UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Análisis de vulnerabilidades y auditoría de Smart Contracts. Estudio de casos famosos.

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Vicent Oltra, Lluís

Tutor/a: Molina Marco, Antonio

CURSO ACADÉMICO: 2023/2024

RESUMEN

Este TFG tiene como objetivo analizar las vulnerabilidades más comunes en los smart contracts mediante el estudio detallado de casos de ataques famosos y la auditoría exhaustiva de contratos inteligentes utilizando herramientas especializadas. Se presentarán recomendaciones y mejores prácticas basadas en los hallazgos, contribuyendo a la mejora de la seguridad y la fiabilidad de las aplicaciones descentralizadas.

Palabras clave: Blockchain; contratos inteligentes; seguridad; criptomonedas;

ABSTRACT

This project aims to analyze the most common vulnerabilities in smart contracts by conducting a detailed study of famous attack cases and thoroughly auditing smart contracts using specialized tools. Recommendations and best practices based on the findings will be presented, contributing to the improvement of security and reliability of decentralized applications.

Keywords: Blockchain; smart contract; security; cryptocurrency;

ÍNDICE

ÍNDICE DE FIGURAS

GLOSARIO

BNB: Binance Coin (BNB) es la criptomoneda nativa de la plataforma Binance, utilizada para pagar transacciones y participar en lanzamientos de tokens dentro del ecosistema Binance.

dApp: Aplicación descentralizada (dApp) es una aplicación que opera en una red blockchain de manera autónoma, utilizando contratos inteligentes sin necesidad de una autoridad central.

DeFi: Finanzas descentralizadas (DeFi) es un ecosistema de aplicaciones financieras construidas sobre blockchain que permite realizar operaciones como préstamos e intercambios sin intermediarios tradicionales.

EVM: Ethereum Virtual Machine (EVM) es el entorno de ejecución para contratos inteligentes en la blockchain de Ethereum, donde se procesa y ejecuta el código de los contratos de manera descentralizada.

Ether: La criptomoneda nativa de la red Ethereum, utilizada para pagar tarifas de transacción y como medio de intercambio dentro de la plataforma Ethereum.

Exchange: Plataforma digital donde los usuarios pueden comprar, vender e intercambiar criptomonedas, ya sea de forma centralizada o descentralizada.

Gas: Unidad que mide la cantidad de trabajo que requiere una transacción o ejecución de un contrato inteligente en la blockchain de Ethereum. Se paga en Ether para compensar a los mineros por su esfuerzo.

NFT: Token no fungible (NFT) es un tipo de token criptográfico que representa un activo digital único e indivisible, a menudo utilizado para objetos de colección, arte y otros activos digitales.

Nodo: Computadora conectada a una red blockchain que participa en el procesamiento y validación de transacciones, manteniendo una copia del libro mayor distribuido.

Oráculo: Servicio que provee datos externos a los contratos inteligentes, permitiéndoles interactuar con información fuera de la blockchain para ejecutar sus funciones correctamente.

Pool de liquidez: Reserva de fondos en un intercambio descentralizado (DEX) que permite a los usuarios intercambiar tokens de manera eficiente, facilitada por contratos inteligentes.

Stablecoin: Criptomoneda diseñada para mantener un valor estable, generalmente respaldada por activos como monedas fiduciarias como el dólar.

Token: Unidad digital de valor creada dentro de una blockchain que puede representar activos, utilidades o derechos específicos y es utilizada en diversas aplicaciones descentralizadas.

Wallet: Aplicación o dispositivo que permite a los usuarios almacenar y gestionar sus criptomonedas de manera segura, interactuando con diversas blockchains.

Wei: La unidad más pequeña de Ether, equivalente a 10^-18 Ether, utilizada para calcular y pagar las tarifas de gas y otras transacciones en la red Ethereum.

1 Introducción.

La tecnología blockchain ha emergido como una revolución novedosa en el campo de las tecnologías de la información y ha transformado la forma en la que se realizan transacciones seguras y se gestionan activos digitales. Los contratos inteligentes, piezas de código ejecutadas en la blockchain, han permitido automatizar y asegurar acuerdos entre partes sin necesidad de intermediarios. Sin embargo, a medida que aumenta la adopción de estas tecnologías, también crece la preocupación por la seguridad de los contratos inteligentes. Las vulnerabilidades en el código pueden ser explotadas por atacantes, resultando en pérdidas millonarias. Este TFG se centra en estudiar casos famosos de ataques a contratos inteligentes y en la auditoría de estos contratos inteligentes, abordando tanto la identificación como la mitigación de vulnerabilidades críticas para mejorar la seguridad en el ecosistema blockchain.

1.1 Motivación.

La motivación principal de este TFG radica en la creciente incidencia de hackeos y robos en el ámbito blockchain. Tal y como se ha señalado en la introducción, estos ataques resultan en pérdidas sustanciales que afectan tanto a compañías como a los inversores particulares. Según el informe "*Crypto Crime Report 2023"* que lleva a cabo la compañía estadounidense Chainalysis especializada en análisis de datos en la blockchain, el valor total de fondos robado en hackeos relacionados con blockchain ha aumentado drásticamente en los últimos años.

En la siguiente figura 1 se puede ver la evolución de fondos robados y de ataques o hackeos al ecosistema blockchain entre 2016 y 2023.

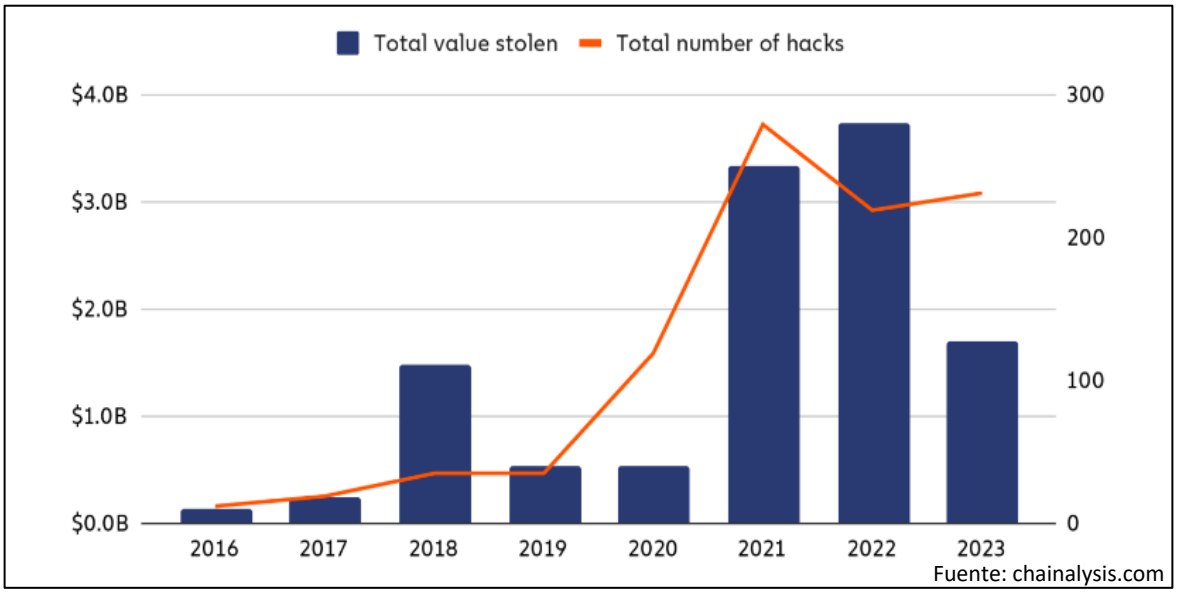


Figura 1: Evolución de fondos robados y hackeos en el ecosistema blockchain

El gráfico muestra una tendencia preocupante; a partir de 2021, el número de hackeos y el valor total robado experimentaron un notable incremento, alcanzando un pico en 2022 con un valor robado de aproximadamente 3,7 mil millones de dólares y casi 200 ataques. Aunque hubo una disminución considerable en 2023, las cifras siguen siendo alarmantemente altas. Es importante resaltar que el volumen de fondos robados sigue el mismo ciclo que el de los precios de las criptomonedas que son robadas en los ataques. Es por ello que se ven máximos locales y globales en los fondos robados en los años 2018 y entre 2021 y 2022.

El creciente número de ataques al ecosistema blockchain se debe a varias razones, entre ellas el rápido crecimiento de la adopción de blockchain en aplicaciones como DeFi y la tokenización de activos, exponiendo a más usuarios y transacciones a riesgos de seguridad. Las pérdidas financieras afectan tanto a usuarios como a la reputación de las plataformas, desincentivando la adopción de nuevas tecnologías. Además, la complejidad y evolución constante del código incrementan la superficie de ataque, requiriendo metodologías avanzadas para identificar vulnerabilidades.

Algunos de los motivos principales por los que se ataca la blockchain incluyen factores humanos, como errores de configuración y negligencia en la gestión de claves privadas, que pueden facilitar el acceso no autorizado de los atacantes. Además, el mal desarrollo de contratos inteligentes, con código vulnerable o mal escrito, abre puertas a explotaciones y hackeos. Asimismo, las estafas de *phishing* y otros métodos de ingeniería social siguen siendo tácticas comunes utilizadas por atacantes para engañar a usuarios y obtener información sensible o fondos, aprovechando la falta de educación y concienciación sobre la seguridad en el ámbito de la blockchain.

Gran parte de los ataques podrían prevenirse si se minimizan las vulnerabilidades en los contratos inteligentes y se realizaran auditorías regulares y exhaustivas. Asegurar que los contratos inteligentes estén libres de fallos críticos es fundamental para proteger tanto a los usuarios como a las plataformas que los implementan. Las auditorías permiten identificar y corregir posibles vulnerabilidades antes de que puedan ser explotadas por atacantes. Este tema será abordado en mayor profundidad a lo largo de este TFG, donde se explorarán diversas técnicas y herramientas para mejorar la seguridad en el desarrollo de contratos inteligentes en el ecosistema blockchain.

Es por estos hechos que la motivación detrás de este trabajo es, por lo tanto, doble: por un lado, fomentar la adopción segura y responsable de las tecnologías blockchain mediante la mejora de las prácticas de desarrollo y, por otro, proteger a los usuarios y sus activos frente a posibles ataques.

1.2 Objetivos.

El objetivo principal de este TFG es identificar y comprender las principales vulnerabilidades en el desarrollo de contratos inteligentes, y aprender a detectarlas y corregirlas mediante la implementación de buenas prácticas de desarrollo y la realización de auditorías utilizando herramientas especializadas.

A continuación se detallan los subobjetivos necesarios para lograr el objetivo principal y que serán los hitos a cumplir en cada uno de los apartados de este TFG:

- Conocer y analizar casos famosos de hackeos a proyectos blockchain basados en vulnerabilidades en smart contracts.
- Examinar y categorizar las principales vulnerabilidades presentes en los contratos inteligentes.
- Comprender la importancia de utilizar herramientas de auditoría de seguridad para analizar contratos inteligentes.
- Proponer un conjunto de mejores prácticas para el desarrollo seguro de contratos inteligentes.
- Incrementar la conciencia sobre la importancia de la seguridad en el desarrollo de contratos inteligentes.

1.3 Estructura.

Una vez descritos los objetivos y las motivaciones que llevan a la realización de este TFG, es importante introducir la estructura genérica que seguirá el documento:

- En el capítulo 2 se desarrolla el marco teórico del TFG y se sientan las bases conceptuales sobre blockchain y smart contracts.
- El capítulo 3 tratará la metodología. Se estructura de forma más concreta el contenido del documento y se describen las herramientas y procesos utilizados para la investigación.
- En el capítulo 4 se analizan incidentes sonados referentes a hackeos a smart contracts para identificar vulnerabilidades y concienciar sobre la importancia del desarrollo seguro.
- En el capítulo 5 se analizan y tratan vulnerabilidades de contratos inteligentes y se profundiza en los tipos de fallos más comunes y sus implicaciones.
- El capítulo 6 trata la auditoría de los contratos inteligentes a través de herramientas especializadas.
- En el capítulo 7 se exponen recomendaciones y mejores prácticas para el desarrollo seguro de los smart contracts.
- Finalmente, en el capítulo 8 de conclusiones se sintetizan los resultados obtenidos y sugieren posibles líneas de investigación futura.

2 Marco teórico.

El marco teórico de este TFG se enfoca en proporcionar una base sólida sobre los conceptos fundamentales y las tecnologías subyacentes a los smart contracts. Primero, se describirán los principios básicos de blockchain, la tecnología que sustenta los smart contracts, incluyendo su estructura, características y mecanismos de consenso. Luego, se introducirá el concepto de smart contract, su funcionamiento, y cómo se implementan en la blockchain. Finalmente, se discutirá la importancia de la seguridad en los smart contracts. Este marco teórico servirá como base para comprender los casos de estudio y los análisis de vulnerabilidades presentados en los capítulos siguientes.

2.1 Blockchain.

Blockchain es una tecnología de sistemas distribuidos que permite mantener una lista creciente de registros agrupados en bloques, de manera segura, transparente e inmutable. Cada bloque contiene un conjunto de registros o transacciones y está vinculado criptográficamente al bloque anterior, formando una cadena. Esta estructura garantiza que los datos registrados en la blockchain sean resistentes a modificaciones y fraudes.

La idea detrás de la tecnología blockchain fue descrita por primera vez en 1991, cuando los investigadores *Stuart Haber* y *W. Scott Stornetta* presentaron una solución computacional para sellar documentos digitales con marcas de tiempo, impidiendo así su modificación o manipulación. El sistema empleaba una cadena de bloques con seguridad criptográfica para almacenar los documentos con marcas de tiempo. Sin embargo, esta tecnología no se aplicó ampliamente y la patente expiró en 2004.

Cuatro años más tarde, en 2008, el concepto de blockchain tal y como lo conocemos actualmente, fue introducido por el autor desconocido *Satoshi Nakamoto* como base de la divisa digital Bitcoin. Desde ese momento, la tecnología blockchain ha ganado gran popularidad y se considera una de las tecnologías más prometedoras y populares dentro del mundo de las tecnologías de la información y la criptografía (Roopika, 2020).

La tecnología blockchain no se trata únicamente de una cadena de documentos enlazados. Para que una de estas cadenas sea considerada una blockchain, debe cumplir ciertas características y tener ciertas propiedades:

- Descentralización: En una blockchain, no existe una autoridad central que controle la red. En su lugar, la red es mantenida por un grupo de nodos distribuidos que trabajan juntos para validar y registrar transacciones. Cada nodo en la red tiene una copia completa del historial de transacciones de la blockchain, lo que elimina la dependencia de un punto único de fallo.

- Inmutabilidad: Una vez que una transacción se registra en un bloque y se añade a la cadena, no puede ser alterada ni eliminada. Esta propiedad se debe a la estructura criptográfica de la blockchain, donde cada bloque contiene un hash del bloque anterior, creando una cadena de registros inalterables. La inmutabilidad garantiza la integridad y veracidad de los datos almacenados en la blockchain.
- Transparencia: Las transacciones en una blockchain son visibles para todos los participantes de la red. Esto proporciona un alto nivel de transparencia, ya que cualquier persona puede verificar y auditar las transacciones. Aunque las transacciones son transparentes, la identidad de los participantes puede mantenerse anónima mediante el uso de claves criptográficas.
- Seguridad: La seguridad de una blockchain se basa en técnicas criptográficas avanzadas. Cada transacción está firmada digitalmente, lo que asegura que solo el propietario de una clave privada específica puede realizar una transacción. Además, el mecanismo de consenso utilizado por la red y los nodos garantiza que los bloques añadidos a la cadena sean válidos y acordados por la mayoría de los nodos de la red.

Para garantizar la descentralización y seguridad de la blockchain se hace uso de mecanismos de consenso. Estos mecanismos son métodos utilizados por los nodos de una blockchain para ponerse de acuerdo sobre el estado de la red. Los dos mecanismos de consenso más conocidos son la prueba de trabajo (Proof of Work, PoW) y la prueba de participación (Proof of Stake, PoS).

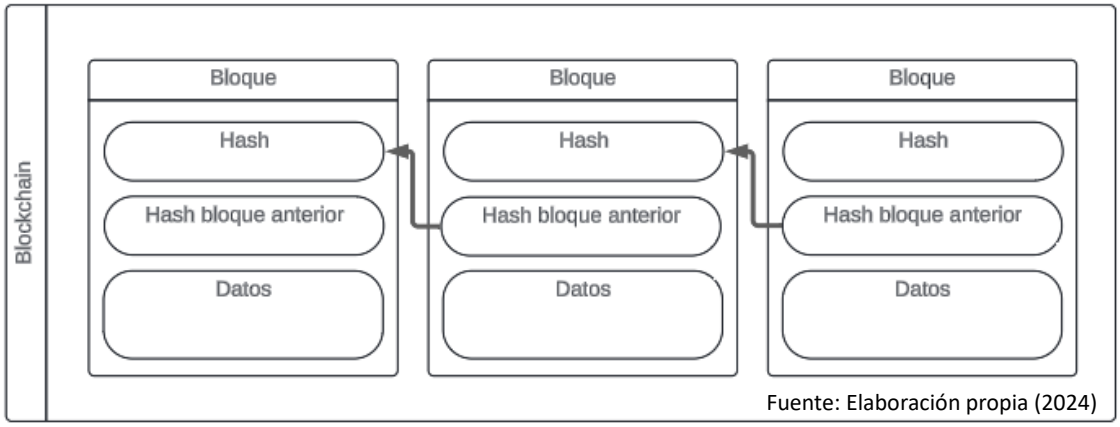
- Prueba de Trabajo (PoW): Introducida por primera vez en la red de Bitcoin, la PoW requiere que unos sistemas de computación llamados mineros resuelvan complejos problemas matemáticos para validar transacciones y crear nuevos bloques. Este proceso consume una gran cantidad de energía computacional.
- Prueba de Participación (PoS): En PoS, los nodos validadores son seleccionados para crear nuevos bloques y validar transacciones en función de la cantidad de criptomoneda que poseen y están dispuestos a depositar como garantía. PoS es más eficiente energéticamente que PoW y se utiliza en varias blockchains modernas, como Ethereum 2.0.

Aunque tanto PoS como PoW son similares en cuanto a que ambos requieren consenso para mantener la seguridad de la blockchain, estos mecanismos tienen funcionamientos e implicaciones muy distintas. La principal diferencia entre estos mecanismos es que PoS necesita significativamente menos trabajo (es decir, energía) para validar los bloques en la blockchain, a diferencia del mecanismo PoW, donde se precisa de una potente capacidad de cómputo para minar los bloques. En PoS el sistema de selección se basa en un sorteo dentro de la blockchain, donde generalmente se elige a los validadores potenciales en función de la cantidad de participación (stake) que tienen (Lin, 2023).

Una vez definidos los elementos y propiedades de la blockchain, se procede a explicar su funcionamiento. De forma básica, una blockchain empieza a funcionar cuando se crea un nuevo registro o transacción. Esta se transmite a todos los nodos de la red y a continuación, estos nodos verifican la transacción siguiendo unas reglas que varían en cada blockchain o protocolo. Una vez verificada, la transacción se agrupa con otras en un nuevo bloque. Para añadir este bloque a la cadena, los nodos deben alcanzar un consenso mediante los mecanismos descritos anteriormente. Una vez alcanzado el consenso, el nuevo bloque se añade a la cadena y se distribuye a todos los nodos, manteniendo la integridad y seguridad de la blockchain.

Para dar un valor único a cada bloque, se hace uso del *hashing*. Un hash es un valor alfanumérico único generado a partir de los datos del bloque mediante una función hash criptográfica. Este proceso convierte una entrada de datos de tamaño arbitrario en una salida de tamaño fijo (Simões, 2022).

En la siguiente figura 2 se muestra un diagrama de una blockchain con sus respectivos bloques y su contenido.



Fuente: Elaboración propia (2024)

Figura 2: Diagrama de una blockchain

Esta figura 2 se trata de una simplificación que sirve para entender de forma visual la estructura de una blockchain. Actualmente, los bloques cuentan con un mayor número de cabeceras como contadores de transacciones o marcas de tiempo *(timestamps)*.

En cuanto a las aplicaciones de esta tecnología, la blockchain está en constante crecimiento y su aplicación se está extendiendo a diversos sectores, incluyendo finanzas, logística, salud, votación electrónica y gestión de identidades, transformando la manera en que se realizan y se aseguran las transacciones y procesos digitales en todo el mundo. Algunas de sus aplicaciones más extendidas y conocidas son:

- Criptomonedas: Las criptomonedas son la aplicación más conocida de blockchain. Bitcoin, Ethereum y otras criptomonedas utilizan blockchain para registrar y verificar transacciones dinerarias de manera segura y descentralizada.

- Supply Chain Management: Blockchain se utiliza para mejorar la transparencia y trazabilidad en las cadenas de suministro. Las empresas pueden registrar cada paso de la producción y distribución en la blockchain, asegurando la autenticidad y calidad de los productos.
- Contratos Inteligentes y Finanzas Descentralizadas (DeFi): Los smart contracts permiten la automatización de acuerdos y transacciones sin intermediarios. Las plataformas DeFi utilizan blockchain para ofrecer servicios financieros como préstamos, intercambios y seguros de manera descentralizada.
- Identidad Digital**:** Blockchain puede utilizarse para gestionar identidades digitales de manera segura y descentralizada, facilitando la verificación y protección de la identidad de los usuarios.

2.2 Smart Contracts.

Un smart contract es un contrato digital cuya ejecución se realiza de forma automática una vez que se cumplen unas condiciones previamente establecidas. Este tipo de contrato se diferencia de los contratos tradicionales en su capacidad para ejecutarse de manera autónoma y sin intermediación, utilizando la infraestructura de la blockchain para garantizar la integridad y seguridad de las transacciones (Levi, Lipton, 2018).

Los smart contracts están compuestos por varios elementos clave que permiten su funcionamiento:

- Código Informático**:** Escrito en lenguajes de programación específicos que define las reglas y condiciones del contrato.
- Condiciones y Lógica de Negocio: Establecen las reglas bajo las cuales se ejecutará el contrato, incluyendo cualquier condición que deba cumplirse para que se ejecute una acción.
- Dirección del Contrato: Una dirección única en la blockchain que identifica el contrato para poder interactuar con él.
- Eventos y Funciones: Los contratos inteligentes pueden emitir eventos cuando ciertas condiciones se cumplen y tienen funciones que pueden ser llamadas por otros contratos o por usuarios.

Para poder explicar el funcionamiento de los smart contracts, en la figura 3 se muestra un esquema del funcionamiento de los mismos.

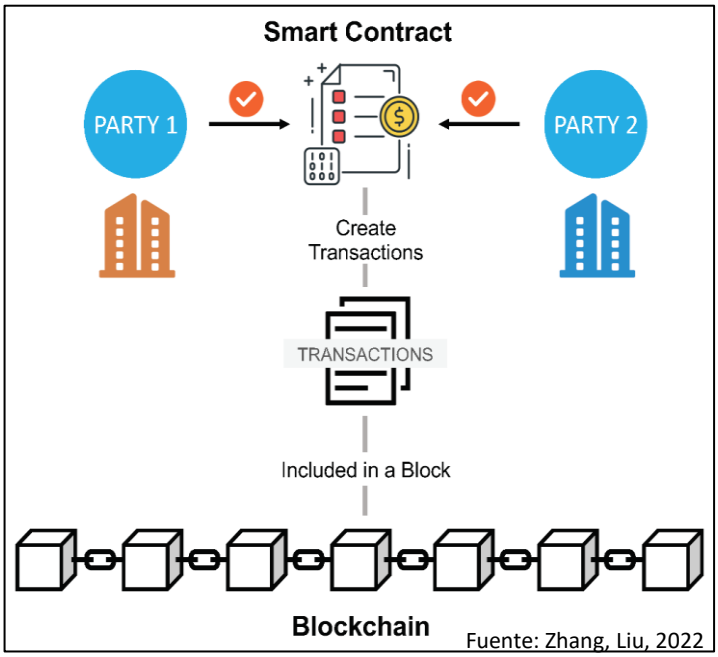


Figura 3: Esquema de funcionamiento de los smart contracts

A partir de esta figura 3 se procede a explicar cómo funcionan los smart contracts:

- Creación del Contrato: Un desarrollador o parte interesada escribe el contrato en un lenguaje de programación compatible con la blockchain en la que se va a desplegar. El contrato contiene todas las condiciones y reglas necesarias para su ejecución.
- Despliegue en la blockchain: Una vez escrito, el contrato se despliega en la blockchain, obteniendo una dirección única que permite a los usuarios interactuar con él. Este despliegue implica el pago de una tarifa en la criptomoneda nativa de la blockchain.
- Interacción con el Contrato: Los usuarios pueden interactuar con el contrato enviando transacciones a su dirección. Estas transacciones pueden activar funciones del contrato que ejecutarán las condiciones especificadas.
- Ejecución Automática: Cuando se cumplen las condiciones predefinidas en el contrato, este se ejecuta automáticamente. La ejecución es inmutable y se registra en la blockchain, garantizando transparencia y seguridad.

Los smart contracts ofrecen varias ventajas significativas, incluyendo la automatización, que elimina la necesidad de intermediarios, reduciendo costes y tiempos de ejecución. Además, proporcionan transparencia, ya que todas las partes pueden ver el código del contrato y las transacciones registradas en la blockchain, lo que aumenta la confianza. La seguridad es otra ventaja clave, ya que la naturaleza inmutable de la blockchain asegura que los contratos no pueden ser alterados una vez desplegados. Por último, el

hecho de que estos contratos estén codificados y ejecutados por computadores hace que se eliminen errores humanos en la ejecución de los mismos.

A pesar de sus ventajas, los smart contracts también enfrentan varios desafíos y limitaciones. La complejidad del código requiere que los contratos sean escritos con precisión, ya que cualquier error puede resultar en comportamientos inesperados. La inmutabilidad del código, una vez desplegado, puede ser problemática si se encuentran errores o si cambian las condiciones legales. Además, la escalabilidad es una limitación, ya que la ejecución de contratos inteligentes puede ser costosa y lenta en blockchains que no están diseñadas para alta capacidad. Por último, la naturaleza descentralizada y autónoma de los smart contracts plantea desafíos regulatorios y legales sobre los mismos.

2.3 Principales blockchains para el desarrollo de smart contracts.

Existen varias blockchains y herramientas diseñadas para el desarrollo y despliegue de smart contracts. Cada una de estas ofrece características y capacidades únicas que las hacen adecuadas para diferentes casos de uso y requisitos de cada proyecto. A continuación, se definen las blockchains más importantes y que tienen más relación con los casos que se estudiarán en apartados posteriores.

- Ethereum:

Ethereum es la plataforma más conocida y utilizada para el desarrollo de smart contracts. Desde su creación en 2015, se ha convertido en el estándar de facto para los contratos inteligentes debido a su flexibilidad y robusta comunidad de desarrolladores.

Ethereum utiliza *Solidity*, un lenguaje de programación de alto nivel diseñado específicamente para escribir smart contracts. *Solidity* permite a los desarrolladores definir la lógica del contrato y las condiciones para su ejecución de manera precisa y segura. Además, cuenta con la Máquina Virtual de Ethereum (EVM) que es el entorno de ejecución para los contratos inteligentes en la blockchain de Ethereum. Este entorno garantiza que todos los nodos ejecuten los contratos de manera consistente y segura.

Ethereum cuenta con un amplio ecosistema de herramientas y bibliotecas, como *Truffle, Remix,* y *Hardhat*, que facilitan el desarrollo, prueba y despliegue de smart contracts.

- Hyperledger Fabric:

Hyperledger Fabric es una plataforma de blockchain privada y empresarial desarrollada por la Fundación Linux. A diferencia de Ethereum y del resto de blockchains públicas,

Hyperledger Fabric está diseñada para redes privadas, lo que la hace ideal para aplicaciones empresariales.

Hyperledger Fabric ofrece una infraestructura modular que permite a las organizaciones personalizar su red blockchain según sus necesidades específicas.

A diferencia de Ethereum, Hyperledger Fabric soporta múltiples lenguajes de programación para escribir contratos inteligentes, incluyendo *Go* y *JavaScript*, lo que facilita la integración con sistemas empresariales existentes.

- Polkadot:

Polkadot es una plataforma de blockchain diseñada para facilitar la interoperabilidad entre diferentes blockchains. Polkadot permite que diferentes blockchains se comuniquen entre sí, compartan datos y transfieran activos.

Polkadot utiliza *Substrate*, un framework para construir blockchains personalizadas que pueden integrarse fácilmente con la red Polkadot. *Substrate* permite a los desarrolladores construir blockchains y smart contracts con funcionalidades específicas y conectarlas a Polkadot para aprovechar su seguridad e interoperabilidad.

Además, Polkadot implementa un sistema de gobernanza que permite a los titulares de tokens votar sobre las actualizaciones y cambios en el protocolo, asegurando que la red evolucione de manera descentralizada y comunitaria.

2.4 Importancia de la seguridad en smart contracts

La seguridad en los smart contracts es crucial debido a las características inherentes de las blockchains y el potencial impacto financiero, legal y reputacional de cualquier vulnerabilidad.

Una de las propiedades más destacadas de las blockchains es su inmutabilidad. Una vez que se registra una transacción o se despliega un contrato, no puede ser alterado ni eliminado. Aunque esta característica garantiza la integridad y la confianza, también significa que cualquier error en un smart contract no puede corregirse a menos que se realice una modificación total de la blockchain.

Muchos smart contracts gestionan activos valiosos, desde criptomonedas hasta tokens no fungibles (NFTs). Un error en el código puede resultar en la pérdida, robo o inmovilización de grandes sumas de dinero, como se ha visto en hackeos históricos que se estudiarán en secciones posteriores de este TFG. Por ejemplo, el hackeo de *The DAO* en 2016 resultó en la pérdida de aproximadamente 60 millones de dólares en Ether (IBM, 2024), lo que subraya la necesidad de robustez en el desarrollo y la auditoría de los contratos inteligentes.

La seguridad de los smart contracts es fundamental para mantener la confianza de los usuarios y la reputación de la plataforma blockchain. Los usuarios deben estar seguros de que sus transacciones y activos están seguros. Un historial de vulnerabilidades y hackeos puede erosionar la confianza en una plataforma específica y en el ecosistema blockchain en general.

Además, la seguridad de los smart contracts también tiene implicaciones legales y regulatorias. Los gobiernos y las autoridades reguladoras están cada vez más interesados en las tecnologías blockchain y DeFi. Incidentes de seguridad pueden acelerar la implementación de regulaciones estrictas.

3 Metodología.

Para llevar a cabo el desarrollo de este TFG y cumplir con los objetivos descritos en el primer apartado, se realiza un exhaustivo trabajo de investigación y análisis sobre multitud de fuentes primarias y secundarias.

El TFG se divide en dos partes:

Una primera parte de investigación sobre casos sonados de hackeos a blockchains por vulnerabilidades en smart contracts.

En esta primera parte, se seleccionan y analizan en profundidad tres casos famosos de hackeos a smart contracts: el hackeo de *The DAO* (2016), el hackeo de *Parity Wallet* (2017) y el hackeo de *Poly Network* (2021). Estos casos se eligen por su relevancia histórica, impacto en el ecosistema blockchain y las lecciones aprendidas en términos de seguridad en smart contracts.

Cada uno de estos casos será investigado y documentado detalladamente, destacando las vulnerabilidades explotadas, el impacto del ataque y las respuestas de la comunidad y las plataformas afectadas. La investigación se basará en informes técnicos, publicaciones académicas, noticias y otros recursos relevantes para proporcionar una visión completa y precisa de cada incidente. Estos recursos se obtendrán principalmente de Google y del portal ResearchGate. Dado que la documentación académica específica sobre ataques a contratos inteligentes es limitada, se recurrirá principalmente a artículos de portales reputados en el ámbito blockchain como lo es Medium.

En la segunda parte del TFG, se describirán y analizarán las principales vulnerabilidades de los smart contracts y se realizará una auditoría exhaustiva de ciertos contratos inteligentes previamente seleccionados de Github y con ciertas modificaciones de código. Se utilizarán herramientas especializadas como Oyente y Slither para la auditoría. El objetivo es identificar las vulnerabilidades más comunes en los smart contracts y proporcionar recomendaciones basadas en los hallazgos.

El proceso de auditoría consistirá en:

- Identificar y seleccionar un conjunto de contratos inteligentes representativos para el análisis.
- Utilizar Oyente y Slither para analizar los contratos seleccionados y detectar vulnerabilidades.
- Registrar y clasificar las vulnerabilidades encontradas, describiendo su naturaleza, impacto potencial y las formas en que pueden ser explotadas.
- Basándose en los hallazgos de la auditoría, se presentarán recomendaciones y mejores prácticas para el desarrollo seguro de contratos inteligentes.

Con esta metodología, el TFG no solo se basará en investigar los casos históricos más relevantes de hackeos a smart contracts, sino que también proporcionará un análisis práctico y basado en herramientas sobre cómo mejorar la seguridad de los contratos inteligentes en el futuro.

3.1 Herramientas.

Las herramientas que se utilizarán para el desarrollo y la auditoría de los smart contracts son las siguientes:

Visual Studio Code (VSC): Es un editor de código fuente desarrollado por Microsoft. Es gratuito, de código abierto y multiplataforma. VSC es ampliamente utilizado por desarrolladores debido a sus características robustas, incluyendo resaltado de sintaxis para una variedad de lenguajes de programación, autocompletado de código, depuración integrada con puntos de interrupción y seguimiento del flujo de ejecución, control de versiones integrado con Git, y extensiones y plugins que permiten personalizar y ampliar las funcionalidades del editor para adaptarse a necesidades específicas de desarrollo.

Solidity: Es un lenguaje de programación orientado a contratos diseñado para implementar contratos inteligentes en la blockchain de Ethereum. Es similar a JavaScript en su sintaxis y es un lenguaje estáticamente tipado. Algunas características clave de Solidity incluyen orientación a contratos, soporte para tipos complejos como *structs* y *mappings*, herencia y polimorfismo para reutilizar y extender contratos.

Documentación oficial: https://soliditylang.org/

Truffle: Es un framework para la creación, prueba y despliegue de contratos inteligentes. Su uso principal está enfocado en el lenguaje Solidity y la blockchain Ethereum, aunque es compatible con otros lenguajes y redes. Se trata una herramienta muy popular en el ecosistema de Ethereum debido a sus amplias capacidades, que incluyen gestión de proyectos, compilación y despliegue de contratos inteligentes en diferentes redes blockchain, pruebas automatizadas para verificar la funcionalidad y seguridad de los contratos, una interfaz de línea de comandos que proporciona comandos para manejar diferentes aspectos del ciclo de vida del contrato, e integración con herramientas externas como Ganache (blockchain local para pruebas) y Drizzle (biblioteca para gestionar el estado en aplicaciones descentralizadas).

Documentación oficial: https://archive.trufflesuite.com/

Ganache: Se trata de una herramienta de desarrollo de blockchain que permite a los desarrolladores crear una blockchain local y privada de Ethereum. Ofrece tanto una interfaz gráfica (Ganache GUI) como una interfaz de línea de comandos (Ganache CLI), facilitando la prueba y el despliegue de contratos inteligentes sin la necesidad de utilizar la red principal de Ethereum.

Documentación oficial: https://archive.trufflesuite.com/docs/ganache/

Slither: Es una herramienta de análisis estático desarrollada por *Trail of Bits*, diseñada para proporcionar un análisis detallado de los contratos escritos en el lenguaje de programación *Solidity.* Slither será utilizada para identificar patrones de código inseguros y vulnerabilidades conocidas en los contratos inteligentes.

Documentación oficial: https://github.com/crytic/slither

Oyente: Es una herramienta de análisis de seguridad estática para contratos inteligentes de Ethereum. Oyente examina el código del contrato para detectar posibles vulnerabilidades de seguridad. Funciona analizando el bytecode de los contratos en busca de patrones de comportamiento que puedan indicar fallos de seguridad, como *reentrancy, integer overflows/underflows* y otros errores comunes en el desarrollo de contratos inteligentes.

Documentación oficial: https://github.com/enzymefinance/oyente

3.2 Planificación.

La planificación de este TFG se distribuye en tareas de las que se estima un tiempo de realización a priori que puede variar según las necesidades o complicaciones que puedan surgir durante la elaboración del documento. Las tareas son secuenciales en su mayoría, aunque hay algunas que por su naturaleza se pueden desarrollar paralelamente, como puede ser la investigación de los casos de ataques a blockchains o la configuración del entorno para la auditoría, ya que esta tarea no está condicionada por ninguna otra y se centra puramente en la instalación y configuración del software necesario para realizar las auditorías de los contratos inteligentes. En el tiempo estimado de cada tarea se incluye tanto el trabajo de investigación como la documentación y la revisión.

Se espera que la elaboración de este TFG consuma un aproximado de 300 horas distribuidas en una media de 30 horas semanales durante 10 semanas.

Las tareas que comprende la elaboración de este TFG son las siguientes:

- Marco Teórico: Investigación y definición de conceptos clave y antecedentes del tema del TFG.
- Introducción y Metodología: Redacción de la introducción, objetivos y descripción de la metodología empleada en el TFG.
- The DAO: Estudio detallado del hackeo a *The DAO*, incluyendo descripción del ataque, impacto y consecuencias.
- Parity Wallet: Estudio detallado del hackeo a *Parity Wallet*, incluyendo descripción del ataque, impacto y consecuencias.
- Poly Network: Estudio del hackeo a *Poly Network*, incluyendo descripción del ataque, impacto y consecuencias.
- Análisis de Vulnerabilidades: Identificación y categorización de las principales vulnerabilidades presentes en los contratos inteligentes.
- Configuración del Entorno para la Auditoría: Preparación del entorno necesario para realizar las auditorías de seguridad.
- Selección de Proyecto: Elección del proyecto a auditar, asegurando que es adecuado para el análisis de vulnerabilidades.
- Estudio del Proyecto: Análisis y pruebas del proyecto seleccionado para comprender su estructura y funcionamiento.
- Auditoría del Proyecto: Realización de la auditoría de seguridad del proyecto utilizando herramientas especializadas.
- Mejores Prácticas: Propuesta de un conjunto de mejores prácticas para el desarrollo seguro de contratos inteligentes.
- Conclusiones: Resumen de los hallazgos y reflexión sobre los objetivos fijados al inicio del docuemento.
- Confección del Documento: Redacción y revisión final del documento del TFG.
- Reflexión ODS: Desarrollo de aportaciones del TFG a los ODS.

Además de estas tareas, también se prevé consumir algunas horas realizando seguimientos con el tutor del TFG.

En la siguiente figura 4 se muestra la planificación del TFG mediante un diagrama de Gantt.

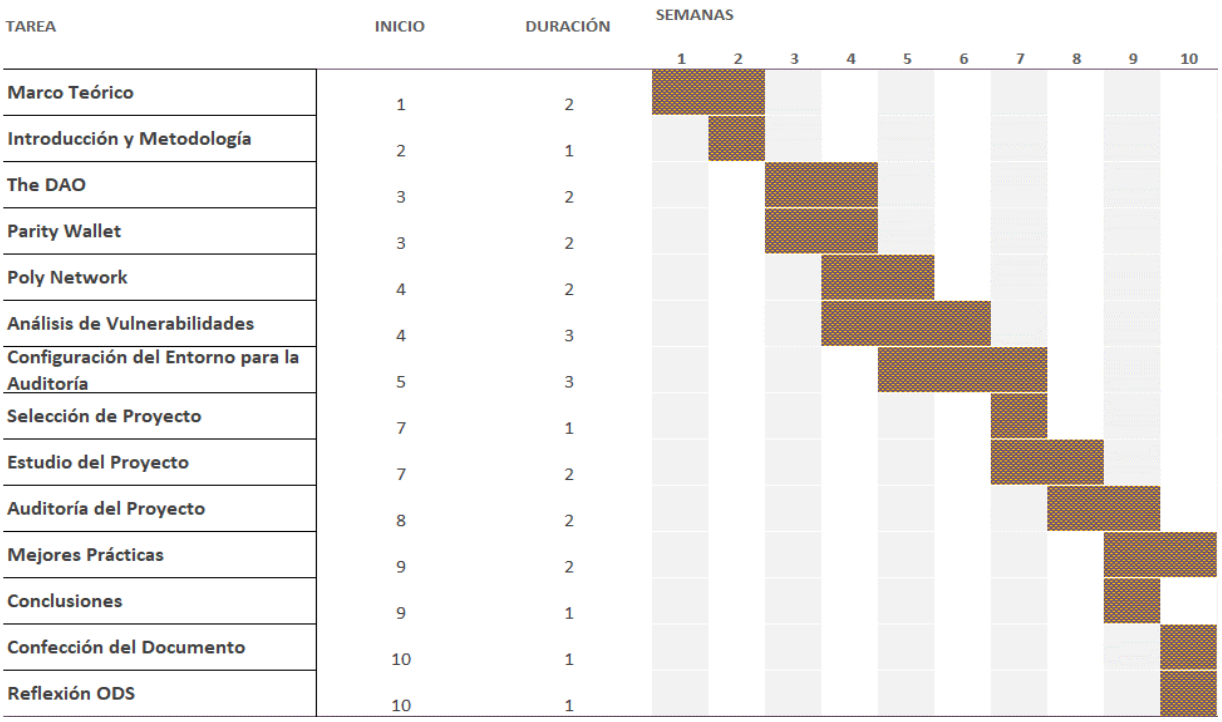


Figura 4: Diagrama de Gantt para la planificación del TFG

4 Estudio de casos famosos de hackeos a smart contracts.

4.1 The DAO Hack.

El caso de *The DAO* es uno de los incidentes más significativos en la historia de las criptomonedas y los contratos inteligentes. Este evento marcó un antes y un después en la percepción de la seguridad y la gobernanza dentro del ecosistema blockchain.

4.1.1 Descripción del ataque.

El nombre de este hackeo viene por explotar una vulnerabilidad sobre la que es considerada la primera DAO. Una organización autónoma descentralizada (DAO) es una cooperativa basada en blockchain que es propiedad de sus inversores, con reglas establecidas y ejecutadas mediante smart contracts. Las DAO reemplazan las estructuras de gestión centralizadas con un enfoque democrático en el que las decisiones son votadas por los inversores. Las DAO se construyen sobre blockchains (en este caso Ethereum) y sus transacciones son visibles en el protocolo subyacente de la blockchain.

El funcionamiento de una DAO es el siguiente:

- Primero, un grupo de individuos crea contratos inteligentes diseñados para gestionar la organización de manera autónoma, sin necesidad de intervención humana.
- El paso siguiente es una fase inicial de financiación, durante la cual las personas pueden adquirir tokens que representan la propiedad en la DAO.
- Después de la fase de financiación y una vez que la DAO está en funcionamiento, los miembros pueden presentar propuestas sobre cómo utilizar los fondos. Estas propuestas son sometidas a votación por los miembros que poseen tokens.

The DAO, creada sobre la red de Ethereum y cuyo objetivo principal era la financiación de un proyecto similar a Airbnb de forma descentralizada, tuvo una ventana de financiamiento de 28 días que comenzó el 30 de abril de 2016. Durante ese tiempo, recaudó más de 100 millones de dólares de más de 11.000 miembros, convirtiéndose en el mayor proyecto de crowdfunding en blockchain hasta ese momento (Gemini, 2023).

A pesar de su éxito en la recaudación de fondos, *The DAO* enfrentó dificultades y críticas debido a vulnerabilidades en su código. Estas preocupaciones fueron planteadas durante el período de financiación, pero no se abordaron hasta después de este periodo.

El 17 de junio de 2016 *The DAO* fue atacada aprovechando una vulnerabilidad conocida como ataque de llamada recursiva o “reentrancy atack” en inglés. Esta vulnerabilidad permite hacer llamadas repetidas a una función antes de que finalice su ejecución, permitiendo así múltiples retiros de fondos antes de actualizar el saldo del contrato inteligente. De este modo, el atacante fue capaz de extraer el equivalente a 60 millones de dólares en Ether. El hackeo ha sido atribuido a un grupo de hackers debido al alto nivel de experiencia requerido para comprender el problema en el código. Según una entrevista en línea no confirmada, el atacante se detuvo porque el agotamiento de los tokens DAO equivalentes a Ether haría bajar el valor del Ether como moneda comercializable (Santos 2018). Esta hipótesis tiene sentido ya que el total de Ether contenido en *The DAO* ascendía al 14% del total de Ether.

4.1.2 Impacto y consecuencias.

El 17 de junio de 2016, el fundador de Ethereum, Vitalik Buterin, se pronunció a través Twitter con un escrito:

> *"A software fork has been proposed, (with NO ROLLBACK; no transactions or blocks will be "reversed") which will make any transactions that make any calls/callcodes/delegatecalls that reduce the balance of an account with code hash0x7278d050619a624f84f51987149ddb439cdaadfba5966f7cfaea7ad44340a4ba (ie. the DAO and children) lead to the transaction (not just the call, the transaction) being invalid …"*
>
> — V. Buterin vía Twitter, 17 de junio de 2016

En este fragmento, Buterin propuso un "soft fork" en la red Ethereum, cuyo objetivo era incluir al atacante en una lista negra e impedirle acceder a los fondos robados. Un *soft fork* es una actualización de protocolo que es compatible con versiones anteriores, lo que significa que los nodos actualizados pueden interactuar con los nodos no actualizados. En otras palabras, es una pequeña actualización de la blockchain que debe ser aceptada por la mayoría de los participantes de la misma y que permite que nodos actualizados y no actualizados puedan minar bloques e interactuar entre ellos.

Sin embargo, en una carta abierta a los usuarios de Ethereum, el atacante respondió afirmando que sus acciones eran "legales" debido a la naturaleza rígida de los smart contracts. El atacante dejaba claro que el código del contrato inteligente era público y que, el *soft fork,* iba en contra de la naturaleza e inmutabilidad de la blockchain. Además, el atacante amenazaba con emprender acciones legales contra cualquiera que intentara apoderarse de los fondos (Gemini 2023).

Esta disputa provocó una división dentro de la comunidad de Ethereum. El atacante ofreció una recompensa colectiva de un millón de Ether y 100 bitcoins para sobornar a los mineros de Ethereum para que no acataran el *soft fork* propuesto. La comunidad se enfrentó a un dilema moral, cuestionando los principios de inmutabilidad y resistencia a la censura de la tecnología blockchain (Santos, 2018).

Finalmente, la comunidad decidió llevar a cabo un "hard fork" debido a la preocupación por las posibles acciones del atacante y las importantes implicaciones financieras para los inversores. Un *hard fork* es una actualización de protocolo no compatible con versiones anteriores, lo que crea una bifurcación permanente en la blockchain, es decir, se pasa a tener dos blockchains que se bifurcan a partir de un bloque.

El *hard fork* pretendía retrotraer la historia de la red Ethereum a un punto anterior al ataque de *The DAO* y reasignar los fondos a un contrato inteligente diferente. Esta decisión fue controvertida, ya que algunos argumentaron que violaba los principios básicos de descentralización e inmutabilidad.

Algunos críticos argumentaron que la decisión de realizar un *hard fork* sentaba un precedente para futuras intervenciones y socavaba la fiabilidad de la tecnología blockchain. Por otro lado, los defensores creían que demostraba la capacidad de la comunidad para adaptarse y abordar problemas complejos con rapidez.

Finalmente, el 20 de julio de 2016, se implementó el *hard fork* en el bloque 192.000 de la red de Ethereum.

Aunque la gran mayoría de los mineros adoptaron el cambio y se implementó la bifurcación, no todos estuvieron de acuerdo. Como resultado, el *hard fork* dio lugar a dos blockchains de Ethereum competidoras y ahora separadas. Aquellos que se negaron a aceptar la bifurcación que revertía el historial de la blockchain apoyaron la versión anterior al *hard fork*, ahora conocida como Ethereum Classic (ETC). La blockchain que actualmente se conoce como Ethereum es la que implementó el *hard fork* y alteró la historia de Ethereum, así como la historia de la tecnología blockchain en general.

Aunque los fondos robados de *The DAO* fueron restaurados a sus inversores, el atacante no perdió por completo todo su botín. Los tokens robados aún permanecían en su posesión en la cadena de Ethereum Classic y valían alrededor de 8,5 millones de dólares en los meses posteriores al ataque.

El hackeo de *The DAO* y la posterior bifurcación de Ethereum sacudieron profundamente a la comunidad de Ethereum y sacaron cuestiones importantes sobre la tecnología blockchain. En retrospectiva, está claro que las decisiones tomadas por Vitalik Buterin, los desarrolladores de Ethereum y la comunidad global garantizaron la supervivencia de la blockchain en sus momentos más críticos, pero las soluciones adoptadas fueron totalmente contrarias a los principios básicos de esta tecnología.

4.2 Parity Wallet Hack.

El hackeo de la *Parity Wallet* es uno de los incidentes más sonados en la historia de Ethereum y de la tecnología blockchain en general, dejando una valiosa lección sobre la importancia de la seguridad en el desarrollo de contratos inteligentes. *Parity Technologies*, una compañía prominente en el espacio de blockchain, ofrecía una de las carteras más utilizadas para almacenar Ether y otros tokens ERC-20 (red de Ethereum). A pesar de su popularidad y la confianza que muchos usuarios depositaron en su software, en 2017, *Parity* fue víctima de un hackeo que resultó en la pérdida de millones de dólares en criptomonedas.

4.2.1 Descripción del ataque.

El hackeo de *Parity Wallet* fue llevado a cabo sobre las billeteras multifirma que ofrecía la compañía *Parity Technologies*. Una billetera multifirma o "multisig wallet" en inglés, es una wallet que ofrece unas características avanzadas de seguridad. Estas carteras requieren múltiples firmas (o claves privadas) para autorizar una transacción, en lugar de una sola firma, lo que agrega una capa adicional de protección contra el acceso no autorizado y el fraude.

En una configuración típica de una cartera multisig, se pueden requerir varias firmas para aprobar una transacción. Por ejemplo, en una configuración *2 de 3*, se necesitan dos de tres posibles firmas para validar y ejecutar una transacción.

El ataque a Parity Wallet ocurrió en dos fases distintas:

- Fase 1: Hackeo de julio de 2017:

El 20 julio de 2017, un atacante explotó una vulnerabilidad en la implementación del contrato inteligente de *Parity* para carteras multifirma. Esta vulnerabilidad estaba relacionada con la inicialización incorrecta de la biblioteca del contrato inteligente de las billeteras multisig, lo que permitió al atacante hacerse propietario del contrato y transferir los fondos a su propia cuenta. Este fallo en el contrato permitió al atacante drenar aproximadamente 150.000 Ether, valorados en ese momento en alrededor de 30 millones de dólares (Medium, 2023). Esta vulnerabilidad fue rápidamente identificada y parcheada, pero el incidente subrayó las vulnerabilidades inherentes en los contratos inteligentes y la necesidad de auditorías exhaustivas y revisiones de seguridad.

- Fase 2: Congelación de fondos en noviembre de 2017:

El segundo y más devastador incidente ocurrió el 6 de noviembre de 2017. Un usuario accidentalmente activó una función en la biblioteca del contrato que le asignaba amplios permisos sobre dicho contrato, posteriormente, tras tener los permisos, destruyó accidentalmente dicha biblioteca. Dado que muchas carteras multifirma de *Parity* dependían de esta biblioteca, la autodestrucción tuvo como resultado la congelación de más de 500.000 Ether, equivalentes a más de 150 millones de dólares en ese momento (Bastardo, 2017). La congelación de fondos se refiere a una situación en la que los fondos almacenados en una cuenta o contrato inteligente se vuelven inaccesibles o inutilizables.

A través de Twitter, el usuario @devops199 se pronunció y aseguró haber causado este incidente mientras investigaba el código del contrato que estaba publicado en GitHub. El usuario afirmaba ser novato en el mundo de la programación y en las tecnologías blockchain y se mostró preocupado en foros por posibles implicaciones legales.

> *"Es realmente simple, imagina que te acercas a la puerta de un banco y hay un botón que dice «Bloquear para siempre» …. y alguien lo presiona accidentalmente."*
>
> — @devops199 vía Twitter, 7 de noviembre de 2017 (traducido del inglés)

Este caso demostró una vez más los peligros de los errores en los contratos inteligentes y la gravedad de las consecuencias que pueden resultar de una implementación defectuosa de los mismos.

4.2.2 Impacto y consecuencias.

Aunque el caso de *Parity Wallet* no tuvo tanto impacto como el de *The DAO*, ya que no supuso una bifurcación de la blockchain, los costes para los inversores y la pérdida de confianza en el protocolo de *Parity* y en la red de Ethereum fueron considerables.

La pérdida de más de 150 millones de dólares en Ether subrayó la necesidad urgente de mejorar las prácticas de seguridad y las auditorías en el desarrollo de contratos inteligentes.

Como consecuencia del ataque, *Parity Technologies* estableció un fondo para compensar a los usuarios que perdieron sus fondos en el hackeo. El fondo, llamado "Ethereum Recovery Initiative", tiene como objetivo ayudar a los afectados y restaurar la confianza en el ecosistema de la cartera *Parity.* Sin embargo, a día de hoy no se ha devuelto ninguna parte de los fondos robados o congelados a los afectados (Medium 2023).

Algunas de las lecciones que deja este caso incluyen:

- La importancia de las auditorías exhaustivas, realizadas por múltiples equipos independientes para identificar posibles vulnerabilidades desde diferentes perspectivas.
- La implementación de mejores prácticas de codificación, con controles estrictos y validaciones en todas las funciones críticas del contrato para prevenir la explotación de vulnerabilidades.
- La necesidad de transparencia y comunicación, donde las compañías deben mantener una comunicación clara con sus usuarios sobre los riesgos y medidas de seguridad implementadas, así como sobre los incidentes ocurridos.

El incidente también llevó a la comunidad de Ethereum a discutir y proponer mejoras en la gobernanza y en las metodologías de desarrollo de contratos inteligentes, fomentando una mayor colaboración en el desarrollo de estándares de seguridad y prácticas recomendadas.

4.3 Poly Network Hack.

El caso de *Poly Network* destaca como uno de los más grandes hackeos en cuanto a cantidad robadas y por la eventual devolución de los fondos a los usuarios por parte del atacante. Ocurrió en agosto de 2021 y expuso serias vulnerabilidades en la seguridad de los contratos inteligentes y la interoperabilidad entre diferentes blockchains.

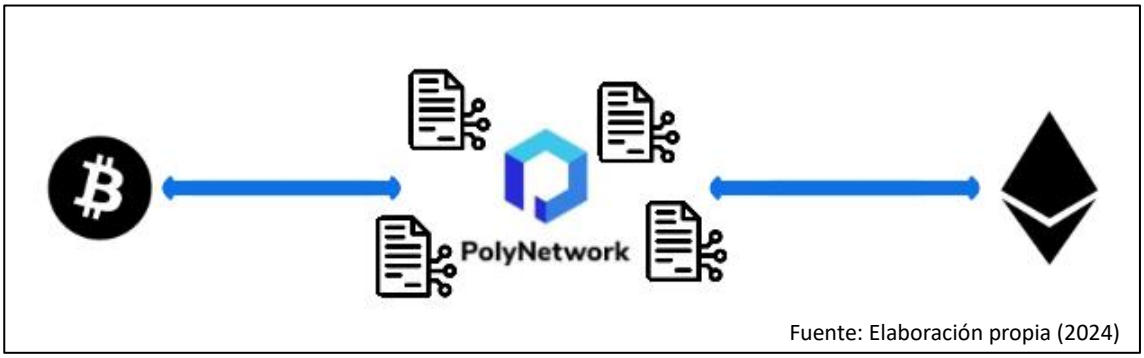
4.3.1 Descrpción del ataque.

Poly Network es un protocolo descentralizado de tipo cross-chain, que permite la interoperabilidad entre múltiples blockchains, lo que significa que los activos digitales y la información pueden transferirse sin problemas entre diferentes redes de blockchain. Su objetivo principal es abordar el problema de la fragmentación en el ecosistema blockchain, permitiendo que diferentes redes interactúen entre sí de manera eficiente y segura.

Las cross-chain suelen estar formadas por los siguientes elementos:

- Una billetera maestra para cada una de las redes subyacentes (por ejemplo, una para Bitcoin, una para Ethereum, etc.), cada una de ellas conteniendo una cierta cantidad de fondos.
- Un conjunto de contratos inteligentes que interpretan y ejecutan las instrucciones de los usuarios (ej. cambiar Bitcoin por Ether) llamando a funciones en las billeteras mencionadas anteriormente.
- Una capa de blockchain (la red *Poly* en este caso) donde se ejecutan los contratos inteligentes mencionados anteriormente.

En la siguiente figura 5 se puede ver un diagrama de interoperabilidad entre las redes de Bitcoin y Ethereum a través de la red de *Poly* y sus smart contracts.



Fuente: Elaboración propia (2024)

Figura 5: Diagrama de funcionamiento de un protocolo cross-chain

El 10 de agosto de 2021, *Poly Network* informó que un atacante no identificado hackeó varios contratos inteligentes de la red, transfiriendo el equivalente a aproximadamente 610 millones de dólares (principalmente en Ether, BNB y en la stablecoin USDC) a direcciones de billeteras externas (Gagliardoni, 2021).

Según la firma de ciberseguridad *SlowMist* y el investigador de seguridad Kelvin Fichter, el hackeo fue posible debido a una mala gestión de los derechos de acceso entre dos contratos inteligentes importantes de *Poly*. El primero es *EthCrossChainManager* y el segundo es *EthCrossChainData* (SlowMist, 2021).

Más en detalle y atendiendo por separado a ambos contratos comprometidos:

- *EthCrossChainData* es un contrato inteligente con altos privilegios que gestiona una lista de claves públicas de nodos autenticadores o keepers, quienes controlan las billeteras de la red. El contrato debe ser invocado solo por sus propietarios. El atacante explotó la función *putCurEpochConPubKeyBytes* para cambiar la clave pública de un keeper por su propia clave, obteniendo así el control para mover grandes cantidades de fondos.

- *EthCrossChainManager* es un contrato inteligente que puede iniciar mensajes entre cadenas. Permite a cualquiera emitir un evento entre cadenas mediante una transacción que invoque la función *verifyHeaderAndExecuteTx* y especificar un contrato objetivo en *Poly*. El atacante utilizó esta capacidad para ejecutar la función que le permitió acceder y transferir fondos.

En resumen, el atacante explotó un fallo en la lógica de los contratos que permitía modificar los identificadores de las transacciones y redirigir los fondos a direcciones controladas por él mismo.

4.3.2 Impacto y consecuencias.

Después del hackeo, el siguiente paso lógico habría sido que el atacante transfiriera los fondos robados al pool de liquidez de un exchange descentralizado anónimo. Por esta razón, *Poly* emitió de inmediato una solicitud a los mineros de criptomonedas y exchanges para "poner en la lista negra" los fondos robados, haciéndolos de facto inaccesibles para el atacante. Esto es teóricamente posible, pero complicado para los exchanges por muchas razones, y no está claro si todas las plataformas de intercambio respondieron a la petición.

Sin embargo, los administradores de la stablecoin Tether lograron congelar los fondos robados en Tether-USDC a tiempo, justo 9 bloques antes de que el atacante intentara lavarlos en el pool de liquidez de Curve (exchange descentralizado anónimo) (Gaglardoni, 2021).

Poly Network pidió al hacker que devolviera los fondos. La empresa de seguridad *SlowMist* publicó hallazgos sobre el presunto hacker, afirmando que la identidad del hacker había sido expuesta y que el grupo tenía acceso al correo electrónico y la dirección IP del hacker.

Pocos días después del incidente, el hacker comenzó a devolver fondos a *Poly*. No se saben los motivos exactos de este acto de buena fe, pero por el comportamiento previo del atacante, se atribuye al miedo de represalias legales (SlowMist, 2021).

Para el 11 de agosto de 2021, casi la mitad del valor de los tokens había sido devuelto, y el hacker afirmó estar dispuesto a devolver más a cambio del descongelamiento de los tokens Tether.

Finalmente, el atacante devolvería todos los fondos a cambio de mantener su anonimato y de una recompensa de la que se desconoce su cuantía y si finalmente se llevó a cabo.

5 Análisis de vulnerabilidades.

En este apartado, se abordarán las vulnerabilidades más críticas que han sido identificadas en el desarrollo y la implementación de smart contracts. Algunas de estas vulnerabilidades son las que han permitido los hackeos estudiados en los apartados anteriores. Analizar cada una de estas fallas de seguridad nos permitirá comprender mejor los riesgos inherentes a la tecnología blockchain y servirán de base para las auditorías de smart contracts en apartados posteriores.

5.1 Reentrancy.

La vulnerabilidad de llamada recursiva o *reentrancy* es una de las fallas más críticas en los contratos inteligentes y fue la causa principal del hackeo de *The DAO* estudiado anteriormente.

5.1.1 Descripción.

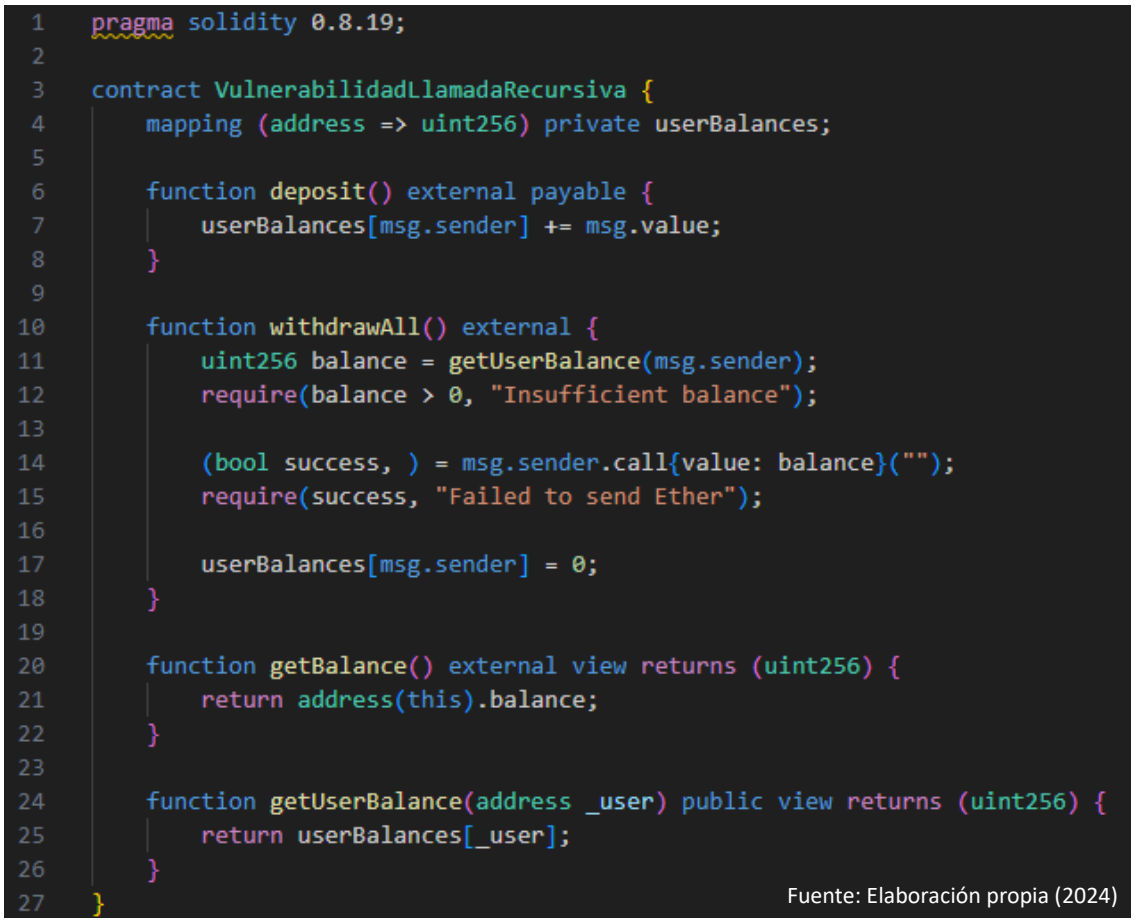
La vulnerabilidad de llamada recursiva ocurre cuando una función de un contrato inteligente realiza una llamada externa a otro contrato antes de completar sus propias transacciones. Si el contrato externo vuelve a llamar a la función original (o a cualquier otra función del mismo contrato) antes de que la primera llamada haya terminado, esto puede causar una serie de efectos no deseados. Este comportamiento se denomina "reentrancy" porque la función se vuelve a ejecutar antes de haber finalizado.

Considerando un contrato que permite retirar fondos. El proceso típico sería:

1- Verificar que el usuario tenga fondos suficientes.
2- Enviar los fondos al usuario.
3- Actualizar el balance del usuario en el contrato.

Si el contrato permite una llamada externa entre los pasos 2 y 3, un atacante puede explotar esta situación. Específicamente, el atacante puede hacer que la llamada externa vuelva a invocar la función de retiro antes de que el balance del usuario se haya actualizado, permitiendo así múltiples retiros fraudulentos.

En la figura 6 se muestra el código de un contrato inteligente que gestiona el balance de Ether de un usuario y cuenta con la vulnerabilidad de llamada recursiva en la función *withdrawAll().*



```
1   pragma solidity 0.8.19;
2
3   contract VulnerabilidadLlamadaRecursiva {
4       mapping (address => uint256) private userBalances;
5
6       function deposit() external payable {
7           userBalances[msg.sender] += msg.value;
8       }
9
10      function withdrawAll() external {
11          uint256 balance = getUserBalance(msg.sender);
12          require(balance > 0, "Insufficient balance");
13
14          (bool success, ) = msg.sender.call{value: balance}("");
15          require(success, "Failed to send Ether");
16
17          userBalances[msg.sender] = 0;
18      }
19
20      function getBalance() external view returns (uint256) {
21          return address(this).balance;
22      }
23
24      function getUserBalance(address _user) public view returns (uint256) {
25          return userBalances[_user];
26      }
27  }
```

Fuente: Elaboración propia (2024)

Figura 6: Contrato con vulnerabilidad reentrancy

En este código:

- La función *withdrawAll()* permite a un usuario retirar todo su balance almacenado en el contrato.
- La línea *uint256 balance = getUserBalance(msg.sender);* obtiene el balance del usuario.
- La línea *require(balance > 0, "Insufficient balance");* asegura que el usuario tiene un balance positivo.
- La llamada *msg.sender.call{value: balance}("");* envía Ether al usuario.

El problema radica en la línea 14 en la llamada *msg.sender.call{value: balance}("");* Esta llamada externa envía Ether al usuario, que podría ser un contrato inteligente. Este contrato podría tener una función *fallback* o *receive* que se ejecutara al recibir Ether. Dentro de esa función, el contrato atacante podría llamar nuevamente a *withdrawAll*, antes de que la línea *userBalances[msg.sender] = 0;* se ejecute en la llamada original.

Como resultado, la llamada recursiva permitiría al atacante drenar el balance del contrato en múltiples iteraciones, explotando así la vulnerabilidad.

5.1.2 Prevención.

Aunque una solución obvia para esta vulnerabilidad podría ser usar mecanismos de control de concurrencia, en Solidity no están disponibles semáforos ni otros mecanismos tradicionales debido a la naturaleza secuencial y *singlethread* del lenguaje.

Para prevenir la llamada recursiva (*reentrancy*), los desarrolladores de contratos inteligentes deben seguir buenas prácticas de codificación, tales como:

- Efectuar los cambios de estado antes de realizar llamadas externas: Asegurarse de actualizar todos los balances y estados antes de realizar cualquier llamada externa.
- Uso de patrones de "pull" en lugar de "push": En lugar de enviar fondos automáticamente, permitir que los usuarios retiren sus fondos de manera proactiva.
- Utilizar bibliotecas de contratos inteligentes que implementen patrones de diseño seguros y ayudan a mitigar riesgos de llamada recursiva.

El contrato de la figura 6 se podría corregir fácilmente actualizando el estado del balance del usuario antes de realizar la llamada externa. Moviendo la línea 17 *userBalances[msg.sender] = 0;* a la línea 13.

En la figura 7 se puede ver la función *withdrawAll* y el contrato corregido.

```
function withdrawAll() external {
    uint256 balance = getUserBalance(msg.sender);
    require(balance > 0, "Insufficient balance");
    userBalances[msg.sender] = 0; //actualiza balance antes de llamar a la función externa
    (bool success, ) = msg.sender.call{value: balance}("");
    require(success, "Failed to send Ether");
}
```

Fuente: Elaboración propia (2024)

Figura 7: Función withdrawAll sin reentrancy

5.2 Overflow y underflow.

5.2.1 Descripción.

El desbordamiento (overflow) y el subdesbordamiento (underflow) son vulnerabilidades comunes en contratos inteligentes que se producen debido a la manera en que se manejan las operaciones aritméticas en Solidity y en otros lenguajes de programación de smart contracts. Estas vulnerabilidades pueden permitir a un atacante manipular los cálculos de manera que se obtengan resultados inesperados, comprometiendo la lógica del contrato y causando pérdidas financieras.

El *overflow* ocurre cuando una operación aritmética produce un valor que excede el tamaño máximo del tipo de datos. En Solidity, los enteros sin signo (uint) tienen un tamaño fijo (por ejemplo, *uint256* tiene un rango de 0 a 2^{256}- 1). Si una operación excede este rango, el valor vuelve al inicio del rango, comenzando de nuevo desde cero (Documentación Solidity, 2024).

El *underflow* es el caso contrario, ocurre cuando una operación aritmética produce un valor por debajo del tamaño mínimo del tipo de datos. En este caso el valor empezará a ubicarse al final del rango.

Esta vulnerabilidad puede tener consecuencias muy graves que pueden llevar a robo de fondos o manipulación de la lógica del contrato.

5.2.2 Prevención.

La vulnerabilidad de *overflow* o *underflow* es considerada como una muy básica y sencilla de corregir. De hecho, a partir de la versión Solidity 0.8.0, las operaciones aritméticas incluyen comprobaciones automáticas para *overflow* y *underflow*. Si se detecta un desbordamiento o un subdesbordamiento, la transacción se revierte automáticamente.

Antes de esta actualización, era común utilizar librerías especializadas para protegerse, como puede ser la librería *SafeMath* de OpenZeppelin (Weston, 2024).

5.3 Vulnerabilidad de generación de números aleatorios.

5.3.1 Descripción.

La generación de números aleatorios es crucial en muchos contratos inteligentes, especialmente en aquellos que dependen de la aleatoriedad para garantizar la imparcialidad, como los juegos de azar y las loterías. Sin embargo, generar números verdaderamente aleatorios en la blockchain es un desafío debido a la naturaleza determinística de los contratos inteligentes.

Habitualmente, los programadores hacían uso de ciertas funciones que recuperan valores de los bloques de la blockchain para usarlos como aleatorios, algunos ejemplos de estas funciones son: *block.number, block.timestamp* o *blockhash.* Sin embargo, los mineros pueden conocer el valor que devuelven estas funciones con anterioridad, por lo que, en algunos casos, mineros maliciosos pueden manipular y conocer las variables para obtener beneficios.

En la figura 8, se muestra un contrato inteligente con una función *Bet* que contiene la vulnerabilidad de generación de número aleatorio al hacer uso de las funciones nombradas anteriormente para obtener dicho valor aleatorio.

```solidity
pragma solidity 0.8.19;

contract Dado {
    function Bet(uint256 userNumber) public payable {
        require(msg.value == 1 ether);
        // Generar números aleatorios basados en múltiples semillas
        uint256 targetNumber = uint256(keccak256(abi.encodePacked(blockhash(block.number - 1), block.timestamp)));
        require(targetNumber == userNumber, "Better next time");
        // Transferir el fondo del premio al usuario
        (bool success, ) = msg.sender.call{value: address(this).balance}("");
        require(success, "Failed to send");
    }
}
```

Fuente: Elaboración propia (2024)

Figura 8: Contrato con vulnerabilidad de generación de número aleatorio

La función *Bet* permite a los usuarios apostar a un número específico enviando 1 Ether.

Este contrato utiliza *keccak256*, el blockhash del bloque anterior y el timestamp del bloque actual para generar un número aleatorio (targetNumber). Después verifica si targetNumber coincide con el número apostado por el usuario (userNumber). Si no coincide, muestra el mensaje "Better next time". Si el usuario gana, transfiere todo el balance del contrato al usuario.

En la figura 9 se muestra el código que podría usar un atacante con influencia sobre la blockchain para comprometer el contrato anterior mediante la predicción de los números aleatorios.

```solidity
pragma solidity 0.8.19;

contract Hacker {
    Dice public dice;

    function attack(Dice dice) external {
        // Calcular números aleatorios ganadores por adelantado
        uint256 targetNumber = uint256(keccak256(abi.encodePacked(blockhash(block.number - 1), block.timestamp)));
        // Atacar usando un número aleatorio ganador calculado por adelantado
        dice.Bet(targetNumber);
    }
}
```

Fuente: Elaboración propia (2024)

Figura 9: Código atacante por vulnerabilidad de número aleatorio

El contrato Hacker tiene una referencia pública al contrato Dice.

La función *attack* calcula el targetNumber de la misma manera que el contrato Dice. Después, se llama a la función *Bet* del contrato Dice con el targetNumber ya calculado, asegurando una ganancia.

La vulnerabilidad en estos contratos radica en el método de generación de números aleatorios. Utilizar *blockhash* y *block.timestamp* no proporciona una verdadera aleatoriedad, ya que estos valores son predecibles dentro del contexto de la blockchain.

5.3.2 Prevención.

Para mitigar este tipo de vulnerabilidades en la generación de números aleatorios, se recomienda usar fuentes de aleatoriedad más seguras, como oráculos de aleatoriedad en blockchains dedicadas a los oráculos, como puede ser Chainlink. Estos oráculos proporcionan números aleatorios impredecibles y verificables en la cadena (Sayeed et al, 2020).

5.4 Falta de control de acceso.

5.4.1 Descripción.

La falta de control de acceso es una de las vulnerabilidades más críticas y comunes en los contratos inteligentes. Esta vulnerabilidad ocurre cuando un contrato no restringe adecuadamente las funciones críticas a usuarios autorizados, permitiendo que atacantes externos ejecuten acciones que deberían estar reservadas para administradores o roles específicos.

Ejemplos de ataques que aprovecharon esta vulnerabilidad son los ya estudiados en apartados anteriores: *The DAO* y *Poly Network*. Se recuerda que en el hackeo a *The DAO*, la falta de validación de permisos adecuados permitió a un atacante explotar la vulnerabilidad de *reentrancy*. En el caso de *Poly Network*, las fallas en el control de acceso de dos contratos críticos permitieron a un atacante modificar las claves públicas de los nodos autenticadores, dándoles el control sobre grandes cantidades de fondos.

En la figura 10 se muestra un contrato que contiene una vulnerabilidad de control de acceso, ya que la función *adminWithdrawAll* no tiene ninguna restricción de acceso, lo que permite a cualquier usuario retirar todos los fondos del contrato.

```solidity
pragma solidity ^0.8.0;

contract ContratoSinControlDeAcceso {
    mapping(address => uint256) private balances;

    // Función para depositar fondos
    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    // Función para retirar todos los fondos
    function withdrawAll() public {
        uint256 amount = balances[msg.sender];
        require(amount > 0, "Insufficient balance");
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");
        balances[msg.sender] = 0;
    }

    // Función solo para administradores para retirar todos los fondos del contrato
    function adminWithdrawAll() public {
        uint256 amount = address(this).balance;
        require(amount > 0, "Insufficient balance");
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");
    }
}
```

Fuente: Elaboración propia (2024)

Figura 10: Contrato con vulnerabilidad de control de acceso

5.4.2 Prevención.

Para mitigar esta vulnerabilidad, se deben implementar controles de acceso adecuados. Esto se puede lograr utilizando patrones de diseño como el de "Ownable" a través de una biblioteca ofrecida por OpenZeppelin, donde solo el propietario del contrato puede ejecutar ciertas funciones. El contrato de la figura 11 implementa este patrón, securizando de este modo la función *adminWithdrawAll* y el contrato en general.

```solidity
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/access/Ownable.sol";

contract ContratoSeguro is Ownable {
    mapping(address => uint256) private balances;

    // Función para depositar fondos
    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    // Función para retirar todos los fondos
    function withdrawAll() public {
        uint256 amount = balances[msg.sender];
        require(amount > 0, "Insufficient balance");
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");
        balances[msg.sender] = 0;
    }

    // Función solo para administradores para retirar todos los fondos del contrato
    function adminWithdrawAll() public onlyOwner {
        uint256 amount = address(this).balance;
        require(amount > 0, "Insufficient balance");
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");
    }
}
```

Fuente: Elaboración propia (2024)

Figura 11: Contrato con acceso seguro

En este código, la función *adminWithdrawAll* está protegida con el modificador *onlyOwner*, garantizando que solo el propietario del contrato pueda ejecutarla.

5.4 Otras vulnerabilidades.

En esta sección, se explorarán otras vulnerabilidades críticas que pueden afectar la seguridad de los contratos inteligentes. Estas vulnerabilidades no tienen que ver directamente en el desarrollo de los contratos inteligentes pero pueden tener impactos devastadores si no se gestionan adecuadamente. A continuación, se describen algunas de las más importantes:

- Ataques de Phishing

Los ataques de *phishing* en el contexto de contratos inteligentes generalmente involucran la creación de contratos maliciosos que imitan a contratos legítimos, engañando a los usuarios para que interactúen con ellos y comprometiendo sus fondos.

Esta vulnerabilidad está más relacionada con las medidas de seguridad que puedan implementar los propios usuarios y las plataformas para evitar este tipo de ataques. Para hacer frente a estos ataques es necesario educar a los usuarios sobre la verificación de contratos antes de interactuar con ellos y utilizar interfaces de usuario seguras que

muestren claramente los detalles de los contratos inteligentes con los que están interactuando (Ledger, 2023).

- Denegación de servicio (DoS)

La Denegación de Servicio (DoS) es una vulnerabilidad delicada en los contratos inteligentes y puede impedir el funcionamiento correcto de una función o del contrato completo. En el contexto de la blockchain, esta vulnerabilidad puede surgir cuando una función consume demasiado gas, impidiendo que se complete la transacción, o cuando un atacante explota ciertas condiciones del contrato para evitar que otros usuarios interactúen con él.

Esto es particularmente problemático en Ethereum y otras blockchains basadas en EVM, donde la ejecución de transacciones está limitada por el gas disponible. Por ejemplo, un atacante podría enviar datos maliciosos o ejecutar una función repetidamente hasta agotar el gas, causando fallos en el contrato y afectando a todos los usuarios que dependen de él (SlowMist, 2023). En estos casos, la resiliencia del contrato y su capacidad para manejar condiciones inesperadas son cruciales para prevenir ataques DoS.

- Transaction Ordering Dependence (TOD)

La Dependencia del Orden de Transacciones (TOD) es una vulnerabilidad que puede permitir que los mineros corruptos tengan un efecto significativo en los contratos inteligentes. Esta vulnerabilidad es un error de seguridad muy común en los contratos inteligentes, ya que estos dependen del orden de ejecución de las transacciones. Si un bloque recién generado contiene dos transacciones que afectan el mismo contrato inteligente, si el resultado de ambas transacciones depende del orden, el contrato resulta en una vulnerabilidad TOD (Sayeed et al, 2020).

En la red de Ethereum (y en la mayoría de las blockchains), los mineros son responsables de controlar el orden de las transacciones, priorizando aquellas con mayor gas (comisión). Por lo tanto, cualquier minero que cierre un bloque puede influir en el orden de una o varias transacciones, pudiendo alterar el comportamiento del contrato.

Como conclusión, es importante conocer el origen y funcionamiento de estas vulnerabilidades adicionales, ya que son esenciales para garantizar la seguridad y la integridad de los contratos inteligentes. La implementación de buenas prácticas de seguridad y la utilización de herramientas de auditoría robustas pueden ayudar a prevenir estos problemas y proteger los activos de los usuarios en el ecosistema blockchain.

6 Auditoría de smart contracts.

Tal y como se ha resaltado a lo largo de este documento, la auditoría de smart contracts es una práctica esencial para garantizar la seguridad y fiabilidad de los contratos inteligentes desplegados en una blockchain. Este proceso implica revisar el código en busca de vulnerabilidades, errores lógicos y cualquier otro tipo de fallo que pueda comprometer la integridad del contrato. En esta sección, se explorará el proceso de auditoría de smart contracts, utilizando herramientas especializadas para identificar y corregir posibles problemas antes de su implementación en una red de producción.

6.1 Proceso de auditoría de smart contracts.

El proceso de auditoría de contratos inteligentes implica una serie de pasos meticulosos diseñados para identificar y corregir vulnerabilidades de seguridad antes de su implementación en una blockchain.

Inicialmente, se realiza una revisión exhaustiva del código fuente del contrato para detectar posibles errores y vulnerabilidades. Luego, se emplean herramientas automatizadas como Slither y Oyente para realizar análisis estáticos y dinámicos, identificando problemas comunes como *reentrancy*, desbordamientos y subdesbordamientos de enteros.

Posteriormente, se ejecutan pruebas unitarias y de integración para verificar que cada componente del contrato funcione correctamente y de manera segura. Adicionalmente, se llevan a cabo pruebas de seguridad para simular ataques potenciales y evaluar la resistencia del contrato.

Finalmente, se documentan los hallazgos y se emiten recomendaciones para mitigar las vulnerabilidades detectadas, asegurando que el contrato inteligente sea seguro y fiable antes de su despliegue.

6.2 Proyecto a auditar.

Para realizar la auditoría de smart contracts, se ha seleccionado un proyecto de exchange descentralizado (DEX) sencillo disponible en Github, desarrollado por el usuario *ltfschoen*.

El proyecto DEX seleccionado permite realizar operaciones básicas de intercambio de tokens en una plataforma descentralizada. Está escrito en Solidity, el lenguaje de programación para contratos inteligentes en Ethereum, y utiliza el framework Truffle para compilar, desplegar y probar los contratos. Adicionalmente, el proyecto está configurado para ejecutarse en una blockchain local de pruebas utilizando Ganache, un

software que simula una blockchain para desarrollo y pruebas. En el ANEXO II: Funcionamiento de un DEX se encuentra la explicación del funcionamiento típico de un DEX.

En su forma original, el proyecto no contiene vulnerabilidades conocidas, lo que lo convierte en una base ideal para la introducción y detección de fallos de seguridad de forma controlada.

En la figura 12 se muestra la estructura de carpetas y ficheros del proyecto DEX.

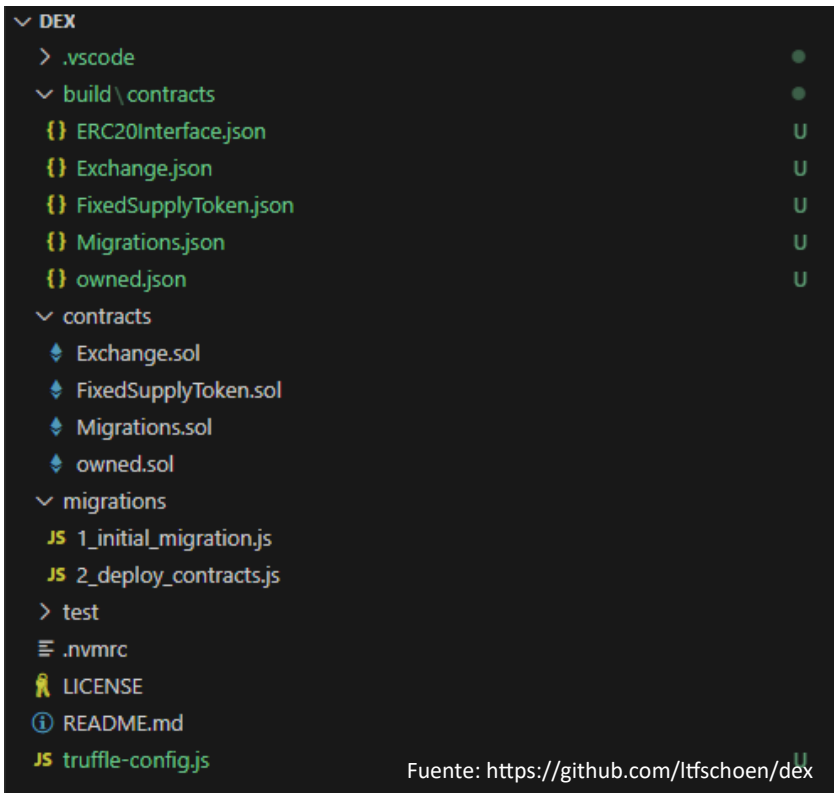


Figura 12: Estructura del proyecto DEX

Se respeta la estructura típica de los proyectos implementados mediante Truffle:

- contracts/: Contiene todos los contratos inteligentes en Solidity.
- migrations/: Contiene los scripts de migración para desplegar los contratos inteligentes en la blockchain.
- test/: Contiene los archivos de prueba para los contratos inteligentes. Pueden ser escritos en JavaScript o Solidity.
- build/: Contiene los archivos compilados de los contratos inteligentes y otros artefactos generados por Truffle.
- truffle-config.js: Archivo de configuración principal de Truffle. Define las redes de blockchain, configuraciones de compilación, y otros ajustes del proyecto.

Los contratos inteligentes que contiene el proyecto son los siguientes:

- Exchange.sol:

Este contrato inteligente implementa la lógica principal del exchange descentralizado. Permite a los usuarios intercambiar tokens entre sí de manera segura y eficiente. Maneja la lógica de las órdenes de compra y venta, asegura que las transacciones sean ejecutadas correctamente y mantiene el libro de órdenes del intercambio.

- FixedSupplyToken.sol:

Este contrato define un token con un suministro fijo, siguiendo el estándar ERC-20. Se encarga de la creación y gestión de tokens con un suministro inicial definido, y proporciona funciones para transferir tokens entre usuarios, verificar balances y otras operaciones básicas de tokens.

- Migrations.sol:

Este contrato es parte del proceso de migración de contratos en Truffle. Ayuda a gestionar las actualizaciones y despliegues del contrato en la blockchain, asegurando que las migraciones se realicen de manera ordenada y que el estado de las migraciones sea correctamente registrado.

- Owned.sol:

Este contrato define un mecanismo de propiedad simple. Permite asignar un propietario al contrato y proporciona funciones para transferir la propiedad. Este es un patrón común utilizado para restringir ciertas operaciones sensibles a solo el propietario del contrato, añadiendo una capa de seguridad.

6.3 Configuración del proyecto y pruebas.

En este apartado se va a configurar el proyecto para poder hacer pruebas en una blockchain local con el uso de la herramienta Ganache. A continuación, se realizarán pruebas, ejecutando ciertas funciones de los contratos y viendo los cambios en el estado de la blockchain. Se recuerda que en el ANEXO II: Funcionamiento de un DEX hay una explicación detallada sobre el funcionamiento de un DEX.

En la figura 13 se muestra el estado inicial de la blockchain de pruebas de Ethereum después de desplegar los contratos inteligentes a través de la interfaz gráfica de Ganache.

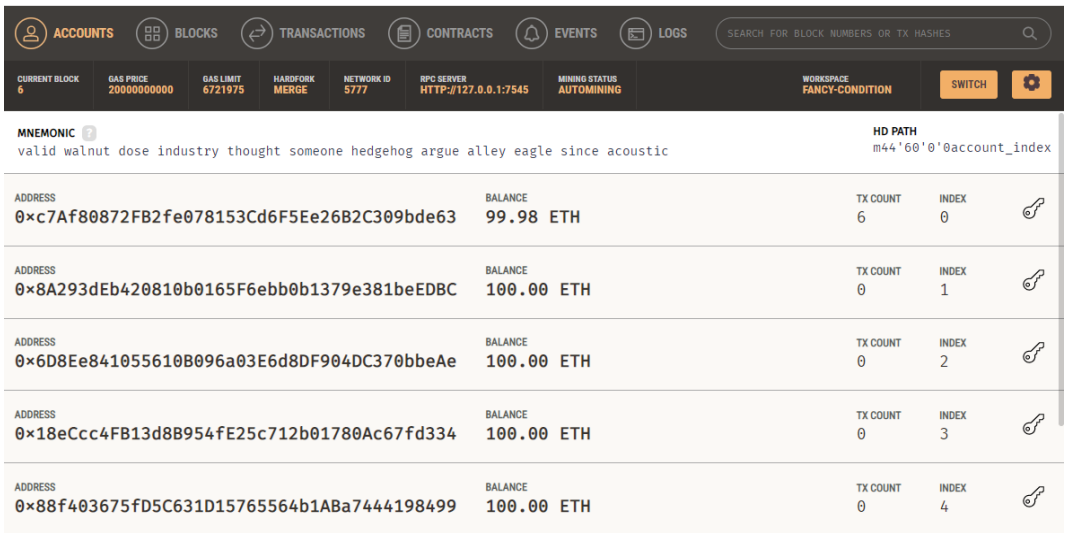


Figura 13: Estado inicial de la blockchain de pruebas

Como se puede ver, la blockchain cuenta con varias cuentas, cada una con un balance inicial de 100 Ether (la cuenta con índice 0 tiene menos Ether debido a que es la cuenta principal del protocolo y se le ha cobrado gas para hacer la migración de los contratos).

En la figura 14 se muestra la primera prueba ejecutada sobre la blockchain a través de la consola de Truffle.

```
truffle(ganache)> const Exchange = await artifacts.require('Exchange');
undefined
truffle(ganache)> const exchangeInstance = await Exchange.deployed();
undefined
undefined
truffle(ganache)> const FixedSupplyToken = await artifacts.require('FixedSupplyToken');
undefined
truffle(ganache)> const tokenInstance = await FixedSupplyToken.deployed();
undefined
truffle(ganache)> const accounts = await web3.eth.getAccounts();
undefined
truffle(ganache)> let balance = await tokenInstance.balanceOf(accounts[0]);
undefined
truffle(ganache)> console.log("Saldo inicial:", balance.toString());
Saldo inicial: 1000000
undefined
```

Figura 14: Prueba 1, obtener balance

En primer lugar, se ha instanciado el contrato *Exchange* y el *FixedSupplyToken*. Se recuerda que este segundo contrato inicializa un token ficticio e introduce un saldo inicial de 1.000.000 de FIXED en la cuenta con índice 0 (accounts[0]). A continuación, se muestra el saldo de esta cuenta y se comprueba que se ha inicializado correctamente el entorno.

Para poder hacer pruebas, transferimos 1.000 tokens a la cuenta accounts[1] mediante la instrucción:

$$await\ tokenInstance.transfer(accounts[1], 1000);$$

Después hay que configurar el DEX. En primer lugar, se debe dar de alta el token en cuestión, que en este caso tendrá el nombre "FIXED". Esto se logra mediante la siguiente instrucción:

$$await\ exchangeInstance.addToken("FIXED", tokenInstance.address, \{\ from: accounts[0]\ \});$$

Seguidamente hay que aprobar los tokens para que puedan ser usados por el DEX mediante la instrucción:

$$await\ tokenInstance.approve(exchangeInstance.address, 1000, \{\ from: accounts[1]\ \});$$

En esta instrucción, se aprueba una cantidad de tokens para que el contrato de intercambio (exchangeInstance) pueda gastar en nombre del usuario (accounts[1]). En este caso son 1.000 FIXED.

En la figura 15 se muestra el bloque en el que se encuentra la transacción que ha aprobado los tokens.

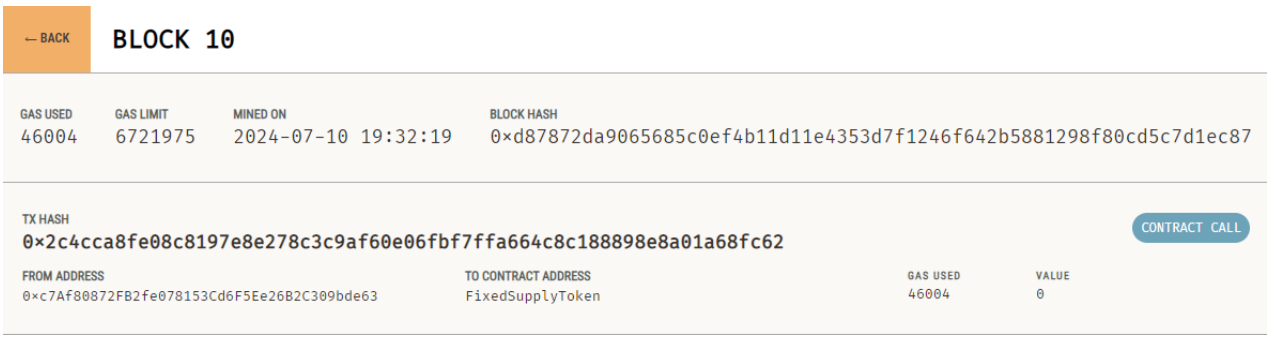


Figura 15: Prueba 2, bloque con aprobado de tokens

A continuación, en la figura 16 se muestra cómo se depositan los 1.000 FIXED del usuario accounts[1] en el DEX.

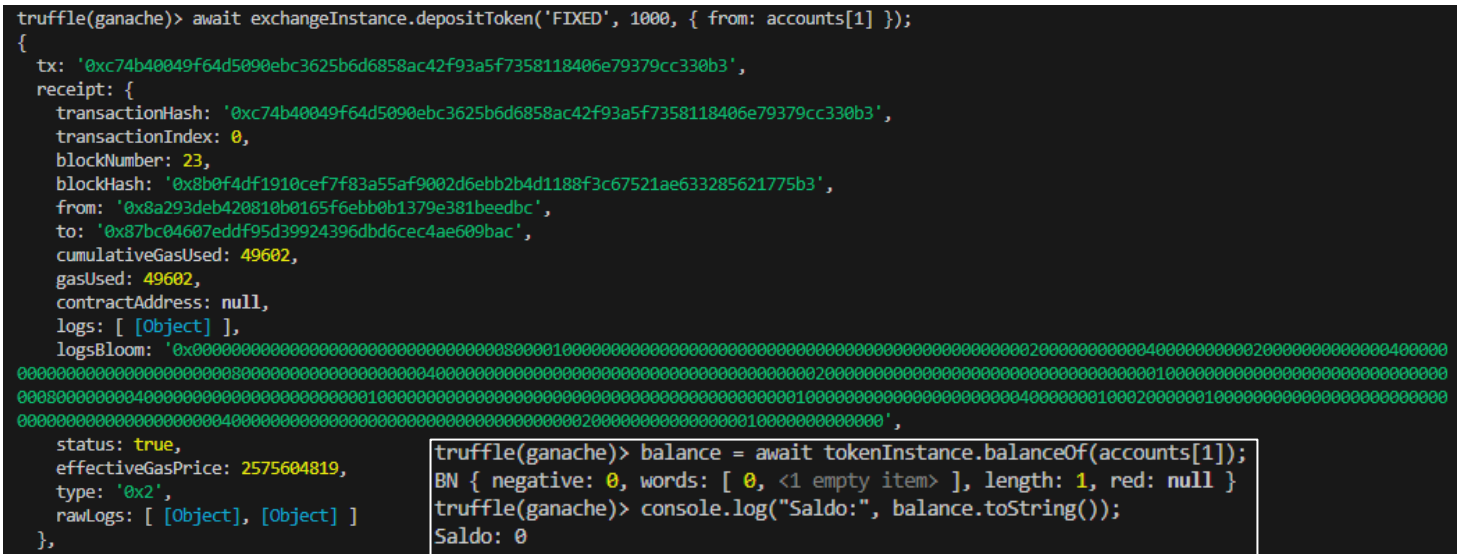


Figura 16: Prueba 3, depósito de tokens en el DEX

También se comprueba en esta figura 16 que el saldo de accounts[1] queda a 0 después de depositar los tokens en el DEX.

A continuación se va a llevar a cabo una operación de compra/venta de tokens FIXED de accounts[1] a accounts[2]. Para llevar a cabo esta operación, es necesario seguir ciertos pasos:

1. Hay que poner a la venta los tokens depositados desde accounts[1]:

$$await\ exchangeInstance.sellToken('FIXED', 10000, 1000, \{\ from: accounts[1]\ \});$$

Esta instrucción pone 1.000 tokens a la venta a un precio de 10.000 Wei cada uno.

2. Después se debe depositar algo de Ether en el DEX desde accounts[2] para poder hacer el pago de los tokens.

$$await\ exchangeInstance.depositEther(\{from: accounts[2], value: web3.utils.toWei('3', "ether")\});$$

En esta instrucción se transfieren 3 Ether (pasados a la unidad Wei) desde accounts[2] al DEX.

En la figura 17 se puede ver el balance de accounts[2] y del contrato DEX después del depósito de Ether.

Figura 17: Prueba 4, balance accounts[2] y DEX

Como se puede ver, accounts[2] cuenta con 97 Ether mientras que el DEX tiene 3 Ether en su balance.

3. Después se compran los tokens FIXED en accounts[2] mediante la instrucción:

$$await\ exchangeInstance.buyToken('FIXED', 10000, 1000, \{\ from: accounts[2]\ \});$$

En la figura 18 se puede ver el bloque 27 que confirma la transacción de compra/venta del token FIXED.

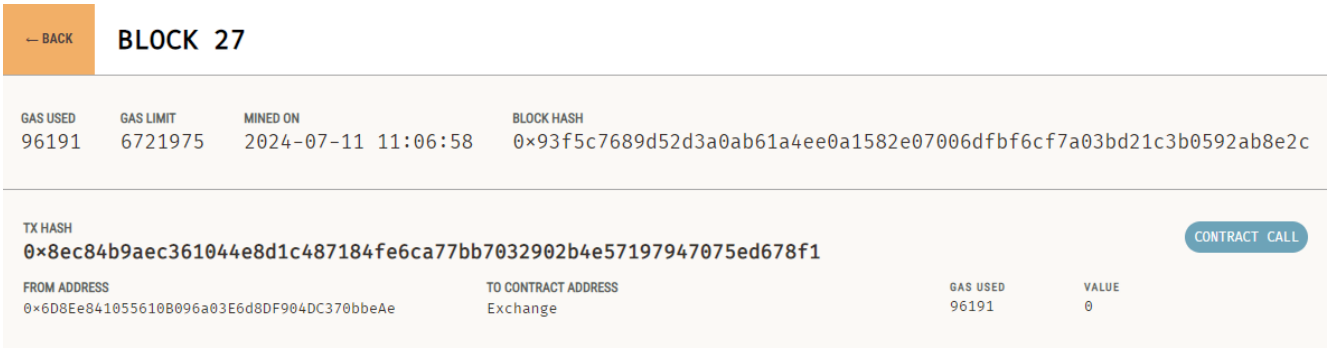


Figura 18: Prueba 4, bloque de transacción de compra/venta

4. Finalmente se retira el saldo del DEX a las cuentas correspondientes, tanto de FIXED como de Ether.

En la figura 19 se muestra el saldo correspondiente que se va a retirar a cada cuenta.

```
truffle(ganache)> const FIXEDaccounts2 = await exchangeInstance.getBalance('FIXED', { from: accounts[2] });
undefined
truffle(ganache)> const etheraccounts1 = await exchangeInstance.getEthBalanceInWei({ from: accounts[1] });
undefined
truffle(ganache)> console.log("Saldo:", etheraccounts1.toString());
Saldo: 10000000
undefined
truffle(ganache)> const etheraccounts2 = await exchangeInstance.getEthBalanceInWei({ from: accounts[2] });
undefined
truffle(ganache)> console.log("Saldo:", etheraccounts2.toString());
Saldo: 2999999999990000000
```

Figura 19: Prueba 4, balance disponible en el DEX

El balance de FIXED será de 1.000 para accounts[2] y 0 para accounts[1], ya que ha sido el importe intercambiado en la transacción.

En cuanto al Ether, tal y como se ve en la figura 19, el saldo de accounts[1] es de 10.000.000 Wei, ya que esta cuenta ha vendido 1.000 FIXED a 10.000 Wei cada uno. En cambio, para accounts[2] tenemos 2999999999990000000 Wei que equivalen a los 3 Ether iniciales menos los 10.000.000 Wei pagados para comprar los FIXED.

Para extraer los saldos correspondientes ejecutamos las siguientes instrucciones:

- Extrae FIXED a accounts[2]:

$$await\ exchangeInstance.withdrawToken("FIXED", FIXEDaccounts2, \{\ from: accounts[2]\ \});$$

- Extrae Ether a accounts[2]:

$$await\ exchangeInstance.withdrawEther(etheraccounts2, \{\ from: accounts[2]\ \});$$

- Extraer Ether a accounts[1]:

$$await\ exchangeInstance.withdrawEther(etheraccounts1, \{\ from: accounts[1]\ \});$$

En la figura 20 se puede ver el balance final de FIXED para accounts[1] y accounts[2].

```
truffle(ganache)> let balanceFinal1 = await tokenInstance.balanceOf(accounts[1]);
undefined
truffle(ganache)> let balanceFinal2 = await tokenInstance.balanceOf(accounts[2]);
undefined
truffle(ganache)> console.log("Saldo final 1:", balanceFinal1.toString());
Saldo final 1: 0
undefined
truffle(ganache)> console.log("Saldo final 2:", balanceFinal2.toString());
Saldo final 2: 1000
```

Figura 20: Prueba 4, comprobación de balances

6.4 Introducción de vulnerabilidades.

En este apartado se van a introducir vulnerabilidades intencionadamente en el contrato *Exchange.sol*. Esto permitirá evaluar la efectividad de las herramientas de análisis en la detección de problemas de seguridad. Las vulnerabilidades que se añaden son *reentrancy*, *overflow/underflow* y falta de control de acceso.

- Reentrancy:

Para introducir la vulnerabilidad de *reentrancy*, se modifica la función *withdrawEther* para permitir una llamada recursiva antes de actualizar el estado. En la figura 21 se muestran los cambios efectuados en el código.

Antes:

```
function withdrawEther(uint amountInWei) public {
    // Balance sufficient to withdraw check
    require(balanceEthForAddress[msg.sender] - amountInWei >= 0);
    // Overflow check
    require(balanceEthForAddress[msg.sender] - amountInWei <= balanceEthForAddress[msg.sender]);
    // Deduct from balance and transfer the withdrawal amount
    balanceEthForAddress[msg.sender] -= amountInWei;
    msg.sender.transfer(amountInWei);
    WithdrawalEth(msg.sender, amountInWei, now);
}
```

Después:

```
function withdrawEther(uint amountInWei) public {
    // Balance sufficient to withdraw check
    require(balanceEthForAddress[msg.sender] - amountInWei >= 0);
    // Overflow check
    require(balanceEthForAddress[msg.sender] - amountInWei <= balanceEthForAddress[msg.sender]);

    // Llamada externa antes de actualizar balance (REENTRANCY)
    msg.sender.transfer(amountInWei);

    // Actualiza balance
    balanceEthForAddress[msg.sender] -= amountInWei;
    WithdrawalEth(msg.sender, amountInWei, now);
}
```

Figura 21: Código de withdrawEther con vulnerabilidad reentrancy

- Overflow y underflow:

Para introducir la vulnerabilidad de *overflow* y *underflow*, se modifican las operaciones aritméticas en la función *depositToken* y *withdrawToken*. En la figura 22 se muestran los cambios efectuados en el código.

Antes:

```
function depositToken(string symbolName, uint amountTokens) public {
    uint8 symbolNameIndex = getSymbolIndexOrThrow(symbolName);
    // Check the Token Contract Address is initialised and not an uninitialised address(0) aka "0x0"
    require(tokens[symbolNameIndex].tokenContract != address(0));

    ERC20Interface token = ERC20Interface(tokens[symbolNameIndex].tokenContract);

    // Transfer an amount to this DEX from the calling address
    require(token.transferFrom(msg.sender, address(this), amountTokens) == true);
    // Overflow check
    require(tokenBalanceForAddress[msg.sender][symbolNameIndex] + amountTokens >= tokenBalanceForAddress[msg.sender][symbolNameIndex]);
    // Credit the DEX token balance for the callinging address with the transferred amount
    tokenBalanceForAddress[msg.sender][symbolNameIndex] += amountTokens;
    DepositForTokenReceived(msg.sender, symbolNameIndex, amountTokens, now);
}

function withdrawToken(string symbolName, uint amountTokens) public {
    uint8 symbolNameIndex = getSymbolIndexOrThrow(symbolName);
    require(tokens[symbolNameIndex].tokenContract != address(0));

    ERC20Interface token = ERC20Interface(tokens[symbolNameIndex].tokenContract);

    // Check sufficient balance to withdraw requested amount
    require(tokenBalanceForAddress[msg.sender][symbolNameIndex] - amountTokens >= 0);
    // Overflow check to ensure future balance less than or equal to the current balance after deducting the withdrawn amount
    require(tokenBalanceForAddress[msg.sender][symbolNameIndex] - amountTokens <= tokenBalanceForAddress[msg.sender][symbolNameIndex]);
    // Deduct amount requested to be withdrawing from the DEX Token Balance
    tokenBalanceForAddress[msg.sender][symbolNameIndex] -= amountTokens;
    // Check that the `transfer` function of the Token Contract returns true
    require(token.transfer(msg.sender, amountTokens) == true);
    WithdrawalToken(msg.sender, symbolNameIndex, amountTokens, now);
}
```

Después:

```
function depositToken(string symbolName, uint amountTokens) public {
    uint8 symbolNameIndex = getSymbolIndexOrThrow(symbolName);
    // Check the Token Contract Address is initialised and not an uninitialised address(0) aka "0x0"
    require(tokens[symbolNameIndex].tokenContract != address(0));

    ERC20Interface token = ERC20Interface(tokens[symbolNameIndex].tokenContract);

    // Transfer an amount to this DEX from the calling address
    require(token.transferFrom(msg.sender, address(this), amountTokens) == true);

    //NO se comprueba el overflow

    // Credit the DEX token balance for the callinging address with the transferred amount
    tokenBalanceForAddress[msg.sender][symbolNameIndex] += amountTokens;
    DepositForTokenReceived(msg.sender, symbolNameIndex, amountTokens, now);
}

function withdrawToken(string symbolName, uint amountTokens) public {
    uint8 symbolNameIndex = getSymbolIndexOrThrow(symbolName);
    require(tokens[symbolNameIndex].tokenContract != address(0));

    ERC20Interface token = ERC20Interface(tokens[symbolNameIndex].tokenContract);

    // Check sufficient balance to withdraw requested amount
    require(tokenBalanceForAddress[msg.sender][symbolNameIndex] - amountTokens >= 0);

    // NO se comprueba el underflow

    // Deduct amount requested to be withdrawing from the DEX Token Balance
    tokenBalanceForAddress[msg.sender][symbolNameIndex] -= amountTokens;
    // Check that the `transfer` function of the Token Contract returns true
    require(token.transfer(msg.sender, amountTokens) == true);
    WithdrawalToken(msg.sender, symbolNameIndex, amountTokens, now);
}
```

Figura 22: Código de depositToken y withdrawToken con vulnerabilidad overflow y underflow

- Falta de control de acceso:

Para introducir la falta de control de acceso, en la figura 23 se eliminan los modificadores de control de acceso de la función *addToken*.

Antes:

```
function addToken(string symbolName, address erc20TokenAddress) public onlyowner {
    // Modifier checks if caller is Admin "owner" of the Contract otherwise return early
    require(!hasToken(symbolName));
    symbolNameIndex++;
    tokens[symbolNameIndex].symbolName = symbolName;
    tokens[symbolNameIndex].tokenContract = erc20TokenAddress;
    TokenAddedToSystem(symbolNameIndex, symbolName, now);
}
```

Después:

```
function addToken(string symbolName, address erc20TokenAddress) public {
    require(!hasToken(symbolName));
    symbolNameIndex++;
    tokens[symbolNameIndex].symbolName = symbolName;
    tokens[symbolNameIndex].tokenContract = erc20TokenAddress;
    TokenAddedToSystem(symbolNameIndex, symbolName, now);
}
```

Figura 23: Código de addToken con vulnerabilidad de falta de control de acceso

Con estos cambios, el contrato *Exchange.sol* ahora contiene vulnerabilidades específicas que serán detectadas mediante las herramientas de análisis de seguridad Slither y Oyente.

6.5 Auditoría con Slither.

En esta sección, se realiza una auditoría de los contratos del proyecto DEX utilizando Slither, después de haber introducido intencionadamente vulnerabilidades en el código.

Slither es una herramienta de análisis estático, ya que examina el código fuente sin ejecutarlo para identificar patrones y posibles vulnerabilidades y errores en el código. Aunque Slither es potente para identificar varias vulnerabilidades, no detecta desbordamientos (overflow) y subdesbordamientos (underflow) de enteros en Solidity.

Algunas de las vulnerabilidades que detecta Slither incluyen: *reentrancy*, dependencias a bibliotecas no seguras y uso incorrecto de estándares como el ERC-20.

Slither requiere tener instalado Python y funciona a través de comandos ejecutados en la terminal.

Para poder empezar con la auditoría, primero hay que compilar el código de los contratos ejecutando la instrucción *"truffle compile"* en el directorio raíz del proyecto. A continuación, ejecutamos el siguiente comando para obtener un resumen de las vulnerabilidades de los contratos:

slither . –print human-summary

Este comando muestra una tabla con un resumen para poder tener una primera impresión sobre el estado de los contratos. En la siguiente figura 24 se puede ver el resultado del comando.

```
INFO:Printers:
Compiled with Truffle
Total number of contracts in source files: 5
Source lines of code (SLOC) in source files: 534
Number of  assembly lines: 0
Number of optimization issues: 10
Number of informational issues: 24
Number of low issues: 9
Number of medium issues: 5
Number of high issues: 0

ERCs: ERC20

+-------------------+---------------+-------+---------------------+--------------+----------------------+
| Name              | # functions   | ERCS  |           ERC20 info | Complex code |             Features |
+-------------------+---------------+-------+---------------------+--------------+----------------------+
| FixedSupplyToken  |            15 | ERC20 |           No Minting |           No |                      |
|                   |               |       | Approve Race Cond.  |              |                      |
|                   |               |       |                     |              |                      |
| Exchange          |            21 |       |                     |          Yes |          Receive ETH |
|                   |               |       |                     |              |             Send ETH |
|                   |               |       |                     |              | Tokens interaction   |
| Migrations        |             3 |       |                     |           No |                      |
+-------------------+---------------+-------+---------------------+--------------+----------------------+
```

Figura 24: Tabla resumen del análisis de Slither

Lo primero que se muestra en el resumen es un desglose de los distintos problemas o vulnerabilidades detectadas, separándolas por su criticidad. El análisis se va a centrar en las vulnerabilidades, dejando de lado las recomendaciones de optimización e informaciones (*optimization issues* e *informational issues*). La tabla muestra alguna información complementaria como el número de funciones en cada contrato y algunas funcionalidades de estas.

Después de generar el resumen, se va a generar el análisis completo para observar más en detalle las vulnerabilidades que hay en los contratos. Esto se logra ejecutando el comando *"slither ."*.

En la siguiente figura 25 se muestran las vulnerabilidades catalogadas como *medium issues* por la herramienta Slither.

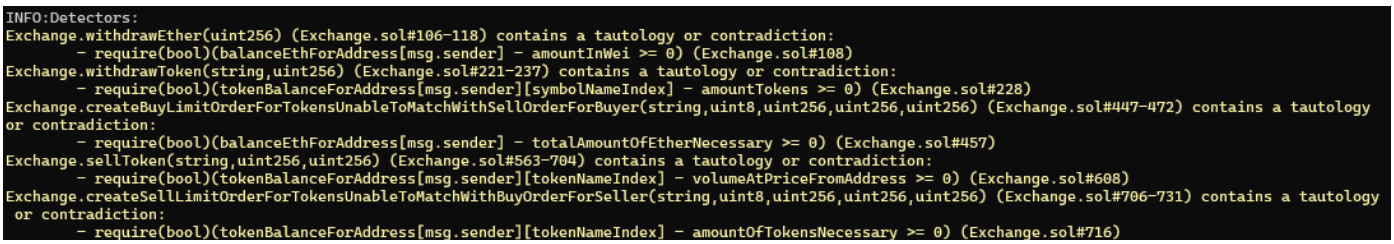


Figura 25: Medium issues en el proyecto DEX

Las 5 vulnerabilidades *medium* que muestra el resumen de la figura 24 se corresponden con estos errores de código llamados tautologías o contradicciones. Una tautología se refiere a una condición que siempre es verdadera, independientemente de las variables involucradas. Una contradicción es una condición que siempre es falsa. Estos patrones pueden indicar complejidad innecesaria o errores lógicos en los contratos inteligentes escritos en Solidity.

En este caso en concreto, es inútil comprobar las condiciones >= 0, ya que se está comprobando que variables del tipo *uint* sean positivas cuando siempre es así por su definición.

En cuanto a los *low issues*, en la figura 26 se muestra la traza del comando que contiene algunos de estos *low issues*.

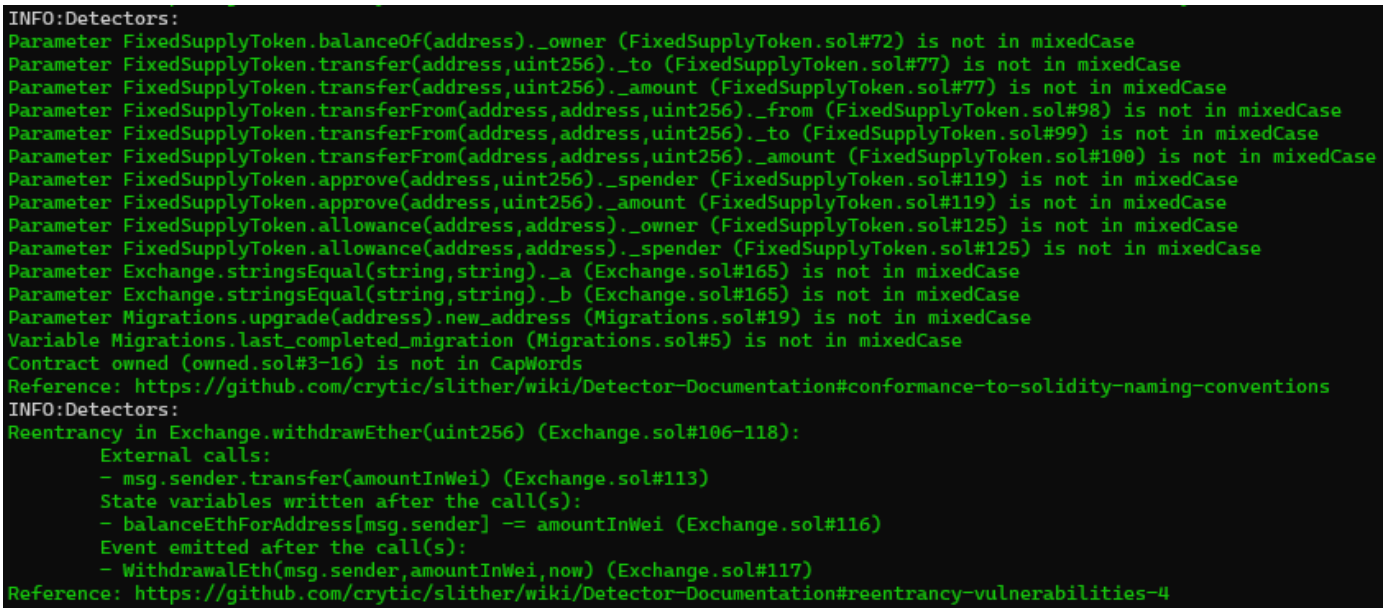


Figura 26: Low issues en el proyecto DEX

Tal y como se puede ver en la segunda entrada de *INFO:Detectors* del terminal, se ha detectado la vulnerabilidad *reentrancy* introducida a propósito en el apartado anterior en la función *withdrawEther*. Es interesante atender al detalle de que al final de cada vulnerabilidad encontrada, se introduce un enlace al repositorio oficial de Github del proyecto en el que se explica en qué consiste cada vulnerabilidad y como corregirla.

Como conclusión, el análisis realizado con Slither sobre el proyecto DEX ha detectado una vulnerabilidad de *reentrancy*, lo que subraya la importancia de medidas de seguridad robustas contra ataques que exploten esta debilidad; sin embargo, Slither no ha logrado identificar los problemas de *overflow* y *underflow* ni la falta de control de acceso, lo que resalta las limitaciones de la herramienta y la necesidad de complementar el análisis con otras técnicas y herramientas de seguridad para una evaluación exhaustiva y efectiva de la seguridad de los contratos.

6.6 Auditoría con Oyente.

En esta sección, se realiza una auditoría de los contratos del proyecto DEX utilizando Oyente.

Oyente examina el bytecode de los contratos inteligentes en lugar del código fuente. Este enfoque le permite identificar patrones de posibles vulnerabilidades y errores sin ejecutar el código.

Para simplificar la instalación y el uso, se empleará un contenedor Docker para ejecutar Oyente. Oyente requiere Python para su funcionamiento y se utiliza mediante comandos en la terminal.

Para poner en marcha el contenedor, se debe descargar la imagen que contiene el entorno y el software pertinente para utilizar Oyente. Esta se encuentra en el Github oficial del proyecto.

$$docker\ pull\ luongnguyen/oyente$$

A continuación, se arranca el contenedor mediante el siguiente comando:

$$docker\ run\ -v\ "C:\backslash Users\backslash Ubicacion\backslash Del\backslash dex:/project"\ -it\ luongnguyen/oyente\ /bin/bash$$

Este comando arranca un contenedor con la imagen de Oyente y monta un volumen, es decir, crea una conexión entre la carpeta donde está ubicado el proyecto DEX en el equipo y la carpeta */project* del contenedor. La creación del volumen es indispensable para poder ejecutar el software de Oyente sobre el proyecto.

Antes de ejecutar el analizador Oyente, ha sido necesario instalar unas versiones específicas del compilador de Solidity y de la EVM en el contenedor para poder trabajar con el código del proyecto DEX, ya que no coincidía con las versiones instaladas en el equipo. Las versiones que se han utilizado son la 0.4.19 para el compilador y la 1.7.3 para la EVM.

Una vez el entorno está configurado correctamente, ya se puede analizar el código del proyecto con Oyente mediante el siguiente comando:

$$python\ oyente.py\ -s\ ../../project/contracts/Exchange.sol$$

Aunque solamente se hace referencia al contrato *Exchange.sol,* al tener este dependencias con los otros contratos contenidos en la carpeta *contracts/* se analizarán todos ellos. En la siguiente figura 27 se muestra el análisis del contrato *Exchange.sol* por la herramienta Oyente.

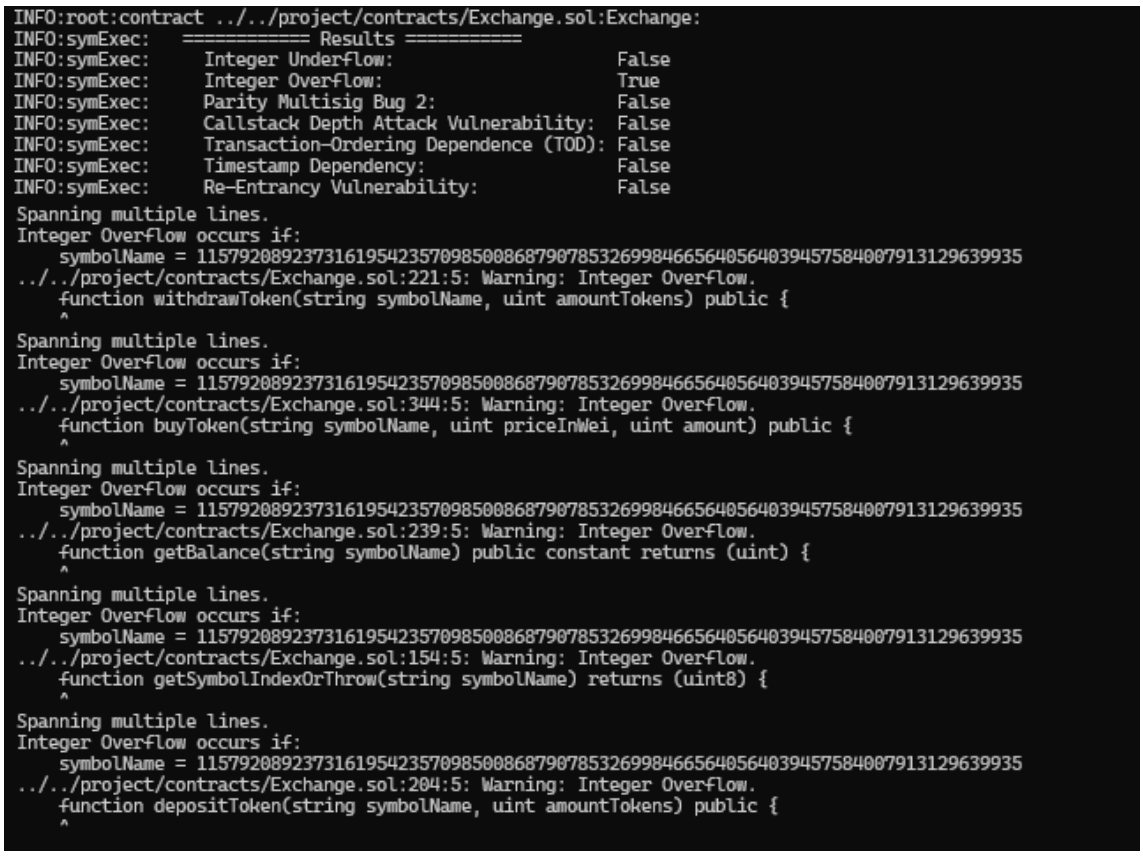


Figura 27: Análisis de Oyente de Exchange.sol

Tal y como se puede ver en la tabla resumen inicial, la herramienta de análisis no ha sido capaz de detectar la mayoría de las vulnerabilidades introducidas en el contrato. Únicamente se ha detectado el *overflow*, de modo que tanto el *underflow* como las vulnerabilidades de *reentrancy* y la de falta de control de acceso no han sido detectadas por Oyente.

Además de detectar el *overflow* introducido a propósito en la función *depositToken,* Oyente ha localizado otras vulnerabilidades *overflow* en muchas otras funciones como *buyToken* o *getBalance.* Además, se ha detectado *overflow* en la función *withdrawToken* cuando en esta no hay una sola operación de suma o multiplicación que pueda originar este desbordamiento aritmético.

En la función *buyToken* sí se comprueba que no haya *overflow* antes de ejecutar las operaciones aritméticas. Esto se logra mediante una línea de código que sirve como comprobación. Es probable que Oyente se base en otros criterios como la utilización de librerías especializadas como *SafeMath* para detectar que no hay vulnerabilidades de

overflow. En cuanto a las otras funciones detectadas, sí que hay peligro de que se produzca un desbordamiento aritmético por un error del desarrollador del contrato, ya que no se comprueba mediante código o librerías especializadas que no exista *overflow*.

En la figura 28 se puede ver el análisis del contrato *FixedSupplyToken.sol.* Este contrato es interesante de analizar porque también contiene varias funciones útiles para el funcionamiento del DEX y sigue el estándar ERC-20. Los contratos *Migrations.sol* y *owned.sol* también han sido analizados, pero no contienen vulnerabilidades ni arrojan información relevante.

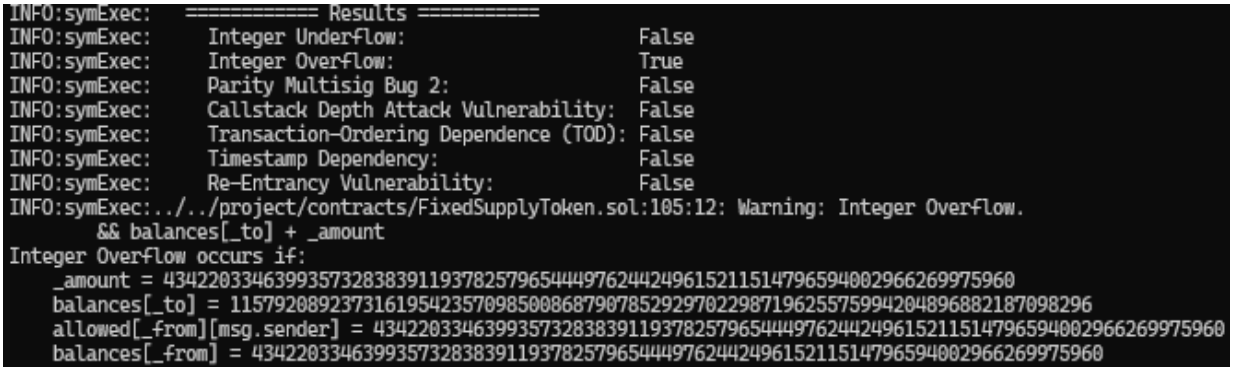


Figura 28: Análisis de Oyente de FixedSupplyToken.sol

Oyente ha detectado en este contrato una vulnerabilidad de *overflow* en la función *transferFrom.*

En resumen, Oyente ha sido capaz de detectar las vulnerabilidades de *overflow* pero no las de *reentrancy*, *underflow* o las de falta de control de acceso. Este hecho es alarmante, ya que en la propia tabla resumen que ofrece Oyente al hacer el análisis se muestra que este es capaz de detectar tanto *reentrancy* como *underflow*. La no detección de estas vulnerabilidades puede deberse a la complejidad y estructura del contrato, que podría haber superado las capacidades actuales de la herramienta para identificar todas las posibles vulnerabilidades. También es posible que Oyente no sea actualmente una herramienta fiable para la detección de vulnerabilidades en smart contrats y esté en desarrollo de versiones mejores y más completas.

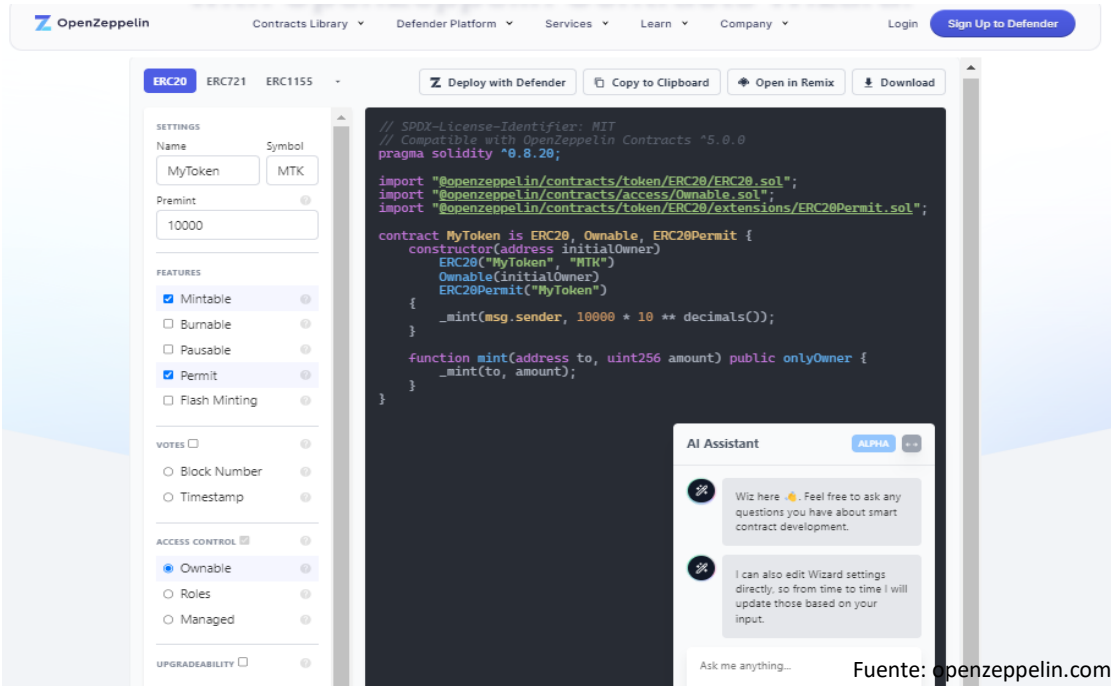
7 Mejores prácticas para el desarrollo seguro de smart contracts.

El desarrollo seguro de smart contracts es fundamental para evitar vulnerabilidades que puedan ser explotadas por atacantes y causar pérdidas millonarias como las estudiadas en apartados anteriores. A partir del trabajo de investigación realizado en este TFG, se proponen una serie de mejores prácticas que los desarrolladores deben seguir para garantizar la seguridad y la integridad de sus contratos inteligentes. Estas propuestas surgen del análisis de incidentes anteriores y de una investigación adicional en webs y artículos especializados en el desarrollo de contratos inteligentes. La implementación de estas prácticas no solo reduce la posibilidad de ataques, sino que también fortalece la confianza en las aplicaciones descentralizadas y en la infraestructura blockchain en general.

7.1 Uso de librerías confiables.

El uso de librerías confiables es una de las primeras líneas de defensa contra las vulnerabilidades en los smart contracts. Bibliotecas como las de OpenZeppelin ofrecen implementaciones seguras y bien auditadas de patrones comunes y estándares de contratos, como ERC-20 y ERC-721. Estas librerías han sido ampliamente probadas y auditadas por la comunidad, lo que reduce significativamente el riesgo de errores o vulnerabilidades. Además, su uso permite a los desarrolladores centrarse en la lógica específica de su contrato sin reinventar la rueda, aprovechando las soluciones robustas ya disponibles.

OppenZeppelin ofrece también una herramienta en su página web que permite confeccionar la estructura básica de un contrato inteligente de forma segura y a medida. En la figura 29 se muestra la interfaz gráfica de esta herramienta.



Fuente: openzeppelin.com

Figura 29: Interfaz web de confección de smart contracts de OpenZeppelin

A través del menú de herramientas de la izquierda se va añadiendo código y bibliotecas según la elección de características que haga el desarrollador.

7.2 Realización de auditorías regulares.

Tal y como se ha comprobado en apartados anteriores de este TFG, las auditorías regulares son esenciales para mantener la seguridad de los smart contracts. Estas auditorías deben ser realizadas por el mismo equipo y por terceros independientes y expertos en seguridad de blockchain. Durante una auditoría, se revisa el código fuente del contrato para identificar posibles vulnerabilidades, errores lógicos y malas prácticas de programación. Las auditorías ayudan a descubrir problemas que pueden no ser evidentes durante el desarrollo y las pruebas internas. Además, es recomendable realizar auditorías periódicas, especialmente después de implementar cambios significativos en el contrato, para asegurarse de que las modificaciones no introduzcan nuevas vulnerabilidades.

7.3 Implementación de tests.

Los tests son cruciales para garantizar que los contratos inteligentes funcionen según lo esperado y no contengan errores. Esto incluye tests unitarios, que verifican la funcionalidad de componentes individuales del contrato, y tests de integración, que aseguran que diferentes componentes interactúan correctamente entre sí. Además, las pruebas de seguridad deben simular ataques comunes, como *reentrancy* y desbordamiento de enteros, para verificar que el contrato es resistente a estos *exploits* (Ortega, 2023). Las herramientas de pruebas automatizadas pueden facilitar el desarrollo y ejecución de estas pruebas, proporcionando un entorno controlado para la evaluación del contrato. Algunas de estas herramientas incluyen el framework *Truffle* ya utilizado durante este TFG u otros frameworks y herramientas como *HardHat* o *Brownie.* Estas herramientas no solo mejoran la eficiencia del proceso de pruebas, sino que también ayudan a identificar y corregir errores más rápidamente, contribuyendo así a la creación de contratos inteligentes más seguros y confiables.

7.4 Control de acceso estricto.

El control de acceso estricto es esencial para prevenir el uso no autorizado de funciones críticas en los contratos inteligentes. Esto se logra mediante la implementación de modificadores de acceso, que restringen quién puede ejecutar ciertas funciones. Por ejemplo, el uso del patrón "onlyOwner" asegura que solo el propietario del contrato pueda ejecutar funciones administrativas. Es importante definir roles y permisos claros y asegurarse de que las funciones críticas solo puedan ser ejecutadas por entidades autorizadas. Además, es recomendable revisar y actualizar regularmente las políticas de control de acceso para adaptarse a cambios en el contrato o en la estructura organizativa.

7.5 Manejo seguro de fondos.

El manejo seguro de fondos es una de las preocupaciones más importantes en el desarrollo de contratos inteligentes. Los desarrolladores deben asegurarse de que los fondos sean manejados de manera segura para evitar pérdidas debido a errores o ataques. Esto incluye el uso de técnicas como el patrón "pull over push", que recomienda que los usuarios retiren fondos en lugar de recibir transferencias automáticas, reduciendo así el riesgo de ataques de *reentrancy* (Alonso, 2022). Además, es crucial implementar verificaciones adecuadas antes de transferir fondos y asegurar que las funciones que manejan fondos no sean vulnerables a desbordamientos y subdesbordamientos de enteros.

7.6 Documentación y comentarios.

La documentación y los comentarios claros son fundamentales para el desarrollo seguro de smart contracts. La documentación debe detallar la funcionalidad de cada función, las dependencias del contrato, y los riesgos conocidos. Los comentarios en el código deben explicar la lógica y el propósito de las secciones de código complejas. Esto no solo facilita el mantenimiento y la actualización del contrato, sino que también ayuda a los auditores y otros desarrolladores a entender el contrato y detectar posibles problemas. La buena documentación es una práctica de seguridad en sí misma, ya que proporciona una base clara y comprensible para la revisión y la mejora continua del contrato (Ortega, 2023).

8 Conclusiones.

A continuación, se exponen las conclusiones que se pueden extraer del presente TFG y se reflexiona sobre el cumplimiento de los objetivos e hitos definidos en el primer apartado de este documento.

El análisis detallado de los ataques a *The DAO, Parity Wallet* y *Poly Network* ha proporcionado valiosas lecciones sobre las consecuencias de las fallas de seguridad y la importancia de abordar y prevenir vulnerabilidades en el contexto de los contratos inteligentes. Es importante recalcar que solo en estos tres ataques se comprometieron más de 800 millones de dólares en criptomonedas (valoradas en el momento de los ataques).

A partir de los casos estudiados y de consultas en portales y webs reputadas se ha logrado examinar y categorizar las principales vulnerabilidades presentes en los contratos inteligentes, tales como *reentrancy*, desbordamientos y subdesbordamientos de enteros, y falta de control de acceso. Esta categorización ha permitido una comprensión más profunda de las áreas críticas que requieren atención durante el desarrollo y la auditoría de contratos inteligentes. Además, se han detallado vulnerabilidades que no tienen que ver directamente con el código de los contratos pero que tienen una importancia vital en el ecosistema de la blockchain, como puede ser el *phishing* o los ataques DoS.

Se han utilizado las herramientas de análisis estático Slither y Oyente para auditar un proyecto con varios contratos inteligentes. Sin embargo, estas herramientas no han sido capaces de detectar todas las vulnerabilidades existentes en el código. Mientras que Slither ha detectado únicamente las vulnerabilidades de *reentrancy*, Oyente solo ha detectado *overflows* de enteros. Estas fallas en la detección de vulnerabilidades acentúan más la importancia de construir un código seguro desde los estados más

tempranos del desarrollo. Además, el auditar los contratos con más de un software o incluso mediante una auditoría especializada por un tercero es más que recomendable.

A partir de los casos estudiados y la investigación en internet se ha propuesto un conjunto de mejores prácticas para el desarrollo seguro de contratos inteligentes. Estas prácticas incluyen el uso de bibliotecas confiables, la realización de auditorías regulares, la implementación de pruebas exhaustivas y la importancia de documentar los proyectos. La adopción de estas prácticas puede mejorar significativamente la seguridad de los contratos inteligentes y prevenir vulnerabilidades comunes.

De este modo, dando respuesta al objetivo principal de este TFG, a raíz del contenido del mismo se han identificado y explicado las principales vulnerabilidades en el desarrollo de contratos inteligentes. Además, se han propuesto correcciones para las mismas y se ha llevado a cabo una auditoría de estos contratos para comprobar la robustez y confianza de las herramientas especializadas para las auditorías.

Para concluir, la seguridad en el desarrollo de software, tanto en contratos inteligentes como en otros ámbitos, es de vital importancia debido a las graves consecuencias que pueden derivarse de fallos en el código. Las vulnerabilidades en los contratos inteligentes pueden llevar a pérdidas financieras significativas, erosionar la confianza de los usuarios y afectar negativamente la adopción de tecnologías emergentes. En el futuro, es esencial seguir mejorando las prácticas de desarrollo seguro y realizar auditorías exhaustivas para detectar y corregir posibles fallos antes de que sean explotados. Solo mediante un enfoque proactivo y riguroso en la seguridad del código podremos asegurar un ecosistema tecnológico confiable y robusto.

A nivel personal, este TFG ha sido una experiencia enriquecedora que me ha permitido profundizar en el desarrollo de smart contracts, una tecnología en la que tengo un gran interés y a la que llevo varios meses dedicando tiempo para aprender. A través de este proyecto, he logrado consolidar conocimientos adquiridos previamente y he enfrentado nuevos desafíos como la utilización de nuevos *frameworks* como Truffle o la instalación y utilización de las herramientas de auditoría. Además, a través de este TFG he podido ampliar mi percepción sobre la importancia de la seguridad y las buenas prácticas de desarrollo en el código, hecho que me será útil tanto en mis proyectos personales como en mi carrera profesional.

8.1 Relación con asignaturas cursadas.

Este TFG se presenta como una aglomeración de conocimientos adquiridos durante el grado en Ingeniería Informática al completo. Sin embargo, hay algunas asignaturas que tienen una influencia mayor en la realización del presente documento. Entre ellas, destaca Ingeniería de Software, donde se adquirieron las bases para el diseño y

desarrollo de sistemas complejos, aplicando principios como la modularidad, la reutilización de código y las metodologías ágiles, que han sido cruciales en la organización y estructuración del código de los smart contracts.

Otra asignatura clave ha sido Gestión de Proyectos, que proporcionó las herramientas necesarias para planificar y gestionar el desarrollo de este TFG de manera eficiente. Gracias a lo aprendido, se pudo establecer un cronograma realista, asegurando que cada etapa del proyecto se completara dentro de los plazos previstos.

Por otro lado, Bases de Datos y Sistemas de Información ha sido esencial en la comprensión y manejo de la información en los sistemas blockchain. Aunque las bases de datos tradicionales difieren en gran medida de las estructuras de datos en una blockchain, los principios de gestión de datos y la importancia de la integridad y seguridad han sido directamente aplicables.

Finalmente, la asignatura Concurrencia y Sistemas Distribuidos ha desempeñado un papel crucial, al proporcionar una sólida base en los conocimientos sobre la comunicación entre sistemas distribuidos. Estos conocimientos han sido fundamentales para entender cómo funcionan las transacciones simultáneas y la coordinación entre nodos en un entorno blockchain. Además, esta comprensión ha sido clave para garantizar que los contratos inteligentes operen de manera eficiente y segura en un sistema descentralizado.

Como caso especial de este TFG y de mi situación de estudiante de Doble Grado, han sido de gran ayuda algunas de las asignaturas cursadas en el grado de Administración y Dirección de Empresas para comprender conceptos financieros que están estrechamente ligados a muchas de las aplicaciones de la tecnología blockchain y de los contratos inteligentes. Principalmente y como representante de las asignaturas relacionadas con las finanzas destacan Dirección Financiera y Economía Financiera.

En conjunto, el conocimiento adquirido en estas asignaturas ha sido invaluable para la realización de este proyecto, demostrando que el aprendizaje acumulado durante la carrera ha sido determinante para abordar con éxito el desarrollo y la auditoría de smart contracts, incluso cuando se trata de una tecnología relativamente nueva como lo es la blockchain.

8.2 Trabajos futuros.

El presente TFG ha abordado el análisis de vulnerabilidades en contratos inteligentes y la auditoría de los mismos utilizando herramientas especializadas. Sin embargo, debido a limitaciones de tiempo y recursos, no se han podido explorar todas las áreas de interés relacionadas con este tema. Por lo tanto, existen varias líneas de investigación y desarrollo que podrían considerarse como una continuación natural de este TFG.

- Uno de los proyectos futuros más evidentes sería la ampliación del análisis a otras redes blockchain y tipos de contratos inteligentes. Este trabajo se ha centrado principalmente en la red de Ethereum, que es la más utilizada para contratos inteligentes. Sin embargo, existen otras plataformas como Binance Smart Chain, Solana, y Polkadot que también soportan contratos inteligentes y tienen sus propias particularidades y vulnerabilidades. Un análisis comparativo de la seguridad entre estas redes podría proporcionar una visión más amplia y generalizable sobre las mejores prácticas en el desarrollo de contratos inteligentes.

- Otro proyecto derivado podría ser la implementación y evaluación de técnicas avanzadas de auditoría de contratos inteligentes. Este TFG ha servido para conocer y probar ciertas herramientas de análisis estático para auditar los contratos inteligentes. Sin embargo, hay muchas otras herramientas que efectúan otro tipo de auditorías dinámicas en contratos desplegados en *testnets* (blockchains globales de prueba). Sin embargo, estas herramientas son costosas y el desplegar estos contratos incluso en estas *testnets* puede suponer algunas complicaciones y costes.

- Además, sería interesante desarrollar un proyecto enfocado en la automatización de auditorías de seguridad en entornos de desarrollo continuos (CI/CD). Aunque este TFG ha explorado herramientas de auditoría manuales y semiautomáticas, la integración de estas herramientas en un flujo de trabajo de desarrollo continuo podría mejorar significativamente la seguridad de los contratos inteligentes desde sus etapas iniciales de desarrollo.

BIBLIOGRAFÍA

Alonso, C. (2022, 12 mayo). *Blockchain & SmartContracts: Patrones y buenas prácticas de seguridad.* Recuperado 22 de julio de 2024, de:

https://www.elladodelmal.com/2022/05/blockchain-smartcontracts-patrones-y.html

Bastardo, J. (2017, 7 noviembre). *Cerca de $150 millones en ETH están congelados tras el accidental «suicidio» de Parity Wallet.* CriptoNoticias. Recuperado 5 de julio de 2024, de:

https://www.criptonoticias.com/tecnologia/cerca-150-millones-congelados-accidental-suicidio-parity-wallet/

Chainalysis (2024, 29 febrero). *Funds Stolen from Crypto Platforms Fall More Than 50% in 2023, but Hacking Remains a Significant Threat as Number of Incidents Rises.* Chainalysis. Recuperado 23 de julio de 2024, de:

https://www.chainalysis.com/blog/crypto-hacking-stolen-funds-2024

Gagliardoni, T. (2021, 12 agosto). *The Poly Network hack explained.* Kudelski Security Research.

https://research.kudelskisecurity.com/2021/08/12/the-poly-network-hack-explained/

Gemini (2023, octubre). *The DAO: What Was the DAO Hack*? Gemini. Recuperado 4 de julio de 2024, de:

https://www.gemini.com/cryptopedia/the-dao-hack-makerdao#section-the-dao-hack

IBM. (s. f.). *¿Qué es Blockchain?* IBM. Recuperado 3 de julio de 2024, de:

https://www.ibm.com/es-es/topics/blockchain

Ledger. (2023, 21 agosto). *Smart Contract Functions - How to Spot a Scam.* Ledger.

https://www.ledger.com/academy/smart-contract-functions-essential-red-flags-to-know-about

Levi, D. & Lipton, B. (2018, 26 mayo). *An Introduction to Smart Contracts and Their Potential and Inherent Limitations.* The Harvard Law School Forum On Corporate Governance.

https://corpgov.law.harvard.edu/2018/05/26/an-introduction-to-smart-contracts-and-their-potential-and-inherent-limitations/

Lin, S. (2023). *Proof of Work vs. Proof of Stake in Cryptocurrency.* Highlights In Science, Engineering And Technology, 39, 953-961.

https://doi.org/10.54097/hset.v39i.6683

Ltfschoen (s. f.) *Proyecto DEX.* Github.

https://github.com/ltfschoen/dex

Medium. (2023, 1 julio). *PARITY Wallet Hack: What, When and How?* Medium.

https://medium.com/@web3author/parity-wallet-hack-demystified-all-you-need-to-know-91b8dcb5b81

Ortega, J. M. (2023, 31 octubre). *Seguridad y auditorías en smart contracts.* Codemotion Magazine.

https://www.codemotion.com/magazine/es/backend-es/blockchain-es/seguridad-y-auditorias-en-smart-contracts/

Roopika J. (2020). *Blockchain Technology: History, Concepts, and Applications.* International Research Journal of Engineering and Technology. Recuperado 1 de julio de 2024, de:

https://www.irjet.net/archives/V7/i10/IRJET-V7I10109.pdf

Russoniello, A. (2021, 7 septiembre). *Piscinas de liquidez*. BITacora del Russo [Blog]. Recuperado 29 de julio de 2024, de:

https://bitacoradelrusso.blogspot.com/2021/09/piscinas-de-liquidez-liquidity-pool.html

Santos, F. (2018). *The DAO: A million dollar lesson in blockchain governance.* TALLINN UNIVERSITY OF TECHNOLOGY.

https://digikogu.taltech.ee/en/Download/42e7380f-9f99-4d98-bd90-64f6a971ff7c/ThedaomiljonidollarippetundBlockchainivali.pdf

Sayeed S. Marco-Gisbert H. and Caira T. (2020). *Smart Contract: Attacks and Protections.* IEEE Access, vol. 8, pp. 24416-24427

https://ieeexplore.ieee.org/abstract/document/8976179/

Simões, C. (2022, 23 marzo). *¿Qué son los bloques en la tecnología Blockchain?* Blog ITDO Recuperado 1 de julio de 2024, de:

https://www.itdo.com/blog/que-son-los-bloques-en-la-tecnologia-blockchain/

SlowMist. (2021, 10 agosto). *The root cause of Poly Network being hacked.* Medium.

https://slowmist.medium.com/the-root-cause-of-poly-network-being-hacked-ec2ee1b0c68f

SlowMist. (2023, 30 marzo). *Common Vulnerabilities in Solidity: Denial of Service (DOS).* SlowMist. Recuperado 17 de julio de 2024, de:

https://www.slowmist.com/articles/solidity-security/Common-Vulnerabilities-in-Solidity-Denial-of-Service-DOS.html

Wejdene, H. & Marios, F. (2024). *Vulnerabilities of smart contracts and mitigation schemes: A Comprehensive Survey.* York University.

https://www.researchgate.net/publication/379726984_Vulnerabilities_of_smart_contracts_and_mitigation_schemes_A_Comprehensive_Survey

Weston, G. (2024, 9 febrero). *Arithmetic Underflow and Overflow Vulnerabilities In Smart Contracts.* 101 Blockchains. Recuperado 10 de julio de 2024, de:

https://101blockchains.com/underflow-and-overflow-vulnerabilities-in-smart-contracts/

Zhang, Y. and Liu, D. (2022, octubre*). Toward Vulnerability Detection for Ethereum Smart Contracts Using Graph-Matching Network.* School of Cyber Science and Engineering, Southeast University.

https://www.mdpi.com/1999-5903/14/11/326

ANEXO I: Relación del trabajo con los Objetivos de Desarrollo Sostenible de la Agenda 2030.

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenibles	**Alto**	**Medio**	**Bajo**	**No Procede**
ODS 1. **Fin de la pobreza.**				**X**
ODS 2. **Hambre cero.**				**X**
ODS 3. **Salud y bienestar.**				**X**
ODS 4. **Educación de calidad.**				**X**
ODS 5. **Igualdad de género.**				**X**
ODS 6. **Agua limpia y saneamiento.**				**X**
ODS 7. **Energía asequible y no contaminante.**				**X**
ODS 8. **Trabajo decente y crecimiento económico.**	**X**			
ODS 9. **Industria, innovación e infraestructuras.**	**X**			
ODS 10. **Reducción de las desigualdades**.				**X**
ODS 11. **Ciudades y comunidades sostenibles**.			**X**	
ODS 12. **Producción y consumo responsables.**				**X**
ODS 13. **Acción por el clima.**				**X**
ODS 14. **Vida submarina.**				**X**
ODS 15. **Vida de ecosistemas terrestres.**				**X**
ODS 16. **Paz, justicia e instituciones sólidas.**				**X**
ODS 17. **Alianzas para lograr objetivos.**				**X**

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

Este TFG contribuye en un alto grado al **ODS 8** promoviendo el crecimiento económico sostenible a través de la mejora en la seguridad de los contratos inteligentes, que son una parte esencial del ecosistema blockchain y de la economía digital emergente. Al identificar y mitigar vulnerabilidades en los contratos inteligentes, se facilita un entorno más seguro para las transacciones y aplicaciones financieras, lo que fomenta la confianza de los usuarios y las empresas. Esto, a su vez, impulsa el crecimiento de nuevas oportunidades de negocio y empleo en el sector tecnológico y financiero, creando un entorno propicio para el trabajo decente y el desarrollo económico inclusivo.

El TFG también se alinea fuertemente con el **ODS 9** al centrarse en la innovación y el desarrollo de infraestructuras seguras y resilientes en el ámbito de la tecnología blockchain. Al proporcionar un análisis exhaustivo de las vulnerabilidades de los contratos inteligentes y proponer mejores prácticas y herramientas de auditoría para su mitigación, este trabajo fomenta la creación de infraestructuras tecnológicas robustas y seguras. Además, impulsa la innovación en el desarrollo de nuevas aplicaciones descentralizadas (dApps) y servicios basados en blockchain, contribuyendo significativamente a la industrialización sostenible y la innovación tecnológica en múltiples sectores.

En menor medida, este TFG contribuye al **ODS 11** al mejorar la seguridad y confiabilidad de las aplicaciones basadas en blockchain que pueden ser utilizadas en entornos urbanos para servicios públicos, como la gestión de identidad, el registro de propiedades y la transparencia en la administración pública. Al garantizar que estos sistemas sean seguros y estén libres de vulnerabilidades, se promueve el desarrollo de ciudades más inteligentes y sostenibles, donde los servicios son eficientes y accesibles, y la confianza de los ciudadanos en las tecnologías emergentes se fortalece.

ANEXO II: Funcionamiento de un DEX

En el ámbito de la tecnología blockchain, los Exchanges Descentralizados (DEX) representan una evolución significativa respecto a los exchanges centralizados tradicionales. Un DEX es una plataforma de intercambio de criptomonedas que opera sin una autoridad central que custodie los fondos de los usuarios. En su lugar, los intercambios se realizan directamente entre los usuarios a través de contratos inteligentes en una blockchain.

La creación y popularización de los DEX han surgido como respuesta a varios problemas asociados con los exchanges centralizados:

- En un exchange centralizado, los fondos de los usuarios se almacenan en wallets controladas por la entidad que gestiona el exchange. Este control centralizado los convierte en un objetivo atractivo para los hackers. Históricamente, varios exchanges centralizados han sido víctimas de hackeos que han resultado en la pérdida de grandes sumas de dinero de los usuarios.
- Los exchanges centralizados requieren que los usuarios confíen en la entidad que opera el exchange, no solo para la seguridad de los fondos sino también para la integridad de las transacciones. Esta falta de transparencia y la necesidad de confiar en un tercero pueden ser problemáticas, especialmente si la entidad central no es confiable o no opera de manera transparente.

Los DEX presentan una serie de beneficios que los hacen atractivos tanto para usuarios como para desarrolladores:

- En un DEX, los usuarios mantienen el control total sobre sus fondos en todo momento. Las transacciones se ejecutan directamente entre las wallets de los usuarios, eliminando la necesidad de confiar en un tercero para custodiar los fondos.
- Debido a que no hay un punto centralizado de falla, los DEX son menos susceptibles a los hackeos masivos. Aunque los contratos inteligentes que operan el DEX pueden tener vulnerabilidades, estas son generalmente menores en comparación con los riesgos asociados con la custodia centralizada de fondos.
- Las operaciones en un DEX se realizan en la blockchain, lo que proporciona un nivel de transparencia que no es posible en un exchange centralizado. Cada transacción puede ser verificada de manera independiente por cualquier usuario de la red.

El funcionamiento de un DEX pasa por las siguientes fases:

1. Añadir un Token

El primer paso para interactuar con un DEX es asegurarse de que el token que se desea intercambiar esté disponible en la plataforma. En la mayoría de los casos, los DEX permiten a los usuarios añadir nuevos tokens siguiendo un proceso específico:

- Verificación del Token: El usuario debe verificar que el token cumple con los estándares necesarios (por ejemplo, ERC-20 en Ethereum).
- Registro del Token: Algunos DEX requieren que los tokens sean registrados en su plataforma. Este registro implica proporcionar detalles como el contrato del token, su símbolo, y su número de decimales.
- Listar el Token: Una vez registrado, el token se lista en el DEX, permitiendo a otros usuarios comerciar con él.

2. Aprobar Fondos

Para operar en un DEX, los usuarios deben aprobar la transferencia de fondos desde sus wallets hacia el contrato inteligente del DEX:

- Aprobación del Contrato: El usuario debe interactuar con el contrato inteligente del token para aprobar que el DEX pueda transferir una cantidad específica de tokens en su nombre.
- Transacción de Aprobación: Esta aprobación se realiza mediante una transacción en la blockchain que el usuario debe confirmar. Esta transacción requiere el pago de tarifas de gas.

3. Transferencia de Tokens

La transferencia de tokens en un DEX implica mover los activos desde la wallet del usuario hacia el contrato del DEX:

- Iniciación de la Transferencia: El usuario inicia una transacción para transferir los tokens aprobados al contrato inteligente del DEX.
- Ejecución del Contrato: El contrato inteligente del DEX ejecuta la transferencia de los tokens. Esta operación es transparente y puede ser verificada en la blockchain.

4. Realización de Intercambios

El intercambio de tokens es el núcleo de la funcionalidad de un DEX. A continuación, se describen los pasos involucrados en este proceso:

- Selección del Par de Intercambio: El usuario selecciona el par de tokens que desea intercambiar (por ejemplo, ETH/DAI).
- Determinación del Precio: El DEX utiliza un mecanismo de precio determinado por el mercado o por algoritmos específicos para establecer el precio de intercambio.
- Ejecutar el Intercambio: El usuario inicia la transacción de intercambio, especificando la cantidad de tokens que desea intercambiar. El DEX calcula la cantidad de tokens que el usuario recibirá a cambio, teniendo en cuenta el precio y las tarifas de transacción.

- Confirmación de la Transacción: El usuario confirma la transacción, que se envía a la blockchain para su validación. Esta transacción también incurre en tarifas de gas.
- Transferencia de Tokens: Una vez confirmada, el DEX realiza la transferencia de los tokens intercambiados a la wallet del usuario. El contrato inteligente se asegura de que los tokens se transfieran correctamente y de manera segura.

5. Liquidez y Pools de Liquidez

La liquidez es esencial para el funcionamiento eficiente de un DEX. Los pools de liquidez proporcionan los fondos necesarios para facilitar los intercambios:

- Creación de Pools de Liquidez: Los usuarios pueden depositar pares de tokens en un pool de liquidez. Estos tokens están disponibles para otros usuarios que desean intercambiarlos.
- Provisión de Liquidez: Los proveedores de liquidez reciben incentivos, como una parte de las tarifas de transacción, por mantener fondos en los pools de liquidez.
- Extracción de Liquidez: Los proveedores pueden retirar sus fondos del pool en cualquier momento, junto con las recompensas obtenidas.

En la figura 30 se muestra un ejemplo de funcionamiento de un pool de liquidez en el que un proveedor deposita 1.000 dólares en Ether y 1.000 dólares en DAI y cobra una comisión cuando otro usuario (Trader) efectúa una transacción.

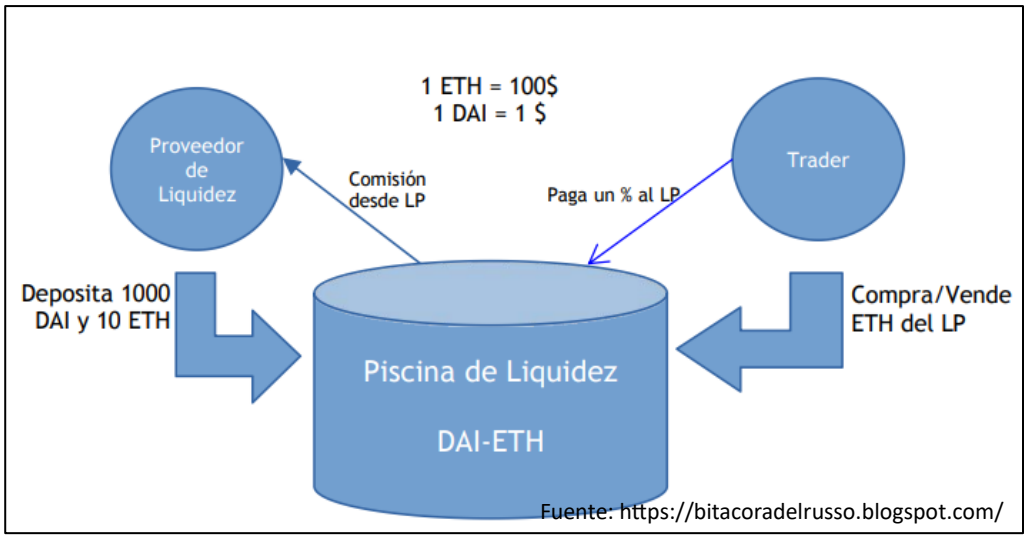


Figura 30: Diagrama de una liquidity pool

Algunos de los DEX más conocidos y utilizados en el ecosistema blockchain incluyen: *Uniswap* y *Sushiswap* principalmente para la red de Ethereum y *Polkadex* para la red de Polkadot, entre muchos otros.