31
32  function getLength() public view returns(uint){
33      return lenght;
34  }
35
36  function addData(uint data) public{
37      dataArray.push(data);
38      lenght +=1;
39  }
40
41  function eraseArray() public{
42      lenght = 0;
43  }
```

Código 36: solución borrado arrays: no borrar array

Tener en cuenta que al usar esta solución es más complejo eliminar elementos en el medio del array. Una solución sería mover el último elemento a la posición del elemento a borrar y disminuir *length* en 1.

Además es importante recordar que en la última solución, la variable *dataArray.length* ya no debería usarse y solo usarse la variable *lengh* (en el caso del ejemplo). Lo más recomendable es mover la funcionalidad del array a una librería por separado para controlar las operaciones sobre el array desde un conjunto de funciones cerrado.

6.1.6 Manejo de errores

Solidity utiliza excepciones que revierten el estado del contrato para manejar errores. Cuando se provoca una orden de *revert*, todos los cambios realizados en la llamada actual, incluyendo las subllamadas, serán deshechas, y además se pasará el error al la llamada "padre".

Las excepciones seguirán disparándose siguiendo la pila de llamadas hasta revertir toda la transacción, o encontrar un bloque *try/catch*. El único caso que se sale de esta norma es la función *send()* y las funciones de bajo nivel *call(), delegatecall y staticcall*, ya que devuelven false si encuentran una excepción en vez de generar otra nueva.

Por lo tanto es necesario prestar especial atención en este tipo de llamadas, comprobando siempre el valor de retorno y actuando en consecuencia.

Funciones para generar excepciones

Solidity proporciona dos métodos para causar una excepción. La primera de ellas *revert(),* que causa la excepción directamente, y la segunda *require(bool condition)*, que causará la excepción cuando la condición sea falsa.

Ambos métodos admiten un argumento adicional para indicar una *string* con un mensaje de error. Además se pueden declarar errores personalizados, aunque solo es posible utilizarlos junto a *revert()*.

```
file: contracts/commit_reveal/blind_auction.sol

162  function claimPrice() external{
163      checkUpdateHighestBidder(AUCTION_FINISHED);
164      require(msg.sender == highestBidder);
165      if (priceClaimed) revert AlreadyClaimed();
166
167      priceClaimed = true;
168      //get some imaginary price
169  }
```

Código 37: ejemplo uso revert y errores personalizados

En el ejemplo se utiliza la expresión *require,* y a continuación la expresión *revert* junto a un error personalizado.

Es recomendable utilizar estas expresiones para advertir al usuario de que algún parámetro es incorrecto o ha ocurrido un error de otro tipo, ya que al revertir la operación, no se generará ningún cambio en el estado del contrato. La alternativa sería no realizar ninguna acción pero se proporciona menos información al usuario que puede creer que la operación solicitada se ha completado, y no darse cuenta del error.

La única consideración a tener en cuenta, es evitar situaciones que provoquen un bloqueo del estado del contrato si se produce un error que revierta una operación indispensable para avanzar a otro estado.

Errores personalizados

Los errores personalizados son una buena forma de mostrar un mensaje util de la causa del error al usuario, sin la necesidad de añadir una *string* a cada posible causa, lo cual aumenta considerablemente los costes de despliegue del contrato.

Se pueden declarar fuera o dentro del contrato de la siguiente forma:

```
1  error RazonError();
```

Además se pueden añadir argumentos que especifiquen con más detalle la causa del error:

```
1  // Balance insuficiente para realizar la transferencia
2  // Se requiere "required", pero solo hay disponible "available"
3  error InsufficientBalance(uint256 available, uint256 required);
```

De esta forma cuando se produce un error de este tipo, la EVM devuelve al usuario el *methodID* del error así como los argumentos, y este a través de la ABI podrá descodificar el mensaje de error.

Esta operación la puede realizar una Dapp de forma transparente para el usuario y mostrar un mensaje de error claro sin necesidad de incluir una *string* en el contrato.

En el contrato de de ejemplo BlindAuction se utilizan varios tipos de errores personalizados:

file: contracts/commit_reveal/blind_auction.sol

```
27  /// Error thrown if auction is in a different state
28  /// than required to perform a certain action
29  /// @param required auction state
30  /// @param actual auction state
31  error IncorrectAuctionState( uint8 required, uint8 actual);
32
33  /// Thrown if msg.value ==0
34  error InvalidPayment();
35
36  /// Thrown if bidValue is higher than payment
37  /// made in commmit phase
38  /// @param commitmentValue payment made in commit fase
39  /// @param bidValue user bid value
40  error NotEnoughCommitmentValue(uint commitmentValue, uint bidValue);
41
42  /// Thrown if the commitment doesnt match seed and
43  /// bidValue from user
44  error InvalidCommitment();
45
46  /// Thrown if user didn't reveal in time
47  error NotRevealed();
48
49  /// Thrown if user of admin already claimed their price
50  error AlreadyClaimed();
```

Código 38: contrato BlindAuction, declaración de Errores personalizados

Assert

Solidity proporciona una función más, *assert*, *con* el fin de comprobar condiciones que en ningún caso deberían fallar, salvo que exista un error en el código del contrato. Además de detectar condiciones donde exista un error, también sirve como barrera para evitar que se materialice, ya que se revertirá la operación si se llegase a ejecutar.

Este método genera un error personalizado predeterminado del tipo *Panic(uint),* y también es emitido de forma automática en comprobaciones que realiza Solidity, por ejemplo, las comprobaciones de underflow/overflow.

Un ejemplo del uso de este método puede ser en un contrato del tipo ERC-20, donde el *supply* de tokens sea constante, nunca se debería poder emitir una transferencia que supere el *supply*. Si se dá esta condición, es que existe un error en el contrato. Una vez realizadas las comprobaciones mediante *require* y antes de realizar la transferencia se hace la comprobación *assert*:

```
1   function batchTransfer(address[] _receivers, uint256 _value)
2           public
3           whenNotPaused
4           returns (bool) {
5
6    uint cnt = _receivers.length;
7    uint256 amount = uint256(cnt) * _value;
8    require(cnt > 0 && cnt <= 20);
9    require(_value > 0 && balances[msg.sender] >= amount);
10
11   assert(_value < totalSupply);
12
13   balances[msg.sender] = balances[msg.sender].sub(amount);
14   for (uint i = 0; i < cnt; i++) {
15          balances[_receivers[i]] = balances[_receivers[i]].add(_value);
16          Transfer(msg.sender, _receivers[i], _value);
17   }
18   return true;
19  }
```

Código 39: ejemplo uso "assert()" en contrato vulnerable a "batchOverflow"

En este caso como el método presenta un error de overflow en la línea 7 (ya analizado en el apartado 5.1), las comprobaciones de require no funcionan según lo esperado. Sin embargo se podría haber detectado y protegido de la vulnerabilidad añadiendo la instrucción *assert*.

6.2 Pruebas de *smart contracts*

La fase de pruebas es una de las más importantes a la hora de detectar errores de programación y comportamientos no deseados del código que puedan causar problemas. Se le debe prestar especial atención debido a las características de los *smart contracts:*

- son aplicaciones de gran valor financiero: ya que trabajan directamente con activos financieros. Esto provoca que pequeñas vulnerabilidades puedan dar como resultado grandes pérdidas económicas para los usuarios del contrato.
- Los *smart contracts* son inmutables: Por naturaleza los *smart contracts* son inmutables. En aplicaciones tradicionales los desarrolladores tienen la posibilidad de arreglar errores que se desarrollen tras el lanzamiento, pero en los *smart contracts* esta práctica no suele ser posible.

  Se pueden añadir mecanismos de actualización, por ejemplo a través de un patrón proxy, pero son difíciles de implementar y además pueden reducir valor a la aplicación a implementar. En la mayoría de los casos debería ser considerado como último recurso.

6.2.1 Entorno de pruebas

Existen múltiples entornos que facilitan el desarrollo y las pruebas de un *smart contract*. Algunos entornos son más sencillos para el usuario como *Remix IDE*, que funciona en el navegador, lo que lo hace ideal para aprender el lenguaje de programación *Solidity* y desarrollar contratos sencillos sin instalar un entorno completo de desarrollo. Permite el despliegue en una red virtual local, así como programar un conjunto de pruebas. Es un buen punto de partida para proyectos pequeños aunque para proyectos más complejos es más limitado.

Otras alternativas más completas son *Hardhat*, *Truffle* y *Brownie*. Son entornos mucho más completos y complejos que *Remix*, pero a cambio son más flexibles. Permiten el desarrollo en equipo y posibilitan la integración de herramientas de trabajo diferentes. *Hardhat* y *Truffle* se basan en *JavaScript* como lenguaje de programación de scripts y manejo de la consola para controlar el entorno, mientras que *Brownie* utiliza *Python*.

El desarrollo y ejecución de los test unitarios y de integración pueden requerir mucho tiempo de desarrollo cuando aumenta el tamaño del proyecto. El entorno *Brownie* facilita la tarea del desarrollador de varias maneras. Por ejemplo permite configurar el estado de la *blockchain* para las pruebas y así realizar pruebas en diversas condiciones. También integra la librería *Hypothesis* destinada a escribir pruebas con las técnicas de "prueba de propiedades" y "prueba de estados".

Estas dos técnicas disminuyen el tiempo de codificar pruebas y aumentan el rango de entradas a probar, lo que disminuye el tiempo de desarrollo y aumenta las posibilidades de detectar vulnerabilidades.

6.2.2 Pruebas unitarias y de integración

Las pruebas unitarias y de integración son un tipo de prueba automatizada que permite comprobar la exactitud del código y su correcto funcionamiento:

- Las pruebas unitarias se encargan de verificar de forma individual un comportamiento o componente del código. Estas pruebas deben ser de rápida ejecución y describir de forma sencilla el error si fallan
- Las pruebas de integración se encargan de comprobar que los diferentes componente interactúan de forma correcta entre ellas. Por ejemplo entre diferentes *smart contracts* con herencia, o también la integración de una DApp con un *smart contract*.

6.2.3 Ejemplo prueba unitaria con Brownie

Se ha realizado un conjunto de pruebas unitarias para las funciones del contrato SendEther en el entorno Brownie[35]. En este caso se trata de un contrato sencillo que no tiene estado, lo que simplifica en gran medida el desarrollo de los test.

En un contrato corriente con varias posibilidades de estado, sería conveniente probar cada función en todos los estados posibles del contrato.

Gracias a la incorporación del módulo "pytest" *[36]*, se pueden añadir "fixtures". Las "fixtures" son funciones que se ejecutan antes de cada test y permiten configurar un estado del contrato. De esta forma se evita duplicidad del código. En el ejemplo se han utilizado para hacer un despliegue de los contratos "ExpensiveWallet", "CheapWallet", así como del contrato a probar "SendEther".

```
1  ## DEPLOY CONTRACTS
2  # SendEther, ExpensiveWallet, CheapWallet
3  @pytest.fixture(scope="module")
4  def send_ether(SendEther, accounts):
5      return accounts[0].deploy(SendEther)
6
7
8  @pytest.fixture(scope="module")
9  def expensive_wallet(ExpensiveWallet, accounts):
10     return accounts[0].deploy(ExpensiveWallet)
11
12
13 @pytest.fixture(scope="module")
14 def cheap_wallet(CheapWallet, accounts):
15     return accounts[0].deploy(CheapWallet)
```

Código 40: declaración fixtures

Además también existen directivas que revierten el estado de la *blockchain* de pruebas tras la ejecución de cada test. que configuren un estado del contrato, y que permitan volver a este estado tras ejecutar cada test.

```
1  ## ISOLATE STATE OF EACH TEST
2  # SendEther, ExpensiveWallet, CheapWallet
3  @pytest.fixture(autouse="true")
4  def isolate_functions(fn_isolation):
5      pass
```

Código 41: aislamiento de tests

Todas las funciones que contengan "test" serán ejecutadas dentro de cada script de Python que se encuentre en la carpeta "tests", al lanzar el comando

```
$ brownie test
```

Brownie ofrece otro módulo para facilitar generación de entradas de los usuarios para las diferentes funciones, "hypothesis"[37]. Mediante "hypothesis" se pueden realizar "test basados en modelos" que genera entradas aleatorias dentro de un rango marcado.

También permite un modelo más complejo llamado "test con estado" (*stateful testing* en inglés), donde se define el contrato como una máquina de estados así como un conjunto de instrucciones primitivas, e hypothesis se encarga de definir los test de forma automática.

En el ejemplo se han incluido tan solo "test basados en modelos", que genera entradas dentro de un rango de forma aleatoria. Además si encuentra un conjunto de entradas que hace fallar un test, guardará este caso para repetirlo más adelante una vez corregido el código.

En este caso se generan de forma automática varios test variando las siguientes entradas:

- la cantidad de Ether a enviar
- el tipo de *smart contract* destinatario:
  - que necesite más de 2300 gas para recibir ether (expensive_wallet)
  - que necesite menos de 2300 de gas (cheap_wallet)

file: tests/send_ether/send_ether_send_test.py

```
69  ##test send to contract
70  ## send to wallet that uses > 2300 gas should fail (expensive_wallet)
71  ## send to wallet that uses < 2300 gas should be ok (cheap_wallet)
72  @given(
73      amount=strategy('uint256', max_value=1000**18),
74      sender=strategy('address'),
75      use_expensive_contract_wallet = strategy('bool')
76  )
77  def test_send_to_wallet(send_ether, amount, sender,
use_expensive_contract_wallet, expensive_wallet, cheap_wallet):

78      sender_balance = sender.balance()
79      receiver =  expensive_wallet if use_expensive_contract_wallet else
cheap_wallet
80      receiver_balance = receiver.balance()
81
82      #check if enough balance in sender
83      if amount > sender_balance:
84          return
85
86      if use_expensive_contract_wallet:
87          with reverts(): #expensive wallet reverts
88              send_ether.sendViaSend(receiver.address,
                          {'from': sender, 'amount': amount})
89
90          assert sender_balance == sender.balance()
91
92      else:
93          send_ether.sendViaSend(receiver.address,
                          {'from': sender, 'amount': amount})
94          #check updated balance
95          #important! assuming gas_price=0
96          assert sender_balance - amount == sender.balance()
97          assert receiver_balance + amount == receiver.balance()
```

Código 42: ejemplo de un test unitario

6.2.4 Red de pruebas públicas

En la realización de los test mediante la suite Brownie, estos se despliegan temporalmente en una red de pruebas local. Sin embargo es muy recomendable realizar los test en las redes de pruebas públicas existentes. Ethereum posee varias redes de pruebas que permiten desplegar un *smart contract* e interactuar con el de forma similar al entorno real[36].

Actualmente *Sepolia* es la red recomendada para realizar pruebas con *smart contracts* y aplicaciones. Tiene como limitación la cantidad de "Ether de prueba" disponible por cuenta, que suelen estar entorno al 0.05Eth.

Se pueden programar script de pruebas de forma similar a la anterior con la salvedad de que no es posible revertir el estado de la *blockchain* al ser ahora una red pública. Aunque si el *smart contract* se utiliza junto a una

36 https://ethereum.org/en/developers/docs/networks/

DApp, es una de las mejores formas de realizar un test de integración de ambos antes de realizar el lanzamiento a la red oficial.

6.3 Herramientas de prueba automatizadas

Las pruebas unitarias y de integración se pueden complementar con herramientas de análisis estático de código. La ventaja principal es que requieren mucho menos trabajo para ejecutarlos y pueden dar advertencias en etapas tempranas de la programación del *smart contract,* facilitando la incorporación de soluciones.

Estas herramientas detectan muchas de las vulnerabilidades y antipatrones específicos de los *smart contracts* que se han analizado en este trabajo. En la Tabla2 mostrada a continuación se recogen que vulnerabilidades son capaces de detectar algunas de las herramientas más populares de acceso libre. Tener en cuenta que también existen soluciones de uso corporativo.

Herramienta	Errores Aritméticos	Reentrada	Delegatecall a contratos inseguros	tx.origin en vez de msg.sender	Comprobación errores	Frontrunning o dependencia orden	Mala aleatoriedad	Dependencia de tiempo	Coste gas excesivo (linc. oops y arrays)
Remix		Si		Si	Si		Si	Si	Si
Mythril	Si	Si				Si	Si	Si	
Slither	Si	Si	Si	Si	Si		Si	Si	
Oyente		Si			Si	Si		Si	Si

Tabla 4: Herramientas de análisis estático

6.3.1 Remix

La plataforma Remix también incluye entre sus extensiones predeterminadas un analizador de código estático. La ventaja principal es que recoge muchas de las vulnerabilidades más importantes y no se requiere instalar ningún programa adicional para ejecutarlo.[38]

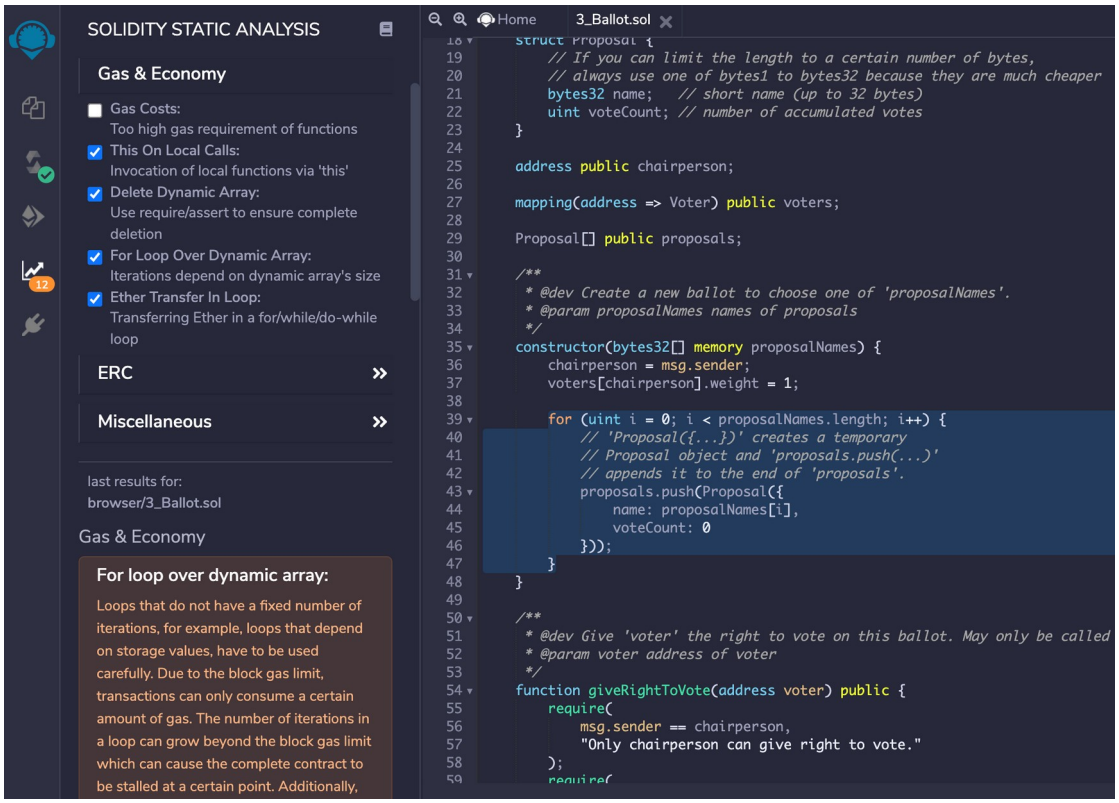

Figura 12: Solidity Static Analysis pluggin en Remix [38]

6.3.2 Mythril

Mythril[39], es una herramienta de código libre mantenida por la empresa ConsenSys. Realiza un análisis simbólico del *bytecode* de un contrato, analizando en el proceso diferentes vulnerabilidades con los módulos incluidos.

La mayor ventaja respecto a un análisis estático tradicional es que esta herramienta puede llegar a explorar muchos sino todos los estados del contrato, detectando situaciones manejadas incorrectamente por el contrato. Como desventaja es el tiempo de ejecución necesario para explorar todos los estados con diferentes parámetros de entrada, es por ello que se le puede especificar un tiempo de ejecución máximo.

En la Figura 13, se muestra el resultado del escaneo del contrato de ejemplo BlindAuction, presentado en el capítulo 5.1.4 Solucción a front-running, patrón "Commit-Reveal". La herramienta muestra dos advertencias de severidad baja, advirtiendo ambos del posible uso incorrecto de la variable *block.number*.

En este caso la variable se usa de forma destinada, pero es util mostrar estas advertencias al desarrollador para evitar vulnerabilidades si no fuese el caso.

```
                :~/tmp$ myth analyze                                    'smart-contract-security/cont
racts/commit_reveal/blind_auction.sol
==== Dependence on predictable environment variable ====
SWC ID: 120
Severity: Low
Contract: BlindAution
Function name: getRefund()
PC address: 2096
Estimated Gas Usage: 1943 - 2368
A control flow decision is made based on The block.number environment variable.
The block.number environment variable is used to determine a control flow decision. Note that the
values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be
 manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks.
 Don't use any of those environment variables as sources of randomness and be aware that use of th
ese variables introduces a certain level of trust into miners.
```

Figura 13: Resultado escaneo mediante Mythril

6.3.3 Slither

Slither es una herramienta mantenida por la empresa Trail of Bits, de análisis estático[40], [41]. Es capaz de identificar varias vulnerabilidades así como de mostrar consejos que ayudan para mejorar la seguridad del código.

Una de las mayores ventajas es la rapidez de ejecución así como la capacidad el lugar del código donde se producen las vulnerabilidades o diferentes advertencias.

En la Figura 14 se realiza un escaneo del contrato BlindAuction, en el cual se muestran varias advertencias no severas pero que son importantes a tener en cuenta. Por ejemplo una de ellas es la recomendación de utilizar una versión más estable y reciente del compilador Solidity. También advierte de las llamadas de bajo nivel mediante "call", para advertir de su peligro en caso de un manejo incorrecto de las mismas.

```
$ slither commit_reveal/blind_auction.sol

BlindAution.claimPrice() (commit_reveal/blind_auction.sol#138-145) compares to a boolean constant:
        -require(bool)(priceClaimed == false) (commit_reveal/blind_auction.sol#141)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#boolean-equality

Different versions of Solidity are used:
        - Version used: ['>=0.7.0<0.9.0', '>=0.8.7<0.9.0']
        - >=0.7.0<0.9.0 (commit_reveal/blind_auction.sol#21)
        - >=0.8.7<0.9.0 (utils/owned.sol#19)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used

Pragma version>=0.7.0<0.9.0 (commit_reveal/blind_auction.sol#21) is too complex
Pragma version>=0.8.7<0.9.0 (utils/owned.sol#19) is too complex
solc-0.8.7 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity

Low level call in BlindAution.getRefund() (commit_reveal/blind_auction.sol#111-135):
        - (success) = msg.sender.call{value: val}() (commit_reveal/blind_auction.sol#133)
Low level call in BlindAution.withdrawWinnerBid() (commit_reveal/blind_auction.sol#147-155):
        - (success) = owner.call{value: val}() (commit_reveal/blind_auction.sol#153)
Low level call in BlindAution.withdrawUncheckedBids() (commit_reveal/blind_auction.sol#157-164):
        - (success) = owner.call{value: val}() (commit_reveal/blind_auction.sol#162)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls

Parameter BlindAuctionHelper.calculateHash(address,uint256,uint256)._sender (commit_reveal/blind_auction.sol#30) is not in mixedCase
Parameter BlindAuctionHelper.calculateHash(address,uint256,uint256)._salt (commit_reveal/blind_auction.sol#30) is not in mixedCase
Parameter BlindAuctionHelper.calculateHash(address,uint256,uint256)._value (commit_reveal/blind_auction.sol#30) is not in mixedCase
Parameter BlindAuctionHelper.checkHash(bytes32,address,uint256,uint256)._hash (commit_reveal/blind_auction.sol#34) is not in mixedCase
Parameter BlindAuctionHelper.checkHash(bytes32,address,uint256,uint256)._sender (commit_reveal/blind_auction.sol#34) is not in mixedCase
Parameter BlindAuctionHelper.checkHash(bytes32,address,uint256,uint256)._salt (commit_reveal/blind_auction.sol#34) is not in mixedCase
Parameter BlindAuctionHelper.checkHash(bytes32,address,uint256,uint256)._value (commit_reveal/blind_auction.sol#34) is not in mixedCase
Parameter BlindAution.sendBidCommit(bytes32)._commitment (commit_reveal/blind_auction.sol#81) is not in mixedCase
Parameter BlindAution.sendBidReveal(uint256,uint256)._seed (commit_reveal/blind_auction.sol#93) is not in mixedCase
Parameter BlindAution.sendBidReveal(uint256,uint256)._bidValue (commit_reveal/blind_auction.sol#93) is not in mixedCase
Parameter BlindAution.checkUpdateHighestBidder(uint256)._bidValue (commit_reveal/blind_auction.sol#166) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
commit_reveal/blind_auction.sol analyzed (3 contracts with 81 detectors), 19 result(s) found
```

Figura 14: resultado escaneo BlindAuction mediante Slither

6.3.4 Oyente

Una de las primeras herramientas para *smart contracts* existentes. Recoge las vulnerabilidades más importantes y no necesita el código fuente para el análisis, aunque debido que no ha sido actualizado recientemente, la instalación puede ser compleja al depender de paquetes antiguos [42], [43]

Oyente extrae el grafo de control desde el *bytecode* EVM del contrato, y lo ejecuta simbólicamente para detectar patrones de vulnerabilidades.

7. Conclusiones y trabajo futuro

Tras un análisis extenso del estado de la seguridad de los *smart contracts* realizado en la memoria podemos repasar los objetivos marcados en la introducción para comprobar si se han cumplido los mismos.

A pesar de explorar las plataformas disponibles que soportan *smart contracts*, una de las primeras necesidades durante la elaboración del trabajo fue la de centrar el foco de atención en una sola plataforma, Ethereum, debido al tiempo disponible y con el fin de profundizar en los detalles técnicos.

En el apartado de vulnerabilidades, se ha detectado una mejoría considerable en el énfasis realizado en la seguridad de los *smart contracts*. Muchas de las vulnerabilidades analizadas son obsoletas en la actualidad gracias a cambios implementados en el compilador y en el diseño de las diferentes plataformas de desarrollo.

Muchos de estos cambios fueron causa de errores ocurridos en el pasado, y a medida que avanza el tiempo se han planteado soluciones para solucionar la mayoría de ellos.

En el último apartado se recogen un gran número de consejos y ejemplos para mejorar la seguridad durante el desarrollo de los *smart contracts,* cumpliendo así el objetivo principal marcado en la introducción de este trabajo.

Se ha descartado la idea de realizar una aplicación completa (*DApp*) que recogiese las buenas prácticas desarrolladas en este trabajo, en favor de ejemplos de contratos más reducidos pero que solucionan problemas más concretos. De esta forma serán de mejor comprensión para el lector que intentar conocer el funcionamiento de un *smart contract* compuesto por varios módulos y de una lógica compleja. Además el desarrollo de esta aplicación conllevaría una carga de trabajo importante, que mermaría el tiempo disponible para el resto de secciones, no menos importantes.

La planificación inicial fue bastante satisfactoria quitando pequeños cambios, como descartar la idea de la *DApp* y otros contratiempos como pequeños retrasos en los objetivos de las entregas parciales, que fueron solventados correctamente en la última entrega final. No se han detectado impactos negativos reconocidos en el apartado 1.6.

Personalmente, la realización de este trabajo me ha aportado una gran cantidad de conocimientos en la materia así como conceptos y prácticas que desconocía hasta entonces y definitivamente me han convertido en mejor desarrollador y más proactivo en la seguridad no solo en los *smart contracts*. Espero que este trabajo le sea tan útil a los futuros lectores del mismo, como lo ha sido para mi realizarlo.

7.1 Trabajo futuro

Debido a la necesidad de centrar el foco de atención en una sola plataforma, una de las primeras ideas para extender este trabajo es explorar el estado de la seguridad en el resto de plataformas.

Muchas de las vulnerabilidades analizadas también son aplicables a otras plataformas descentralizadas basadas en la *blockchain*. Sin embargo muchas plataformas utilizan una máquina virtual diferente a Ethereum y por tanto otros lenguajes de programación que pueden presentar diferentes retos y vulnerabilidades de otra naturaleza a las exploradas en este trabajo.

Además considero que se debe hacer un esfuerzo adicional para advertir y enseñar a los desarrolladores de los detalles que deben tener en cuenta a la hora de desarrollar código seguro mediante la incorporación de advertencias y errores que adviertan al desarrollador. Ya que debido a la gran cantidad de factores a tener en cuenta, pueden sentirse abrumados y pasar detalles importantes por alto. Se podría analizar la forma de mejorar la integración de las diferentes herramientas en una sola.

Anexo I: repositorio de código

El trabajo actual contiene múltiples códigos fuente de ejemplos así como scripts para probar diferentes conceptos mencionados en este documento. Estos archivos se encuentran en la carpeta "smart-contract-security", también accesibles mediante un repositorio de código en GitHub en el siguiente enlace:

- https://github.com/roirh/smart-contract-security

Bibliografía

[1] T. Claburn, «Smart contract developers "not really" focused on security». https://www.theregister.com/2022/04/26/smart_contract_losses/ (accedido 9 de enero de 2023).

[2] A. López Vivar, A. T. Castedo, A. L. Sandoval Orozco, y L. J. García Villalba, «An Analysis of Smart Contracts Security Threats Alongside Existing Solutions», *Entropy*, vol. 22, n.º 2, 2020, doi: 10.3390/e22020203.

[3] H. Zhou, A. Milani Fard, y A. Makanju, «The State of Ethereum Smart Contracts Security: Vulnerabilities, Countermeasures, and Tool Support», *J. Cybersecurity Priv.*, vol. 2, n.º 2, pp. 358-378, 2022, doi: 10.3390/jcp2020019.

[4] «Clean Contracts - a guide on smart contract patterns & practices · useWeb3.xyz», *useWeb3.xyz*. https://www.useweb3.xyz/guides/clean-contracts (accedido 11 de octubre de 2022).

[5] «Best Practices for Smart Contract Development – Yos Riady · Software Craftsman». https://yos.io/2019/11/10/smart-contract-development-best-practices/#understand-security-vulnerabilities (accedido 11 de octubre de 2022).

[6] R. Ma y K. Honda, *Fundamentals of smart contract security / Richard Ma [and four others]; foreword by Keisuke Honda.* New York, New York: Momentum Press, LLC, 2019.

[7] K. Solorio, R. Kanna, y D. H. Hoover, *Hands-on smart contract development with Solidity and Ethereum : from fundamentals to deployment / Kevin Solorio, Randall Kanna, and David H. Hoover.*, First edition. Sebastopol, California: O'Reilly Media, Inc., 2020.

[8] S. Nakamoto, «Bitcoin: A Peer-to-Peer Electronic Cash System», p. 9.

[9] «Blockchain in the food supply chain - What does the future look like?», *Blockchain in the food supply chain - What does the future look like?* https://one.walmart.com/content/globaltechindia/en_in/Tech-insights/blog/Blockchain-in-the-food-supply-chain.html (accedido 15 de octubre de 2022).

[10] «¿Qué es NFT o NFTs (Token No Fungible)? – Axency». https://www.axency.com/que-es-nfts-token-no-fungible/26/03/2021/ (accedido 26 de octubre de 2022).

[11] J. Lluís de la Rosa Esteva y A. Ballesteros Rodríguez, «Fundamentos de Ethereum», en *Fundamentos de Ethereum*, UOC.

[12] «Litecoin LTC whitepapers - whitepaper.io». https://whitepaper.io/document/683/litecoin-whitepaper (accedido 26 de octubre de 2022).

[13] «What is Monero (XMR)?», *getmonero.org, The Monero Project*. https://www.getmonero.org//get-started/what-is-monero/index.html (accedido 26 de octubre de 2022).

[14] «Ethereum Whitepaper | ethereum.org». https://ethereum.org/en/whitepaper/ (accedido 26 de octubre de 2022).

[15] «The Merge», *ethereum.org*. https://ethereum.org (accedido 26 de octubre de 2022).

[16] G. Cardozo y P. Perdomo, «Comparación de plataformas para smart contracts basadas en blockchain», PROYECTO DE GRADO, UNIVERSIDAD DE LA REPÚBLICA - FACULTAD DE INGENIERÍA, 2020. [En línea]. Disponible en: https://www.colibri.udelar.edu.uy/jspui/bitstream/20.500.12008/24600/1/CP20.pdf

[17] «DASP - TOP 10». https://dasp.co/ (accedido 9 de enero de 2023).

[18] «Overview · Smart Contract Weakness Classification and Test Cases». http://swcregistry.io/ (accedido 9 de enero de 2023).

[19] «Solidity Security: Comprehensive list of known attack vectors and common anti-patterns». https://blog.sigmaprime.io/solidity-security.html (accedido 9 de enero de 2023).

[20] PeckShield, «Integer Overflow (i.e., proxyOverflow Bug) Found in Multiple ERC20 Smart Contracts (CVE-2018–10376)», *Medium*, 2 de mayo de 2018. https://peckshield.medium.com/integer-overflow-i-e-proxyoverflow-bug-found-in-multiple-erc20-smart-contracts-14fecfba2759 (accedido 5 de enero de 2023).

[21] PeckShield, «New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018–10299)», *Medium*, 2 de mayo de 2018. https://peckshield.medium.com/alert-new-batchoverflow-bug-in-multiple-erc20-smart-contracts-cve-2018-10299-511067db6536 (accedido 5 de enero de 2023).

[22] S. Tual, «No DAO funds at risk following the Ethereum smart contract 'recursive call' bug discovery», *Stephan Tual's Blog*, 13 de junio de 2016. https://medium.com/ursium-blog/no-dao-funds-at-risk-following-the-ethereum-smart-contract-recursive-call-bug-discovery-29f482d348b (accedido 11 de diciembre de 2022).

[23] «Genesis», *Ethereum Classic*. https://ethereumclassic.org/why-classic/genesis (accedido 5 de enero de 2023).

[24] «How to split the DAO: Step-by-Step - The DAO - Confluence». https://daowiki.atlassian.net/wiki/spaces/DAO/pages/3833911/How+to+split+the+DAO+Step-by-Step (accedido 11 de diciembre de 2022).

[25] «Deconstructing theDAO Attack: A Brief Code Tour», *archive.vn*, 27 de febrero de 2018. https://archive.vn/apdnd (accedido 11 de diciembre de 2022).

[26] «Analysis of the DAO exploit», *Hacking Distributed*. https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/ (accedido 11 de diciembre de 2022).

[27] «Smart Contract Attacks [Part 2] - Ponzi Games Gone Wrong | HackerNoon». https://hackernoon.com/smart-contract-attacks-part-2-ponzi-games-gone-wrong-d5a8b1a98dd8 (accedido 11 de diciembre de 2022).

[28] G. Wood, «walletLibrary.sol». parity-contracts, 3 de octubre de 2022. Accedido: 11 de diciembre de 2022. [En línea]. Disponible en: https://github.com/parity-contracts/0x863df6bfa4/blob/4a20201558b87d0eb841d368a4912d250b8b744d/contracts/walletLibrary.sol

[29] A. Reutov, «Predicting Random Numbers in Ethereum Smart Contracts», *Medium*, 30 de mayo de 2018. https://blog.positive.com/predicting-random-numbers-in-ethereum-smart-contracts-e5358c6b8620 (accedido 5 de enero de 2023).

[30] P. Bylica, «How to Find $10M Just by Reading the Blockchain», *The Golem Project*, 6 de abril de 2017. https://medium.com/golem-project/how-to-find-10m-by-just-reading-blockchain-6ae9d39fcd95 (accedido 5 de enero de 2023).

[31] T. Be'ery, «The BadgerDAO Hack: What really happened and why it matters», *ZenGo*, 15 de diciembre de 2021. https://zengo.com/the-badgerdao-hack-what-really-happened-and-why-it-matters/ (accedido 5 de enero de 2023).

[32] «BadgerDAO Malicious Script». [ZenGo X], 8 de agosto de 2022. Accedido: 5 de enero de 2023. [En línea]. Disponible en: https://github.com/ZenGo-X/badger_dao_script_analysis

[33] «Ethereum Smart Contract Best Practices». https://consensys.github.io/smart-contract-best-practices/ (accedido 5 de enero de 2023).

[34] «bogatyy/bancor: Code corresponding to my analysis of Bancor front-running». https://github.com/bogatyy/bancor (accedido 5 de enero de 2023).

[35] «Brownie». Brownie, 8 de enero de 2023. Accedido: 8 de enero de 2023. [En línea]. Disponible en: https://github.com/eth-brownie/brownie

[36] «pytest: helps you write better programs — pytest documentation». https://docs.pytest.org/en/7.2.x/ (accedido 8 de enero de 2023).

[37] «Most testing is ineffective - Hypothesis». https://hypothesis.works/ (accedido 8 de enero de 2023).

[38] «Solidity Static Analysis — Remix - Ethereum IDE 1 documentation». https://remix-ide.readthedocs.io/en/latest/static_analysis.html (accedido 5 de enero de 2023).

[39] «Mythril». ConsenSys, 5 de enero de 2023. Accedido: 5 de enero de 2023. [En línea]. Disponible en: https://github.com/ConsenSys/mythril

[40] J. Wu, «Slither – a Solidity static analysis framework», *Trail of Bits Blog*, 19 de octubre de 2018. https://blog.trailofbits.com/2018/10/19/slither-a-solidity-static-analysis-framework/ (accedido 5 de enero de 2023).

[41] «crytic/slither: Static Analyzer for Solidity». https://github.com/crytic/slither (accedido 5 de enero de 2023).

[42] «Oyente». Enzyme Finance, 5 de enero de 2023. Accedido: 5 de enero de 2023. [En línea]. Disponible en: https://github.com/enzymefinance/oyente

[43] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, y A. Hobor, «Making Smart Contracts Smarter», en *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, Vienna Austria, oct. 2016, pp. 254-269. doi: 10.1145/2976749.2978309.