Bastionado de un contrato inteligente

Alumno: Joan Oliva Costas
Curso: 2021-2022
Director: Jorge Miguel Moneo

Resumen del trabajo

Los contratos inteligentes son unos contratos que se pueden ejecutar automáticamente sin requerir de terceras partes que verifiquen su cumplimiento. El proyecto de Ethereum proporciona las herramientas para desarrollar y desplegar contratos inteligentes en la *blockchain.* Su lenguaje de desarrollo estrella es *Solidity*, que permite realizar programaciones complejas de alto nivel. Los contratos inteligentes son susceptibles a vulnerabilidades que pueden suponer pérdidas de criptomonedas de forma directa para las víctimas. En este proyecto se crea un contrato inteligente que implementa una aseguradora descentralizada, para posteriormente analizar sus vulnerabilidades, documentarlas, y corregirlas. Para detectar posibles amenazas, se utilizan herramientas de auditoría de contratos inteligentes. Las herramientas utilizadas se ponen a prueba en función de sus resultados. Este proyecto produce dos versiones de una aseguradora descentralizada. Una de ellas vulnerable a posibles ataques. La otra, completamente bastionada con las medidas de seguridad más actuales.

Abstract

Smart contracts are self-executing contracts that do not require a third party for their verification. The Ethereum project provides the required tools for developing and deploying smart contracts on the blockchain. They are usually programed in Solidity, a high-level programming language. Smart Contracts are vulnerable to attacks that can suppose direct losses in cryptocurrencies. In this project we create a smart contract that implements a decentralized insurance company. We analyze, document, and correct vulnerabilities present in the contract. We try and test different auditing tools for smart contracts. These tools are later compared and classified. This project creates two versions of the contract. The first version is vulnerable. The second version is fully secured.

Índice de contenidos

Introducción

Presentación del problema

Los contratos inteligentes, también llamados *Smart Contracts*, son un tipo de contratos que se ejecutan de forma automática sin requerir de terceros para asegurar su cumplimiento.

Los contratos inteligentes permiten que dos o más partes, ya sean personas u organizaciones, cumplan las condiciones pactadas en una negociación sin necesidad de un organismo intermediario. Esto se consigue mediante la ejecución de un **código** que representa el flujo del contrato.

Por ejemplo, un símil de contrato inteligente es una máquina expendedora. El usuario deposita las monedas en la máquina, y después de seleccionar el producto concreto, la máquina lo proporciona sin necesidad de intervención alguna. [8] [17] [2]

Para que un contrato inteligente pueda ser desplegado, se puede apoyar en la tecnología ***blockchain***.

La *blockchain* es una libreta de registros que actúa como una base de datos descentralizada, dónde cada inserción en esa libreta es verificada por múltiples nodos en la red. El estado pasado de la *blockchain* no se puede modificar, por lo que todos sus registros son inmutables. [16]

El código de un contrato inteligente está escrito en un lenguaje de programación concreto, y se ejecuta en cada nodo de la red descentralizada. El estado final de la ejecución del programa es escrito en la *blockchain* y por lo tanto debe ser igual para todos los nodos. Esto garantiza que no pueda ser alterado.

Aunque estos programas no puedan ser alterados, sí que son susceptibles a fallos de programación, que introducen **vulnerabilidades** al sistema.

Los contratos inteligentes son capaces de almacenar datos en su interior, y, por lo tanto, pueden ser utilizados como depósitos de dinero. Es muy común encontrar contratos con **propósitos financieros** como por ejemplo realizar transacciones, recaudar financiación para proyectos, apostar sobre mercados…

Esto convierte a los contratos inteligentes en especialmente atractivos para los atacantes que desean adquirir esas ingentes cantidades de dinero, ya que pueden explotar las vulnerabilidades introducidas por los programadores y **transferir el dinero** almacenado a otras direcciones distintas a las intencionadas. [26]

Este proyecto se centra en el bastionar un contrato inteligente para así demostrar cómo se pueden solucionar dichas vulnerabilidades. En concreto, el proyecto trabajará con los contratos inteligentes dentro de la red de Ethereum, utilizando un lenguaje de programación de contratos de alto nivel.

Revisión de la literatura

Los contratos inteligentes se definen por primera vez en el 1996 por Nick Szabo. Su idea era la de poder ejecutar las partes contractuales sin necesidad de un organismo de intermediación. Esta idea inicial quedó en reposo durante más de diez años, ya que la tecnología en esa época no permitía la implementación de los contratos inteligentes. [25]

Con la aparición de **Bitcoin** en el 2009, nace el lenguaje de programación de contratos nombrado "*Script*". Este permite realizar transacciones entre direcciones siempre bajo cierta lógica programable. La mayor desventaja de este lenguaje es que no fue concebido como un sistema Turing completo, por lo que carece de funcionalidades de programación avanzadas (como las iteraciones o la recursividad). [19][23]

En el año 2015, de mano de Vitalik Buterin, se lanza la primera versión de la red **Ethereum**. Ethereum se inspira en Bitcoin y trata de ser una red descentralizada que ofrece más capacidades que Bitcoin. Una de las características principales de la red de Ethereum es la capacidad de crear contratos inteligentes completamente programables y con capacidades de interacción dinámica. [30]

Así, un contrato inteligente en Ethereum se forma de una colección de código y de unos datos almacenados que representan su estado. El contrato actúa como un participante más en la red descentralizada. Esto significa que los contratos son capaces de almacenar dinero (Ether) y pueden realizar transacciones a través de la red. Así mismo, pueden ofrecer funciones al público que toda la red puede invocar, incluso otros contratos desplegados. [8][10]

La red de Ethereum ofrece lenguajes de programación más completos que los scripts de Bitcoin. Estos son **Solidity** y **Vyper**. Se tratan de lenguajes de programación de alto nivel que permiten definir funciones, variables, operaciones, y trabajar con las direcciones de la red de Ethereum (enviar o recibir dinero, para simplificar), entre muchas otras capacidades. Básicamente, permiten implementar cualquier programa. Estos lenguajes se compilan a un lenguaje "ensamblador" que luego puede ser ejecutado en la máquina virtual de Ethereum (EVM). [14]

Con la popularización de los contratos inteligentes en la red de Ethereum nacen los primeros intentos de ataques informáticos. Los contratos, al almacenar dinero en formato de criptomonedas, y en contener código invocable, son el principal foco de atención de los atacantes.

El código de un contrato inteligente es visible para todo el mundo (debido a su naturaleza descentralizada), por lo que su seguridad es crítica.

Entre los ataques más famosos que ha sufrido la red de Ethereum se encuentra el ataque DAO. Una DAO es una organización descentralizada que almacena fondos. Uno de estos fondos sufrió un robo de sus 150 millones de dólares que había recaudado en un contrato inteligente. El impacto del ataque fue tal que se restauró el estado de la *blockchain* de Ethereum a una versión anterior. [21]

Este ataque, entre muchos otros, representan el potencial que pueden llegar a tener las vulnerabilidades introducidas por falta de conocimiento durante la programación de los contratos inteligentes. Estas vulnerabilidades pueden presentar una gran variedad de tipos, por lo que se tiene que tomar consciencia de ellas durante su programación.

Objetivos

Este proyecto tiene como principal objetivo **bastionar un *Smart Contract*.** Es decir, convertir un contrato inteligente inicialmente vulnerable hacia una versión más segura.

Este objetivo principal se puede descomponer en varios subobjetivos:

1. Comprender las tecnologías involucradas y las vulnerabilidades existentes en los contratos inteligentes.
2. Crear un contrato inteligente que pueda ser desplegado en una red descentralizada que opera bajo una *blockchain*.
3. Estudiar las vulnerabilidades que presenta el contrato.
4. Modificar la versión inicial del contrato y demostrar que las vulnerabilidades se hayan corregido.

Metodología

Descripción del método

La metodología del proyecto es propia y se ajusta al problema a resolver (*adhoc)*. Esta metodología consta de un análisis inicial, ya que se desea ampliar el conocimiento respecto el ecosistema que engloba los contratos inteligentes. Se proponen las siguientes fases para alcanzar los objetivos:

1. Análisis del estado del arte.
2. Diseño del software a implementar.
3. Implementación y despliegue del software.
4. Auditoría de seguridad.
5. Bastionado del software.

En la fase del **análisis del estado del arte**, se realiza una investigación más intensiva acerca de las tecnologías y las vulnerabilidades involucradas en el proyecto. Esta parte es imprescindible para ampliar la visión del proyecto y poder realizar las siguientes fases con más seguridad y conocimiento. El resultado de esta fase es un documento explicativo con un resumen de la información recopilada.

En una segunda fase de **diseño del software**, se planifica el esquema general del contrato a implementar, así como los posibles agentes que estén involucrados en su ejecución. El resultado de esta fase es la explicación del plan de implementación en alto nivel.

En la tercera fase de **implementación**, se escribe el código del contrato y de los posibles agentes que estén involucrados. El resultado es un entorno que puede ejecutar el contrato y dónde los distintos agentes puedan interactuar con él.

En la cuarta fase de **auditoría**, se realiza un estudio de las vulnerabilidades reales que presenta el software implementado, ya sea mediante herramientas de auditoría existentes en el mercado o mediante la programación específica de situaciones de explotación. El resultado de esta fase es un listado de vulnerabilidades con su correspondiente detalle. También se pueden comparar las herramientas de auditoría existentes en el mercado.

Por último, en la fase de **bastionado del software**, se realizan las correcciones necesarias para mitigar o eliminar las vulnerabilidades detectadas en la fase anterior, verificando que ya no sean explotables. El resultado de esta fase es un nuevo código del contrato inteligente creado anteriormente, pero en una versión más segura.

Tareas

Las tareas para realizar están relacionadas con las fases de la metodología y se pueden desglosar de la siguiente forma:

1. Análisis del estado del arte.
    a. Estudio de la red descentralizada.
    b. Estudio de las tecnologías involucradas.
    c. Estudio de las vulnerabilidades conocidas en los contratos inteligentes.
2. Diseño del software a implementar.
    a. Análisis de requisitos.
    b. Diseño del contrato y de sus agentes.
    c. Diseño del entorno de despliegue.
3. Implementación del software.
    a. Implementación del contrato y sus agentes.
    b. Implementación de las pruebas.
    c. Despliegue del software.
4. Auditoría de seguridad.
    a. Auditoría con herramientas automáticas.
    b. Auditoría manual mediante la programación de agentes.
5. Bastionado del software.
    a. Implementación de la nueva versión del contrato.
    b. Ejecución de la auditoría para verificar los cambios.
6. Documentación del proyecto. Redacción de la memoria final.
7. Presentación del proyecto. Realizar y grabar la presentación del proyecto.
8. Defensa del proyecto.

Planificación temporal

La planificación temporal ajusta las dos primeras tareas a la segunda entrega del proyecto (PEC2), la tercera tarea y parte de la cuarta a la tercera entrega (PEC3), y parte de la cuarta, más la quinta y sexta tareas a la cuarta entrega del proyecto (PEC4). La presentación se ajusta a la quinta entrega (PEC5).

El cambio más importante respecto a la planificación propuesta por el calendario de la asignatura es la dedicación de tiempo de la cuarta entrega (PEC4) para el desarrollo del proyecto, ya que se estima que las tareas de implementación, auditoría y bastionado serán complejas y por lo tanto no es posible realizarlas en el marco de tiempo que ofrece la PEC3.

Para compensar este préstamo de tiempo, se reduce la redacción de la memoria final del proyecto a 2 semanas, ya que se estima que no será necesario dedicar un mes entero a la documentación del proyecto (gran parte del desarrollo ya produce mucho material explicativo que únicamente deberá ser resumido).

Por lo tanto, en la PEC3 del proyecto entregará como mínimo la implementación del software, pudiendo complementar dicha entrega con parte de la auditoría realizada.

La planificación temporal resulta en el siguiente calendario:

Dia inicio	Dia fin	Fase	Tareas
14/3	20/3	Estado del arte	1.a, 1.b
21/3	27/3	Estado del arte	1.c
28/3	3/4	Diseño del software	2.a, 2.b, 2.c, 2.d
4/4	14/4	Implementación del software	3.a, 3.b
15/4	20/4	Vacaciones	-
21/4	24/5	Implementación del software	3.c
25/4	15	Auditoría de seguridad	4.a
2/5	8/5	Auditoría de seguridad	4.b
9/5	15/5	Bastionado del software	5.a, 5.b
16/5	31/5	Redacción de la memoria	6
1/6	9/6	Presentación del proyecto	7
10/6	12/6	-	-
13/6	17/6	Defensa	8

Tabla 1 planificación temporal

Estado del arte

La red Ethereum y los Smart Contracts

Una ***blockchain*** es una cadena de bloques, enlazados unos a otros, cuyo estado se almacena de forma descentralizada en todos los nodos de la red. Una *blockchain* se puede entender como una libreta de registros inmutable, donde cada entrada representa una transacción. La única operación que se permite en una *blockchain* es añadir nuevas transacciones en la libreta.

Las transacciones se agrupan en **bloques**, y los bloques se enlazan entre sí mediante una cadena. Esta cadena representa todas las transacciones pasadas en la red.

Cada bloque nuevo es fruto de un proceso de minado, que recompensa a los mineros que han tenido éxito con una moneda virtual (o también llamada criptomoneda). Los nuevos bloques contienen un **hash** calculado a partir de todos los bloques anteriores existentes en la red y el bloque nuevo, por lo que la validez de la cadena de bloques siempre puede ser verificada. [16]

Los mineros **son recompensados** ya que tienen que resolver un problema computacionalmente complejo para poder añadir los bloques. El primero que lo resuelva es quién gana la recompensa. Los otros verificarán que el bloque sea correcto y que el proceso de minado sea correcto. Este paradigma se denomina *proof-of-work*.

En la práctica, se suelen formar bolsas de mineros que resuelven el problema de forma conjunta, y luego se reparten los beneficios.

En las redes tradicionales, como Bitcoin, el uso principal de la libreta de registros es almacenar las transacciones de la criptomoneda entre pares. Los mineros son los que alimentan al sistema con nuevas criptomonedas, ya que son recompensados por validar dichas transacciones. [19]

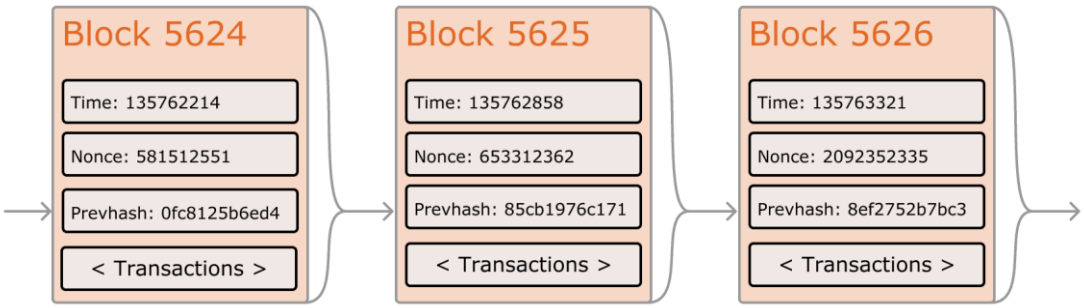

Ilustración 1 Estructura de una cadena de bloques. [10]

La red de **Ethereum** es una red descentralizada de código abierto. Su principal moneda de operación es el **Ether** (ETH). La red de Ethereum incorpora nuevas funcionalidades que la diferencian de su antecesora, la red de Bitcoin. Estas diferencias se detallan a continuación.

En primer lugar, la red de Ethereum opera bajo el combustible del **gas**, que es una unidad de impuesto mínimo para realizar transacciones y cálculos en la red. El gas se representa en la unidad **gwei** y siempre equivale a 10^{-9} ETH.

Por ejemplo, si se desea realizar una transacción del nodo A al B, el nodo A deberá pagar siempre una cantidad en concepto de **gas a los mineros** que validan esta transacción. El gas es un mecanismo inventado para evitar que la red se colapse por culpa de ataques de denegación de servicio, ya que aquellos participantes que intenten hacerlo deben pagar un alto precio en gas. [29]

En segundo lugar, a diferencia de las redes convencionales como Bitcoin, la red de Ethereum dispone de la capacidad de desplegar **contratos inteligentes**. Estos contratos contienen código ejecutable y pueden realizar transacciones de criptomonedas entre las distintas direcciones de la red. Los contratos se ejecutan de manera automática sin necesidad de que intervenga ningún particular.

La capacidad de ejecutar contratos se debe al sistema conocido como la *Ethereum Virtual Machine* (EVM). Esta máquina virtual ejecuta un conjunto de instrucciones simples en formato de EVM *bytecode*. Este código normalmente se obtiene al compilar un lenguaje de alto nivel, como *Solidity* o *Vyper*. La máquina virtual dispone de memoria virtual, que es almacenada en la misma *blockchain* de Ethereum.

A grandes rasgos, es un computador descentralizado que tiene memoria y puede ejecutar instrucciones, leyendo y escribiendo los datos de la memoria y realizando operaciones sobre estos datos con el objetivo final de enviar y recibir criptomonedas entre los participantes de la red.

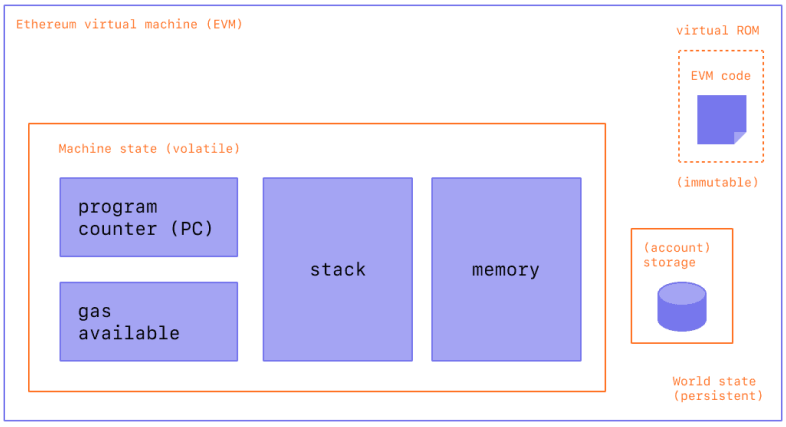

Ilustración 2 Estructura de la EVM. [7]

Los nodos de la red que participan en la ejecución del código son los mismos **mineros**. Estos se encargan de verificar mediante una prueba de consenso que el estado final de la máquina virtual es el mismo para la mayoría de los mineros. Al ejecutar el código, reciben una recompensa en formato de gas. Esto significa que desplegar y ejecutar un contrato conlleva un **coste** para el interesado.

En función del gas que el interesado ofrezca a los mineros, estos deciden si quieren o no ejecutar el contrato en el mismo momento o bien **posponerlo**. Cada bloque sólo puede almacenar un número finito de transacciones, por lo que los mineros deciden qué oferta es mejor incorporar en cada momento.

Todo este mecanismo consigue que la ejecución de los contratos sea segura, ya que un atacante debería disponer del 51% de la red para poder manipular los resultados de la *blockchain*. Si tuviera tal control de ETH, entonces su valor caería en picado.

En la red Ethereum, tanto los contratos como los usuarios tienen una **dirección asociada.** Esto significa que no hay distinción entre un contrato o un usuario cuando se envían o reciben criptomonedas. Esta dirección es única e identifica inequívocamente una cartera de criptomonedas.

Existen tres tipos de transacciones que las direcciones pueden realizar respecto a los contratos. Todas estas acciones quedarán registradas en la *blockchain*:

- Se puede publicar un contrato en la red.
- Se puede llamar a una función de un contrato existente en la red.
- Se puede transferir ETH hacia una dirección.

La **programación de los contratos inteligentes** se realiza en lenguajes de alto nivel, como ya se ha comentado. En función del lenguaje se pueden realizar abstracciones de mayor o menor nivel, aunque el código sea siempre compilado al *bytecode* de la EVM.

El código compilado tiene soporte nativo para enviar y recibir criptomonedas hacia direcciones, realizar operaciones matemáticas, almacenar datos en la memoria… Se comporta como un sistema Turing completo, por lo que puede implementar cualquier programa.

Cuando se despliega un contrato inteligente en la red, se invoca su **método constructor**, que es un método que sólo se puede invocar una sola vez. En el método constructor se deberían inicializar las variables y asignar el propietario del contrato, si fuera necesario.

Uno de los mecanismos particulares del EVM es que, aunque se pueden definir funciones en los lenguajes de alto nivel, la EVM no tiene soporte para dichas funciones. El mecanismo de invocación de funciones consiste en buscar en todo el código

compilado, si alguna signatura coincide con la función a llamar. En caso de coincidencia (de nombre y argumentos de entrada), se invoca dinámicamente la función.

Cuando no se encuentra ninguna función que coincida con la invocada, se ejecuta la función ***fellback*****,** una función especial en EVM que sirve para poder cubrir este caso.

Otro mecanismo particular es el tratamiento de excepciones. Cuando ocurre una excepción en el código de un contrato, se cancela todo posible cambio que la ejecución pudiera provocar. Se cancela el gas hacia los mineros, y se revierten todas las transacciones que pudieran haber ocurrido en el contrato. Básicamente se realiza un *rollback* del estado de la red. [1]

En definitiva, los contratos inteligentes son uno de los componentes troncales de la red Ethereum. Su despliegue se lleva a cabo por un interesado particular y se comportan como una dirección más en la red. Las otras direcciones pueden enviar o recibir dinero de los contratos, así como invocar sus funciones.

Los contratos requieren el combustible del gas, que limita la cantidad de operaciones que el contrato puede ejecutar. Los mineros que se encargan de ejecutar y verificar que el estado final de la máquina virtual sea correcto son recompensados con este gas.

Tecnologías de desarrollo

En la red de Ethereum, existen un total de 4 lenguajes de programación orientados a construir contratos inteligentes. Estos son:

- *Solidity*: el lenguaje más popular en la comunidad y que cuenta con más soporte.
- *eWASM*[1]: propuesta de sobreescritura de los contratos inteligentes bajo la tecnología de *WebAssembly*.
- *Vyper*: un lenguaje similar a *Solidity* pero con menos características y más enfocado a la seguridad.
- Ethereum *bytecode*: el código de máquina que se ejecuta en la EVM.

Entre ellos, el que se encuentra con más mantenimiento y más recursos en la web es *Solidity*, por lo que se elige este lenguaje para la implementación del contrato inteligente a bastionar.

Solidity es orientado a objetos, de tipado estático y soporta herencias, uso de librerías y la definición de tipos complejos, entre otras capacidades. Está basado en JavaScript. [9] [15]

[1] Disponible en https://ewasm.readthedocs.io/en/mkdocs/

Para poder crear un contrato inteligente y desplegarlo en una *blockchain*, no es suficiente con escribir un programa en el lenguaje *Solidity*. Se requiere un entorno de ejecución.

En este aspecto, se disponen de dos alternativas:

- *Remix*: un IDE de programación y despliegue de contratos inteligentes en la nube.[2]
- Preparar un entorno de desarrollo en local: requiere la instalación de varios componentes software para poder emular una *blockchain* en local, así como desplegar e interactuar con el contrato.

Aunque el IDE online es bastante completo, se prefiere poder desplegar el software en la propia máquina y así poder tener más flexibilidad si se requiere programar algún agente o realizar auditorías con herramientas. Por lo que se elige utilizar el entorno de desarrollo en local.

Respecto al desarrollo en local, existen varias opciones, pero la más recomendada es utilizar el siguiente *stack:*

- *Visual Studio Code Editor*: como editor de código ya que soporta la sintaxis de *Solidity* y tiene soporte para extensiones de contratos inteligentes. [3]
- *NodeJS*: servidor para poder desplegar las aplicaciones que crean la *blockchain* y el entorno de despliegue. También incluye el gestor de paquetes *npm* para instalar todas las dependencias necesarias.[4]
- *Truffle*: un entorno de desarrollo para los *Smart Contracts* en Ethereum. Incluye un compilador de código, un *framework* para pruebas de contratos, y varias herramientas más para el despliegue automático de contratos.[5]
- *Ganache*: herramienta creada por los mismos desarrolladores que *Truffle* que permite el despliegue de una *blockchain* en local.[6]

[2] Disponible en Remix - Ethereum IDE & community (remix-project.org)
[3] Disponible en https://code.visualstudio.com/
[4] Disponible en https://nodejs.org/en/
[5] Disponible en https://github.com/trufflesuite/truffle
[6] Disponible en https://github.com/trufflesuite/ganache-ui

Respecto a las herramientas de auditoría de seguridad, existe un gran conjunto de posibilidades, aunque muchas de ellas se encuentran en desuso o no tienen suficiente potencia para ser consideradas [20]. Las seleccionadas para realizar las pruebas de auditoría son:

- *MythX*: suite de análisis de seguridad de contratos inteligentes. Funciona a través de una API (requiere registro) y es compatible con *Truffle*. Únicamente tiene versión de pago.
- *Slither*: otra herramienta de análisis estático de contratos escritos en *Solidity*. Presume de tener muy pocos falsos positivos.[7]
- *Mythril*: otra herramienta de análisis estático que es utilizada por *MythX*. Se puede ejecutar de forma independiente para analizar contratos.[8]
- *Remix*: el propio editor online de código presenta sugerencias de seguridad, por lo que se puede realizar un análisis estático del código.

Adicionalmente, también se pueden programar pruebas dinámicas con *Truffle*, creando así un script de auditoría propio.

Vulnerabilidades en los Smart Contracts

Los contratos inteligentes, por su naturaleza descentralizada, son vulnerables a tipos de ataques muy distintos a los tradicionales ataques a servidores de arquitecturas centralizadas.

Adicionalmente, hay que tener presente que todos los participantes de la red pueden observar un contrato inteligente, y por lo tanto el código se encuentra completamente desnudo al público. Esto significa que un atacante puede estudiar con detenimiento la programación de un contrato en búsqueda de vectores de ataque.

Los ejemplos de vulnerabilidades son numerosos en la red, pero existe una taxonomía común que agrupa los tipos de vulnerabilidades según el componente de Ethereum que los ataques explotan. [1]

Las vulnerabilidades que se detallan a continuación están relacionadas directamente con el lenguaje de programación ***Solidity***. No son las únicas vulnerabilidades que pueden existir en los contratos inteligentes, ya que también existen vulnerabilidades en la *blockchain* y en la EVM de forma independiente. Estas últimas no se estudian en el proyecto. [5]

Hay que mencionar que la gran mayoría de vulnerabilidades presentes en los contratos inteligentes son debido a malas prácticas de programación de los contratos. En principio, la prevención es bastante efectiva cuando se tiene consciencia de los tipos de

[7] Disponible en https://github.com/crytic/slither
[8] Disponible en https://github.com/ConsenSys/mythril

vulnerabilidad. Por lo tanto, se deben estudiar estos tipos para poder bastionar correctamente un contrato. [24]

En primer lugar, los **errores aritméticos de desbordamiento** aprovechan la debilidad de los números enteros de ser desbordados a cero (o negativo) cuando sobrepasan su valor máximo representable.

Este tipo de error se da cuando se utilizan tipos de datos simples (como el *Integer* en *Solidity*) en vez de utilizar librerías matemáticas seguras. Los atacantes pueden aprovechar este fallo para desbordar una variable y cambiar su valor drásticamente.

De manera similar, también es posible cometer errores cuando se tratan con números decimales, ya que en Ethereum el tipo de datos de punto flotante no está implementado. Esto puede provocar que los desarrolladores implementen su propia aritmética de tipo flotante mediante números enteros, añadiendo posibles vulnerabilidades de fallos de precisión en el intento.

En segundo lugar, la **reentrada** es quizá la vulnerabilidad más peligrosa de todo contrato inteligente en la red de Ethereum. La reentrada explota la capacidad de los *callbacks* en la EVM durante la transferencia de fondos. Un *callback* es una función que se ejecuta en el código del contrato que previamente ha invocado la función de otro contrato.

Por ejemplo, un código que puede ser explotado mediante la reentrada tiene la siguiente forma:

```
// INSECURE
mapping (address => uint) private userBalances;

function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    (bool success, ) = msg.sender.call.value(amountToWithdraw)(""); // At this point,
the caller's code is executed, and can call withdrawBalance again
    require(success);
    userBalances[msg.sender] = 0;
}
```

El contrato atacante invoca la función *withdrawBalance,* que retira todos los fondos del solicitante y luego llama a la función *msg.sender.call.value.* Esta función es un *callback* y bloquea la ejecución del contrato hasta que el *callback* finalice. Esto redirige el flujo de ejecución al contrato atacante, que puede entonces realizar más llamadas a la función de retirar los fondos. Esta secuencia se puede ir repitiendo hasta que el contrato se quede sin fondos, ya que nunca llega al punto de actualizar el balance del atacante a 0.

En tercer lugar y de manera similar, la **llamada a lo desconocido** es otra característica que presenta graves vulnerabilidades si no se programa adecuadamente. En *Solidity*,

se permite llamar a una función de un contrato sin conocer el código del contrato, ya que únicamente se requiere la dirección y el nombre de la función.

Estas llamadas se pueden realizar con las directivas *call, send, y delegatecall.* Cuando la función invocada **no existe** en el contrato objetivo, la ejecución automáticamente invoca una función llamada **fellback** en el contrato emisor. Depende de cómo se programa esta función se pueden llegar a realizar llamadas no intencionadas a otros contratos, e incluso proporcionar el control total al atacante.

Similarmente, también se puede llegar a realizar un **denial of service (DoS)** mediante una combinación de excepciones y llamadas a lo desconocido. Se puede llegar a inhabilitar completamente un contrato, impidiendo que cualquier otro usuario interactúe con él, y así bloquear los posibles fondos almacenados.

En cuarto lugar, también existen las posibles **dependencias en el orden de ejecución**. Para invocar una función de un contrato inteligente se requiere realizar una transacción en la red de Ethereum. Esta transacción no tiene porqué ejecutarse en el mismo orden en que se ha enviado, ya que los mineros de la red pueden decidir posponer su inserción en un bloque posterior o incluso en qué orden concreto se inserta en el bloque actual, como ya se ha comentado anteriormente.

Esta incertidumbre del momento exacto de la ejecución de un contrato es una vulnerabilidad introducida, ya que las pruebas se pueden haber realizado en una red sin este tipo de inconveniente, y, por lo tanto, en una situación distinta. Un atacante puede intentar explotar esta posible ejecución no determinista.

En quinto lugar, las **excepciones** en los contratos inteligentes también son un vector de entrada para los atacantes. Cuando ocurre una excepción en un contrato (código que produce un error), se realiza un *rollback* en el estado de la máquina virtual. Esta situación se puede explotar, ya que modifica el resultado esperado de la llamada a una función de un contrato.

Una de las excepciones que pueden aparecer si el desarrollador no es consciente es el **out of gas exception**. Esta situación aparece cuando un contrato realiza más transacciones de las que puede hacer dado su gas disponible, provocando un error.

En sexto lugar, existen las llamadas **vulnerabilidades de visibilidad.** En *Solidity*, existen distintos modificadores de visibilidad de objetos (público, privado, accesible por otros contratos…) Si no se declaran explícitamente, los objetos declarados son siempre **públicos**. Cuando un desarrollador ignora el modificador en una función que debería ser privada, puede suponer un vector de entrada para los atacantes.

En séptimo lugar, se pueden comentar las **ilusiones de la entropía**. La EVM es una máquina completamente determinista, lo que significa que no existe el concepto de azar ni de números aleatorios. Cuando un contrato quiere generar algún tipo de evento

aleatorio, puede utilizar, por ejemplo, el hash de los bloques anteriores en la *blockchain* como generador de números aleatorios. Esto supone una gran amenaza, ya que los mineros podrían decidir explotar esta característica manipulando qué bloques incorporar en la libreta, adivinando de esta manera el hash que el contrato inteligente está utilizando como semilla.

Por último, la vulnerabilidad **ERC2.0 short address** es una vulnerabilidad compleja que permite realizar una especie de inyección (similar a la inyección SQL) al llamar a un contrato de la red. Para simplificar, se trata de un ataque susceptible a aquellas direcciones que terminan con 0's (por ejemplo, 0xF38D3400). El atacante puede ignorar los 0's a la derecha y aun así conseguir llamar a esa dirección. Esto puede provocar un desalineamiento entre los parámetros que espera la función y lo que realmente recibe, convirtiéndose en un vector de entrada para explotaciones.

Adicionalmente, existan muchas otras posibles vulnerabilidades en el estado del arte actual, aunque no son tal relevantes como las mencionadas.

- Nombramiento erróneo: nombrar de forma incorrecta un constructor del contrato. puede provocar que el constructor pueda ser invocado múltiples veces.
- Punteros no inicializados: utilizar una variable para acceder a regiones de la memoria no permitidas a priori.
- Uso incorrecto de *Tx.Origin*: *Tx.Origin* es una variable que indica la primera dirección en la pila que ha invocado una llamada al contrato. Se puede manipular a un usuario para usurpar su identidad mediante esta variable.
- Otras muchas vulnerabilidades más particulares.

El consenso principal es que con unas buenas **prácticas de programación** se pueden llegar a evitar gran parte de los posibles escenarios que se han presentado. [18] [1] [6] [5]

La siguiente tabla resume las vulnerabilidades explicadas hasta el momento:

Vulnerabilidad	Descripción
Errores aritméticos	Desbordamiento de los tipos *Integer.*
Reentrada	Múltiples llamadas mientras el contrato está bloqueado.
Llamada a lo desconocido	Invocar un contrato sin conocer la función que se ejecutará.
Denegación de Servicio	Bloquear un contrato a toda la red.
Dependencias de orden	No ejecutar la secuencia esperada de contratos.
Excepciones de ejecución / Out of Gas	Excepciones que reinician el estado de la EVM.
Errores de visibilidad	Mala declaración de la visibilidad de variables y funciones.
Ilusiones de entropía	Intentar realizar eventos aleatorios en el código.
ERC2.0 Token	Inyección mediante explotación de las direcciones en la red.

Tabla 2 vulnerabilidades en contratos inteligentes

Diseño del software

Análisis de requisitos

El contrato inteligente que se implementa es una **aseguradora descentralizada**. Esta aseguradora permite a los asegurados depositar dinero en formato de criptomonedas para poder afrontar un posible riesgo de forma conjunta con todos los fondos acumulados.

El contrato será una dirección más en la *blockchain*, por lo que los asegurados podrán enviar y recibir dinero de esa dirección. El contrato, aunque simple, debe proporcionar características que permitan a los asegurados verificar los posibles siniestros que sucedan.

Se permite contratar un conjunto de **productos**, cada uno con un tipo distinto de objeto asegurado. Para simplificar, los productos estarán limitados a cuatro tipos: autos, hogar, salud y barcos. Las primas de cada producto serán fijas y no se podrán cambiar.

Los asegurados podrán realizar **aportaciones periódicas** a cualquier producto. Esta aportación debe ser igual a la prima y en el momento de realizarla, el seguro entra en vigor hasta que termine el período cubierto por la aportación. Si se realiza una aportación superior a la prima, se devuelve la diferencia. Se comporta exactamente de la misma manera que cuando se contrata una póliza.

Cuando ocurre un siniestro, el asegurado realiza una petición de apertura de siniestro hacia el contrato. Esta petición tiene un título, una descripción, la fecha del siniestro, el **importe del siniestro** y un conjunto de documentos que sirven para demostrar que el siniestro es verídico (por ejemplo, fotografías, partes de la denuncia…). Todos estos datos se almacenan en el **mismo contrato**. Para simplificar, sólo se permitirá añadir una única evidencia al siniestro (que puede ser un fichero comprimido que contenga varios ficheros).

Esto inicia un proceso de **evaluación del siniestro**, donde todos los otros asegurados del producto pueden visualizar la petición del siniestrado y votar si aceptan o no cubrir el importe del siniestro con los fondos disponibles.

Si el proceso de votación finaliza con mayoría simple o empate, el importe del siniestro se **transfiere** al asegurado que ha iniciado el trámite (bajo petición). Se debe mencionar que el asegurado no puede votar en su propio siniestro.

El proceso de apertura del siniestro tiene una duración fija de una semana. El contrato cierra automáticamente el proceso una vez transcurrido ese tiempo y ya no se puede votar. Si hay como mínimo un voto, éste determinará el resultado de la transferencia de fondos. Si no hay votos, también se da por positivo el resultado de la votación.

Aunque este diseño se trate de un proceso simple, cubre las necesidades del proyecto ya que un atacante tiene distintas maneras de obtener los fondos disponibles en cada producto. Puede intentar manipular los resultados de las votaciones, cambiar las primas para invalidar pólizas, robar los fondos de forma directa, etc.

Diseño orgánico

El diseño orgánico consiste en describir cómo se deben plasmar los requisitos propuestos en el software final del producto. En este caso, cada actor en el sistema tendrá una **dirección** en la red, incluso el propio contrato.

Los actores pueden invocar las operaciones del contrato, que se ejecuta automáticamente en la *blockchain* y almacena el estado final después de cada invocación.

Los contratos disponen de memoria virtual. Pueden almacenar estructuras de datos e información. Las entidades que entran en juego en la aseguradora se pueden almacenar como conjuntos de elementos o bien diccionarios (que son un conjunto de elementos con una llave de acceso rápido a cada elemento).

El contrato debe almacenar los siguientes datos en su memoria:

- **Productos**. Es una lista que se inicializa estáticamente en el momento de desplegar el contrato. No se puede modificar. Cada producto tiene la siguiente información:
    - Nombre del producto: nombre distintivo del producto. De entrada, solo existirá *autos, hogar, salud y barcos.*
    - Prima del producto: cantidad que se debe aportar para que el seguro entre en vigor.
    - Periodo de validez: período temporal expresado en unidades de tiempo que especifica la validez máxima del seguro cuando se realiza una aportación.
    - Fondos del producto: fondos acumulados de todas las pólizas del producto.
    - Fondos reclamados: fondos en disputa por siniestros activos. Se transfieren de los fondos del producto cuando se abre un siniestro.
- **Pólizas**: son los contratos de los asegurados. Una póliza pertenece a un producto y tiene un período de cobertura. Almacena la siguiente información:
    - Id: número identificativo de póliza. Se autoincrementa.
    - Producto: nombre del producto para conocer a qué producto pertenece.
    - Dirección del asegurado: asegurado que ha realizado la aportación.
    - Fecha de inicio de cobertura.
    - Fecha de fin de cobertura.

- **Siniestros**: conjunto de siniestros que los asegurados han añadido al contrato para su verificación descentralizada. Cada siniestro contiene la siguiente información:
    - Id: identificador del siniestro. Se autoincrementa.
    - Número de póliza del siniestro.
    - Descripción del siniestro.
    - Fecha del siniestro.
    - Fecha de cierre del siniestro (calculada automáticamente como una semana después de la fecha del siniestro).
    - Importe reclamado.
    - Evidencia del siniestro: es una tupla que contiene la **URL** del recurso a descargar y su **hash** de verificación. El hash asegura que el recurso no haya cambiado en la URL después de ser añadido en el siniestro. Para simplificar, sólo se permite añadir una única evidencia, como ya se ha comentado.
    - Lista de votaciones: lista de tuplas que contiene el asegurado (dirección de la red) y su voto (positivo, negativo o neutral).

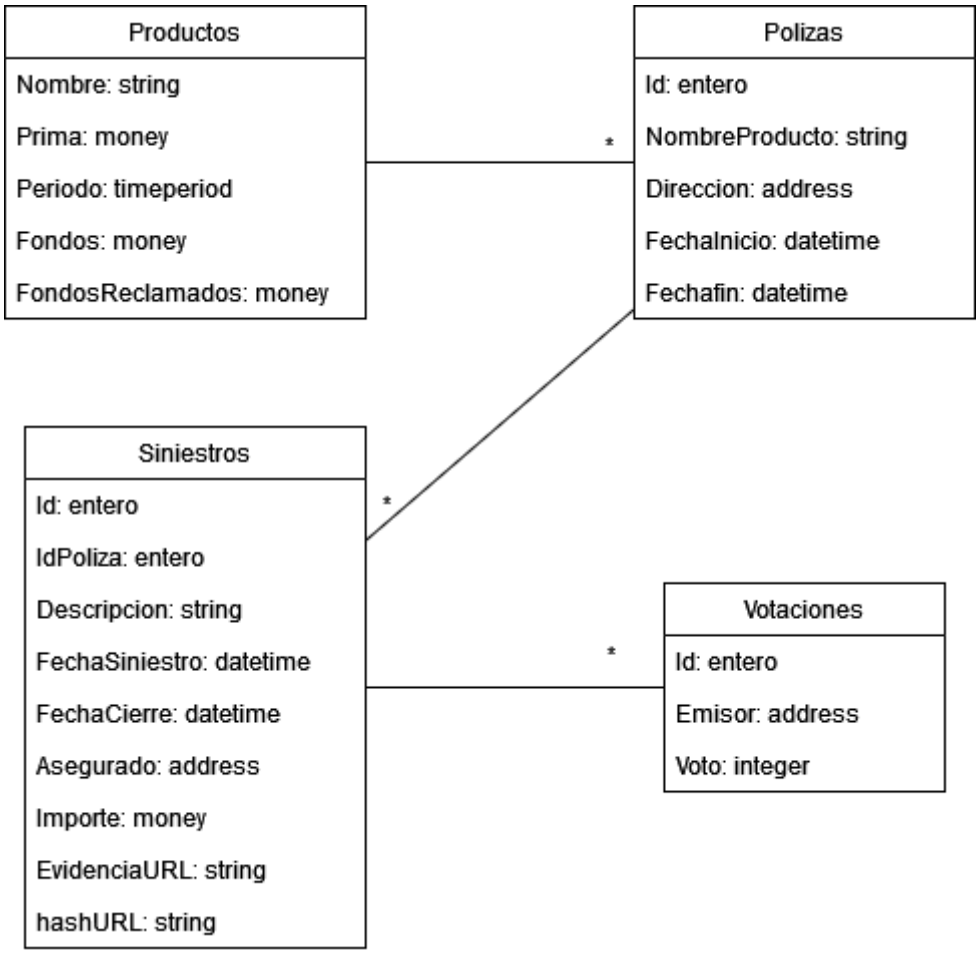

Ilustración 3 Representación UML del modelo. Fuente: elaboración propia

La razón por la cual se almacena el hash del recurso de la denuncia es para poder verificar que el recurso no haya cambiado después de su inserción en la *blockchain*.

Es importante mencionar que el contrato **no almacena** directamente las evidencias de la denuncia, sino que son los asegurados los que se encargan de subir en algún servidor estas evidencias y luego calcular su respetivo hash.

Es decir, el contrato en sí mismo no será el encargado de descargar el recurso y verificar que se hash sea correcto. El motivo es que el resultado de una transacción en un contrato inteligente **debe ser igual** para todos los nodos que lo verifican. Si se delega la responsabilidad de consultar un recurso web al contrato, puede producir distintos resultados en función de la región, de la disponibilidad de los servidores… y, por lo tanto, no es determinista.

Existen soluciones al respecto, como los **oráculos**, aunque se comentarán luego de realizar la auditoría de seguridad, ya que su implementación excede el alcance del proyecto. [27].

Hasta ahora se han visto los datos que se almacenan en la memoria de la máquina virtual que aloja el contrato. Ahora se detallan los contratos de las operaciones, que son las que cualquier nodo de la red puede invocar. En este caso, todas las operaciones son invocadas por otras direcciones físicas (no otros contratos).

Cada operación conlleva de forma explícita la dirección de la persona que la invoca, que es el asegurado (o solicitante). Las distintas operaciones son:

- *VerProductos*: devuelve la información de todos los productos. No tiene argumentos de entrada.
- *AportarRecibo*: aporta un recibo a un producto concreto. Si el usuario no había realizado ninguna aportación, esta operación crea la póliza. Si, por lo contrario, ya había realizado aportaciones, se extiende el período asegurado. Sus argumentos son:
    - Nombre del producto.
    - Cantidad aportada en criptomonedas.
- *VerFondos:* devuelve los fondos totales (sólo la información) en criptomonedas de un producto concreto. Sus argumentos son:
    - Nombre del producto.
- *VerPolizas:* proporciona las pólizas de un asegurado. Su argumento de entrada es:
    - Dirección del asegurado.
- *DeclararSiniestro*: declara un nuevo siniestro para una póliza. El solicitante debe tener contratada la póliza y la fecha del siniestro debe estar comprendida dentro de las fechas de cobertura de la póliza. Un asegurado no puede tener más de un siniestro activo por póliza. Los argumentos son:
    - Id de la póliza afectada.

    - Descripción del siniestro.
    - Fecha del siniestro.
    - Importe reclamado.
    - URL de la evidencia del siniestro.
    - Hash de la evidencia en SHA-256.
- *VerSiniestros*: obtiene los siniestros pendientes de resolución de un producto. Devuelve toda la información de un siniestro para que el asegurado pueda verificar su veracidad.
    - Nombre del producto: nombre del producto a filtrar.
- *RealizarVotacion*: permite que un asegurado de un producto pueda votar en un siniestro. Este asegurado no puede ser el que ha declarado el siniestro. El voto puede ser positivo, negativo o neutral. El voto se puede cambiar mediante otras invocaciones de esta función.
    - Id del siniestro: código que identifica el siniestro a votar.
    - Votación: voto emitido.
- *VerVotaciones*: devuelve una lista con todas las votaciones emitidas para un siniestro. Sus parámetros son:
    - Id del siniestro: código identificativo del siniestro.
- *RetirarFondos*: esta función se encarga de transferir los fondos acumulados de un producto hacia la cuenta de un asegurado que haya ganado una disputa de siniestro. Esta función comprueba si un siniestro ya se ha cerrado (fecha actual superior a la fecha de cierre del siniestro). Si está cerrado, se suman todas las votaciones y se obtiene el resultado. Un resultado mayor o igual que cero implica que el asegurado ha ganado la disputa. Los argumentos de esta función son:
    - Id del siniestro.

En definitiva, estas son las operaciones y los datos que el contrato deberá almacenar para poder representar una aseguradora descentralizada. Esta funcionalidad se implementará en *Solidity* para posteriormente bastionarlo para reducir el riesgo de ser atacado.

Desarrollo del proyecto

En este apartado se resumen los pasos seguidos para alcanzar los objetivos finales del proyecto. Esta entrega se divide en las siguientes partes:

1. Implementación del proyecto: implementación en código del software a construir. Su artefacto de salida es el código del contrato inteligente en una versión inicial.
2. Pruebas funcionales: pruebas que se han realizado para garantizar la funcionalidad propuesta en el análisis de requisitos.
3. Análisis de vulnerabilidades: resultados de las herramientas de análisis de vulnerabilidades en los contratos inteligentes propuestas en los capítulos anteriores.
4. Bastionado final: explicación de los cambios realizados en el código respecto a la primera versión del contrato para asegurar su bastionado.

Implementación del contrato

El entorno de desarrollo local, con todo el código y las pruebas implementadas, se puede consultar en el siguiente repositorio: https://github.com/Jolicost/Smart-Insurance. Todo el código se ha escrito en **inglés**. Las funciones se han traducido del diseño orgánico con la máxima fidelidad posible.

La implementación en *Solidity* de la aseguradora descentralizada se puede consultar en el fichero *Insurance.sol* del repositorio de código, que es el código del contrato.

La implementación es exitosa y permite cubrir todos los requisitos propuestos. Los aspectos más relevantes durante la primera fase de programación se explican a continuación.

Gestión de las unidades temporales

En primer lugar, en este tipo de contrato, la capacidad de manipular y trabajar con fechas es crucial. Pero la EVM no puede trabajar de forma nativa con fechas, ya que su naturaleza es descentralizada. Para solucionar este problema, se puede utilizar la función:

```
block.timestamp
```

Que devuelve la fecha de minado del bloque actual en la *blockchain*. No es una medida universalmente precisa, pero es suficiente para los propósitos del proyecto. Esta función devuelve segundos que han pasado desde la época de Unix (1 de enero de 1970) hasta el minado del último bloque en la *blockchain*.

Por lo tanto, todas las variables que trabajan con unidades temporales se pueden almacenar como **números enteros** y con segundos como unidad principal. De esta

forma se puede averiguar si una póliza está inactiva, si un voto se encuentra dentro del período de votación de un siniestro, etc. [22]

Almacenamiento de la memoria

En segundo lugar, se debe mencionar el tipo de datos estrella de *Solidity:* el ***mapping***. El *mapping* es un diccionario que tiene una clave y un valor. Las operaciones permitidas sobre este tipo de datos son las de inserción/actualización y consulta. [12]

El *mapping* resulta extremadamente idóneo ya que permite almacenar los objetos del modelo de la aseguradora separados por su clave, y acceder a ellos directamente sin necesidad de realizar operaciones complejas:

- Los productos por su alias de producto.
- Las pólizas por su identificador.
- Los siniestros por su identificador.
- Los votos por su identificador.

El código de los *mappings* que almacena la información del modelo asegurador toma la siguiente forma, en código *Solidity*:

```
// Main object storage
mapping(string => Product) private products;
mapping(uint256 => Policy) private policies;
mapping(uint256 => Sinister) private sinisters;
mapping(uint256 => Vote) private votes;
```

Cada objeto del modelo (Productos, Pólizas…) se define en un ***struct***. Una estructura que permite almacenar las propiedades de cada clase de objeto (por ejemplo, la de *Vote* almacena el emisor del voto, el identificador del siniestro y el valor del voto).

En tercer lugar, también se deben comentar los problemas encontrados al intentar almacenar los datos. En la EVM, los datos principalmente pueden ser almacenados en dos sitios distintos:

- El *Storage*. También nombrado estado de la *blockchain*. Son datos que se almacenan en la *blockchain* de Ethereum y por lo tanto son muy caros de modificar (ya que deben ser minados para ser incorporados al bloque). En el *storage* se almacenan los datos persistentes que deben estar presentes en el contrato.
- La *Memory*: Seria el equivalente a almacenar datos en la memoria RAM de la EVM. Es memoria volátil y que desaparece una vez se ha finalizado la función invocada en el contrato.

El contrato debe almacenar todos los objetos en el *Storage*, obviamente. Si no lo hiciera, la información de las pólizas, siniestros, votos… se perdería entre invocaciones. El problema aparece cuando se requiere obtener información derivada de esos objetos.

Por ejemplo, para obtener las **pólizas activas de un asegurado** para un producto, se dispone de dos alternativas:

1. Almacenar la información mínima en el *Storage* y calcular dinámicamente esa información cuando se invoca la función, iterando todas las pólizas y filtrando aquellas de un asegurado concreto.
2. Almacenar directamente la información derivada en el *Storage* en forma de *mappings.*

A priori se podría pensar que la opción 1 es mejor que la 2, ya que no consume recursos en la *blockchain*. Sin embargo, esa opción también conlleva unos costes, ya que tener que calcular esa información derivada mediante código puede suponer **costes computacionales.**

La red de Ethereum prevé la denegación de servicio mediante el uso del **gas** para ejecutar código. Esto significa que, a más instrucciones ejecutadas, mayor coste supone al interesado que invoca la función, por lo que ambas opciones pagan un precio.

Aunque calcular ese precio se escapa de los límites del proyecto, la opción de guardar los cálculos derivados en memoria resulta mucho más sencilla, ya que entra en sinergia con otro conflicto del lenguaje de programación de *Solidity:* **no es posible** que una función devuelva **una lista de elementos dinámicos** (con posibilidad de devolver un número distinto de elementos entre diferentes llamadas).[9]

Por lo que, para calcular las funciones derivadas como:

- Obtener las pólizas de un asegurado.
- Obtener la póliza de un asegurado y producto.
- Obtener los siniestros de un producto.
- Obtener los votos de un siniestro.
- Otras.

Resulta más adecuado utilizar los *mappings* en la memoria de tipo *storage* de la *blockchain*. Por ejemplo, las funciones derivadas de consulta de pólizas corresponden a los siguientes *mappings:*

```
// Accessors. Sacrifice memory to gain quick access to specific queries
mapping(string => uint256[]) private policiesByProduct;
mapping(address => uint256[]) private policiesByOwner;
mapping(address => mapping(string => uint256)) private policyByAddressAndProduct;
```

[9] Fuente: https://ethereum.stackexchange.com/questions/46761/how-to-fill-dynamic-in-memory-array

Cada *mapping* almacena identificadores, por lo que para obtener la información final se deberá consultar el *mapping* de objeto adecuado. Para actualizar estos diccionarios, sólo es necesario hacerlo cuando se insertan nuevos datos o bien cuando se modifican:

```
// Adds the policiy to the persistent storage
function addPolicy(string memory product_alias, address owner, Policy memory policy)
private {
    policiesByProduct[product_alias].push(policy.id);
    policiesByOwner[owner].push(policy.id);
    policyByAddressAndProduct[owner][product_alias] = policy.id;
    policies[policy.id] = policy;
}
```

Mecanismos de rollback

En cuarto lugar, también resulta relevante comentar el mecanismo utilizado para realizar ***rollbacks*** de las transacciones. Cuando una transacción no finaliza como debería ser (por ejemplo, porque una dirección ha intentado abrir un siniestro con un importe reclamado superior a los fondos), se debe lanzar una excepción y volver al estado inicial de la *blockchain* para que no haya ningún tipo de cambio persistente.

Esto se consigue gracias a la función ***require*** de *Solidity.* Esta función realiza una evaluación booleana. Si no es cierta, se lanza una excepción (pudiendo especificar un mensaje), y se vuelve al estado original antes de la llamada. Por ejemplo, para comprobar que una póliza no esté expirada, se utiliza el siguiente código:

```
// Check that policy is active
require(isPolicyExpired(senderPolicy) == false, "The current policy for the sender
has expired");
```

Medidas de seguridad básicas

Por último, comentar algunas de las medidas de seguridad que se han llevado a cabo durante esta primera versión del contrato:

- Visibilidad de las funciones: aquellas funciones que alteran los datos de la memoria persistente se han declarado como **privadas.**
- Se controla que los votos solo puedan tomar los valores 1,0 o -1.
- No se utilizan generadores de números aleatorios en ningún lugar. Tampoco se genera ningún hash de datos.
- Se han eliminado las operaciones complejas (como cálculos derivados) para evitar problemas relacionados con el *out of gas.* La mayoría de las operaciones son accesos directos a memoria.
- Todas las aserciones con ***require*** para prevenir posibles explotaciones de la lógica.

- No se ha enfatizado en la **reentrada**. Se conoce que la función de reclamar un siniestro debería ser vulnerable a la reentrada. Se quiere validar si las herramientas de auditoría son capaces de detectar dicha vulnerabilidad.

Estructura final del contrato

El contrato final se estructura en las siguientes secciones:

1. Declaración del *pragma*: informa la versión del compilador de *Solidity* a utilizar.
2. *Structs* de datos: declaración de las estructuras de datos.
    a. *Vote*: tipo de datos de la votación, que incluye el identificador del siniestro, del emisor y el voto realizado.
    b. *Sinister*: representa un siniestro con sus fechas, su importe reclamado, el emisor…
    c. *Policy*: representa una póliza e incluye el propietario, las fechas de cobertura y el nombre del producto asegurado.
    d. *Product*: propiedades del producto como su prima, los fondos disponibles, los fondos reservados…
3. Memoria en la blockchain: datos que persisten entre llamadas y ejecuciones.
    a. *Mappings* de consulta: todos los *mappings* que almacenan en la *blockchain* aquellas funciones difíciles de calcular en tiempo de ejecución como las pólizas por producto, los votos por siniestro…
    b. *Mappings* de entidades: almacenan todas las entidades (votos, siniestros, pólizas y productos) a partir de su identificador.
    c. Otros datos: otros datos que se materializan en la *blockchain* como los últimos índices de cada entidad, el propietario del contrato y los nombres de los productos disponibles, entre otros.
4. Constructor del contrato: función que se invoca una única vez al crear el contrato.
5. Funciones *fallback* y *receive*: necesarias para poder enviar y recibir dinero en formato de ETH. No tienen cuerpo.
6. Funciones de prueba: para poder realizar las pruebas y que deben ser eliminadas en la versión pública del contrato.
7. Funciones públicas: funciones que pueden ser llamadas por otros nodos (usuarios o contratos) en la *blockchain*.
8. Funciones privadas: funciones que sólo puede invocar el propio contrato y que sirven para encapsular y proteger cierto código.

El diagrama final del contrato público *Insurance.sol* se detalla a continuación:

Tipos de datos (struct)

Vote | Product | Sinister | Policy

Memoria *Blockchain* (persistente)

Consultas de datos

```
mapping(string => uint256[]) private policiesByProduct;
mapping(address => uint256[]) private policiesByOwner;
mapping(address => mapping(string => uint256)) private policyByAddressAndProduct;

mapping(string => uint256[]) private sinistersByProduct;
mapping(uint256 => uint256) private sinisterByPolicy;

mapping(uint256 => uint256[]) private votesBySinister;
mapping(address => uint256[]) private votesByAddress;
mapping(address => mapping(uint256 => uint256)) private voteByAddressAndSinister;
```

Entidades

```
mapping(string => Product) private products;
mapping(uint256 => Policy) private policies;
mapping(uint256 => Sinister) private sinisters;
mapping(uint256 => Vote) private votes;
```

Otros

```
string[] private products_names;
uint256 private lastPolicyId;
uint256 private lastSinisterId;
uint256 private lastVoteId;
```

Funciones públicas del contrato

```
function getProducts() public view returns (string[] memory)
function getProductByAlias(string memory product_alias) public view returns(bool success, Product memory)
function getPoliciesByProduct(string memory product_alias) public view returns (uint256[] memory)

function getPolicyByOwnerAndProduct(string memory product_alias, address owner) public view returns(bool success, Policy
memory)
function getPoliciesByOwner(address owner) public view returns(uint256[] memory)

function addReceipt(string memory product_alias) public payable
function checkFunds(string memory product_name) public view returns(bool, uint256)

function declareSinister(uint256 policy_id, string memory description, uint256 timestamp_seconds_sinisterDate, uint256
claim, string memory url, uint256 _sha256) public returns(uint256 sinister_id)
function isSinisterActive(uint256 sinister_id) public view returns (bool)
function getSinister(uint256 sinister_id) public view returns (bool success, Sinister memory sinister)
function getSinistersByProduct(string memory product_alias) public view returns (uint256[] memory)
function voteSinister(uint256 sinister_id, int8 vote) public
function getSinisterVotes(uint256 sinister_id) public view returns (uint256[] memory)
function getSinisterVoteScore(uint256 sinister_id) public view returns(int8)
function claimSinister(uint256 sinister_id) public
```

Ilustración 4 estructura pública del contrato. Fuente: elaboración propia

Pruebas funcionales

Las pruebas funcionales se localizan en el siguiente fichero del repositorio. Todas las pruebas se han pasado satisfactoriamente en una *blockchain* local con 10 cuentas y 100 ETH para cada cuenta desplegada con *Ganache.*

El *framework* de pruebas utilizado es *Truffle*, que permite despelgar automáticamente el contrato en la red y ejecutar un fichero de pruebas.

En este fichero de pruebas se definen un conjunto de actores y una lista de operaciones que verifican el correcto comportamiento del contrato.

Para poder trabajar con facilidad con los tiempos y períodos de tiempo, se ha implementado la operación *setTimestamp* en el contrato, que sobrescribe el comportamiento normal para que todo el contrato tenga una referencia de tiempo actual artificial. Esta funcionalidad debe ser completamente eliminada si se despliega el contrato en un entorno real. Sólo la dirección que despliega el contrato puede utilizar dicha función.

Los personajes utilizados para las pruebas son ficticios y tienen los nombres: *Jorge, IronMan, Eren, Homer, ManoloEscobar, Sabina, Rosalía y Picasso.* Cada uno parte con 100 ETH para las pruebas. Las pruebas propuestas son las siguientes:

***Caso de prueba**: Vida básico* ***Personajes involucrados**: Jorge, IronMan* ***Producto**: vida* ***Prima producto**: 1 ETH.*					
Personaje	*Acción*	Resultado	ETH_0	ETH_f	*Fondo*
Jorge	Añadir recibo en vida. Pago 1 ETH.	Recibo añadido.	100	99	1
-	Comprobar que la póliza de Jorge esté en vigor.	Está en vigor.	-	-	-
Jorge	Añadir recibo en vida. Pago 1 ETH.	Recibo añadido.	99	98	2
-	Comprobar que el período se ha extendido.	El período se ha extendido.	-	-	-
IronMan	Añadir recibo en vida. Pago 5 ETH.	Recibo añadido. Se devuelven 4 ETH por exceder la prima.	100	99	3

Tabla 3 Caso de prueba vida básico

***Caso de prueba**: Hogar básico* ***Personajes involucrados**: Picasso* ***Producto**: hogar* ***Prima producto**: 2 ETH.*					
Personaje	*Acción*	Resultado	ETH_0	ETH_f	*Fondo*
Picasso	Añadir recibo en hogar. Pago 1 ETH.	Excepción. El pago no es suficiente	100	100	0
Picasso	Añadir recibo en hogar. Pago de 3 ETH.	Recibo añadido. Se devuelve 1 ETH por exceso.	100	98	2
-	Establecer tiempo actual luego de expiración de la póliza	Póliza expirada	98	98	2
Picasso	Añadir recibo en hogar. Pago de 2 ETH.	Las fechas de inicio y fin de cobertura se han actualizado.	98	96	4

Tabla 4 Caso de prueba hogar básico

***Caso de prueba**: Siniestros básico* ***Personajes involucrados**: Eren, Homer* ***Producto**: autos* ***Prima producto**: 3 ETH.*						
Personaje	*Acción*	*Resultado*	ETH_0	ETH_f	*Fondo*	*Reserva*
Eren	Añadir recibo en autos. Pago de 5 ETH.	Recibo añadido. Se devuelven 2 ETH por exceso.	100	97	3	
Homer	Añadir recibo en autos. Pago de 3 ETH.	Recibo añadido.	100	97	6	
Eren	Declarar siniestro reclamando 5 ETH.	Siniestro añadido.	97	97	1	5
Homer	Declarar siniestro con la póliza de Eren.	Excepción. El solicitante no es el propietario de la póliza.	-	-	-	-
Homer	Declarar siniestro con su póliza reclamando 2 ETH.	Excepción. Los fondos disponibles no son suficientes.	-	-	-	-
-	Actualizar tiempo actual para expirar las pólizas.	Pólizas expiradas.	-	-	-	-
Homer	Declarar siniestro reclamando 1 ETH.	Excepción. La póliza está expirada.	-	-	-	-
-	Actualizar tiempo actual para que las pólizas estén en vigor.	Pólizas en vigor.	-	-	-	-
Eren	Declarar siniestro reclamando 1 ETH.	Excepción. Ya existe 1 siniestro activo para esa póliza	97	97	1	5

Tabla 5 Caso de prueba siniestros básico

***Caso de prueba**: Siniestros y votaciones* ***Personajes involucrados**: Manolo Escobar, Sabina, Rosalía, Eren* ***Producto**: barcos* ***Prima producto**: 5 ETH.*						
Personaje	*Acción*	*Resultado*	ETH_0	ETH_f	*Fondo*	*Reserva*
Manolo	Añadir recibo en barcos. Pago 5 ETH.	Recibo añadido.	100	95	5	0
Sabina	Añadir recibo en barcos. Pago 5 ETH.	Recibo añadido.	100	95	10	0
Rosalía	Añadir recibo en barcos. Pago 5 ETH.	Recibo añadido	100	95	15	0
-	Hay que asegurar que el número de pólizas es 3	Son 3.	-	-	-	-
Manolo	Declarar siniestro reclamando 10 ETH.	Siniestro añadido	95	95	5	10
Sabina	Declarar siniestro reclamando 10 ETH.	Siniestro añadido.	95	95	0	15
Eren	Intentar votar en el siniestro de Manolo.	Excepción. No se tiene una póliza en vigor para el producto.	-	-	-	-
Manolo	Intentar votar en su propio siniestro.	Excepción. El propietario no puede votar.				
-	Actualizar tiempo para que la fase de disputa de los siniestros haya terminado.	Tiempo actualizado.	-	-	-	-
Sabina	Intentar votar en el siniestro de Manolo.	Excepción. La fase de votación del siniestro ha terminado.	-	-	-	-
-	Actualizar tiempo para que todos los siniestros estén abiertos.	Tiempo actualizado.	-	-	-	-
Sabina	Intentar votar en un siniestro inexistente.	Excepción. No existe el siniestro.	-	-	-	-
-	Actualizar tiempo para que las pólizas hayan expirado.	Tiempo actualizado.	-	-	-	-
Sabina	Intentar votar en el siniestro de Manolo.	Excepción. La póliza de Sabina ya no está en vigor.	-	-	-	-
-	Actualizar tiempo para que las pólizas estén en vigor.	Tiempo actualizado.	-	-	-	-
Sabina	Votar +1 en el siniestro de Manolo.	Voto emitido.	-	-	-	-
Rosalía	Votar +1 en el siniestro de Manolo.	Voto emitido.	-	-	-	-

Tabla 6 caso de prueba siniestros y votaciones

***Caso de prueba**: Siniestros y votaciones (continuación)* ***Personajes involucrados**: Manolo Escobar, Sabina, Rosalía, Eren* ***Producto**: barcos* ***Prima producto**: 5 ETH.*						
Personaje	*Acción*	*Resultado*	ETH_0	ETH_f	*Fondo*	*Reserva*
Manolo	Intentar reclamar el importe del siniestro.	Excepción. El período de votación no ha terminado.	-	-	-	-
-	Actualizar fecha para que el período de votación del siniestro haya terminado	Fecha actualizada	-	-	-	-
Sabina	Intentar reclamar el importe del siniestro de Manolo.	Excepción. No es el propietario de la póliza.	-	-	-	-
Manolo	Reclamar el importe del siniestro	Fondos transferidos. Siniestro cerrado.	95	105	0	5
Manolo	Intentar volver a reclamar el importe	Excepción. Siniestro cerrado.	-	-	-	-
-	Actualizar tiempos para que se pueda votar en el siniestro de Sabina.	Tiempos actualizados	-	-	-	-
Manolo	Votar -1 en el siniestro de Sabina.	Voto emitido	-	-	-	-
Rosalía	Votar -1 en el siniestro de Sabina.	Voto emitido	-	-	-	-
-	Actualizar fecha para que el período de votación haya terminado	Fecha actualizada.	-	-	-	-
Sabina	Intentar reclamar su siniestro.	El siniestro se cierra. No se transfieren fondos ya que el resultado es negativo.	95	95	5	0
Manolo	Declarar un siniestro para barcos con importe reclamado de 5 ETH.	Siniestro abierto	105	105	0	5
-	Actualizar fecha para que el siniestro ya esté cerrado.	Fechas actualizadas	-	-	-	-
Manolo	Reclamar el importe del siniestro.	Como nadie ha votado, se da por buena la reclamación.	105	110	0	0

Tabla 7 caso de prueba siniestros y votaciones (cont.)

Todas las pruebas se pasan correctamente por lo que se da por validado el aspecto funcional del contrato. Ahora es el turno de realizar la auditoría de seguridad.

Análisis de vulnerabilidades

El análisis de vulnerabilidades se realiza con distintas herramientas. El objetivo es, en primer lugar, detectar posibles vulnerabilidades a corregir. En segundo lugar, también resulta interesante realizar una pequeña comparativa entre las herramientas de análisis disponibles en el mercado.

La primera herramienta utilizada es ***slither*,** de código abierto y programada en Python. La auditoría es de tipo estático y simplemente se debe informar la ruta del fichero del contrato a auditar. Los resultados detallan la descripción de la vulnerabilidad, el nivel (*high, medium, low, informational*) y un hipervínculo[10] hacia un listado de vulnerabilidades con ejemplos en la wiki de su repositorio.

Vulnerabilidad	Nivel	Descripción	Notas
Reentrancy en *claimSinister*	High	Se manipulan variables de estado después de realizar un *msg.sender.call*	Vulnerabilidad crítica detectada.
Dangerous strict equalities en algunas evaluaciones de *require*	Medium	Utilizar comparaciones estrictas que pueden fallar si un atacante envía ETH de antemano en el contrato. [28]	**Falso positivo** ya que está detectando aserciones de direcciones como si fueran aserciones de cantidades de ETH.
Uninitialized local variables al devolver una Póliza vacía cuando no se encuentra en *getPolicyByOwnerAndProduct*	Medium	No inicializar variables que luego son devueltas.	Es correcta y se debe corregir. Inicializar la variable a una póliza con id = 0 de forma explícita.
Missing events para la función *setTimestamp*	Low	No lanzar eventos para funciones que están protegidas al propietario.	Es correcto, pero hay que tener en cuenta que la función de *setTimestamp* no puede existir en producción.
Reentrancy en *addReceipt*	Low	Reentrada benigna al tener el mismo efecto que N llamadas consecutivas.	No es del todo correcto ya que puede tener implicaciones complejas y alteraciones en los fondos de los productos. Se debería revisar.
Block timestamp en varias comparaciones de tiempo	Low	El *block.timestamp* puede ser manipulado por los mineros.	Es correcto pero su probabilidad es muy baja. Se pueden investigar posibles alternativas.
Boolean equality en algunos *require* del código	Low	Utilizar *require*(condición == true) cuando se puede simplificar por *require*(condición)	Es correcto y se debe modificar.
Dead-code en una función huérfana	Low	Detectada una función privada huérfana que se debe eliminar. Desarrollo antiguo que fue reemplazado.	Eliminar función.
Incorrect-versions-of-solidity en el pragma del compilador	Low	La versión del compilador especificada es demasiado concreta. Se deben utilizar otras versiones recomendadas	Recomienda cambiar el pragma a las versiones 0.7.5 – 0.7.6. Pero el compilador instalado es el 0.8.13.

[10] La wiki se encuentra en https://github.com/crytic/slither/wiki/Detector-Documentation

Low-level-calls al transferir dinero a las cuentas (por exceso o por reclamar un siniestro)	Low	La función *call* para enviar dinero no tira nunca una posible excepción. Sino que devuelve un booleano de éxito o fracaso	Falso positivo ya que se comprueba posteriormente con un *require* que el envío de dinero haya tenido éxito.
Naming-conventions en declaraciones de variables	Info	Algunas variables no siguen el estándar de *mixedCase* de *Solidity*.	Arreglar los nombres.
Public functions should be external en algunas funciones públicas.	Low	Aquellas funciones con visibilidad pública que no son llamadas por el propio contrato deberían ser declaradas como *external* (ahorra gas)	Cambiar las visibilidades de las funciones.

Tabla 8 Vulnerabilidades detectadas en Slither

Como se puede observar, la herramienta detecta una sola vulnerabilidad crítica. La reentrada aparece en dos lugares en el contrato, aunque sólo detecta una como crítica. Adicionalmente, detecta dos falsos positivos y muchas vulnerabilidades sin demasiada importancia de tipo *low.*

En general, se puede decir que obtiene buenos resultados. Y, además, proporciona un enlace directo a la documentación sobre cómo solventar la vulnerabilidad, por lo que es una herramienta más que correcta.

La siguiente herramienta que se pone a prueba es ***mythril***. Esta herramienta puede realizar análisis estáticos de código y también tiene la particularidad de poder realizar inspecciones dinámicas de un contrato desplegado en una *blockchain*. Se evalúan sus dos modos de operación.

En análisis estático del código en *Solidity* reporta que **no se ha encontrado ninguna vulnerabilidad**. Después, se lanza la misma ejecución, pero bajo la *blockchain* en local. Esta ejecución puede tardar mucho tiempo por lo que se puede limitar el tiempo máximo que la herramienta está comprobando transacciones en la red.

```
PS C:\Users\joan> docker run -v ${PWD}:/tmp mythril/myth analyze -a 0xD410Eaae3a7951c00E564981Ea92663406D9aCe1 --rpc hos
t.docker.internal:7545 --execution-timeout 600
The analysis was completed successfully. No issues were detected.
```

Ilustración 5 Output de Mythril en Docker

Después de analizar también el contrato en tiempo real, tampoco se reporta ningún resultado. Algunos autores mencionan que esta herramienta se congela cuando el contrato a analizar es demasiado complejo. En definitiva, ***Mythril*** no cumple las expectativas.

Respecto al IDE online para *Solidity*, *Remix*, sí que dispone de herramientas de análisis estático. Se puede seleccionar entre distintas clases de análisis:

- De seguridad.
- De consumo de gas.
- De ERC.

- Misceláneas.

Se marca únicamente la opción de seguridad, que es la relevante para el proyecto. Después de compilar, el IDE genera una lista de potenciales amenazas, sin distinguir el nivel de criticidad.

Vulnerabilidad	Nivel	Descripción	Notas
Potencial de reentrada en *addReceipt* y en *claimSinister*	-	Detecta la posibilidad de reentrada y sugiere revisar la causa efecto enlazando un artículo[11] de la documentación de *Solidity*.	Vulnerabilidad crítica detectada.
Uso de *block.timestamp*	-	No recomienda utilizar *block.timestamp* ya que puede ser influenciado por los mineros.	Es correcto y se debería replantear.
Low-level-calls al transferir dinero a las cuentas (por exceso o por reclamar un siniestro)	-	La función *call* para enviar dinero no tira nunca una posible excepción. Sino que devuelve un booleano de éxito o fracaso	Falso positivo ya que se comprueba posteriormente con un *require* que el envío de dinero haya tenido éxito. (Igual que en *Slither*)

Tabla 9 Vulnerabilidades encontradas por Remix

Por lo tanto, *Remix* encuentra un subconjunto de las vulnerabilidades que ya había detectado *Slither*, pero sin describir su nivel de criticidad.

Por último, se pone a prueba la *suite* de análisis *Mythx*. Este software es propietario y únicamente se puede probar a través de la compra de packs de escaneo. Se adquieren tres escaneos para probar la versión no segura del contrato, y la futura versión bastionada.

Los escaneos que se pueden realizar son de tres niveles distintos:

- *Quick mode:* escáner simple que solo tarda unos 5 minutos.
- *Standard mode:* escáner normal que suele tardar unos 30 minutos.
- *Deep mode:* escáner más avanzado que puede llegar a tardar 90 minutos.

Se realizan dos escáneres para el contrato. Uno con el modo *standard* y el otro d*eep*. Sorprendentemente, los dos escáneres devuelven únicamente dos tipos de vulnerabilidades de nivel *low*. Estas se resumen a continuación:

[11] Disponible en https://docs.soliditylang.org/en/v0.8.7/security-considerations.html#re-entrancy

Vulnerabilidad	Nivel	Descripción	Notas
A floating pragma is set	Low	Recomienda no utilizar un rango de compiladores en el pragma del contrato para asegurar que el *Bytecode* compilado sea siempre igual.	Ya detectada anteriormente y es correcta.
State variable visibility is not set.	Low	La variable *insuranceOwner* que representa el propietario del contrato no tiene visibilidad definida.	Detección correcta, aunque ya se ha comentado que estas variables son sólo de pruebas.

Tabla 10 vulnerabilidades de Mythx

Por lo tanto, esta herramienta **falla** al detectar vulnerabilidades tan importantes como la reentrada. Y además su uso es de pago por suscripción. Sus resultados son un poco decepcionantes, si se comparan con otros proporcionados por herramientas de código abierto.

Finalmente, también se han intentado probar otras herramientas, con resultados poco favorables. La mayoría de las otras herramientas están desactualizadas, o no tienen soporte para compiladores modernos de *Solidity*, o bien no funcionan directamente. La siguiente tabla resume todas las herramientas de auditoría probadas hasta el momento, junto con las otras más desactualizadas y sus respectivas características:

Herramienta	Resultados	Ventajas	Desventajas
Slither	Muy buenos.	*Open-source*, Facilidad de instalación y de uso	Sólo realiza análisis estático. Algunos falsos positivos.
Remix	Buenos.	Online.	No distingue entre tipos de criticidad. Sólo análisis estático.
Mythx	Insuficientes	Facilidad de uso y reportes online en un *dashboard* propio. Análisis dinámico.	De pago por suscripción. Resultados muy deficientes respecto otras herramientas *open-source*.
Mythril	No parece analizar nada.	Análisis de contratos desplegados.	No soporta contratos complejos. No funciona en Windows.
SmartCheck[12]	No funciona.	-	Sólo funciona hasta el compilador 0.6.0 (el actual es el 0.8.X)
ContractGuard[13]	Error de red. No puede analizar el contrato	Online.	Se congela.
Securify2[14]	Error al ejecutar las pruebas en versiones modernas del compilador de Solidity (0.8.X)[15]	-	Desactualizada.

Tabla 11 resumen de herramientas de detección de vulnerabilidades

[12] Disponible en https://github.com/smartdec/smartcheck
[13] Disponible en https://contract.guardstrike.com/#/scan
[14] Disponible en https://github.com/eth-sri/securify2
[15] Problema notificado en https://github.com/eth-sri/securify2/issues/12

La conclusión de esta comparativa es que muchas de las herramientas de código abierto están desactualizadas y no tienen soporte activo de la comunidad. La única herramienta de pago que se ha probado ha reportado resultados poco satisfactorios.

Para finalizar, se diseña el caso de prueba que explota la vulnerabilidad de la **reentrada.** Para conseguir dicho fin, se crean un nuevo contrato: *ClaimAttack*. Este contrato se despliega en la misma *blockchain* que la aseguradora. El código del contrato importa el contrato de la aseguradora y puede interactuar con sus funciones.

Se programan unas pruebas en *Truffle* para demostrar los efectos adversos que pueden provocar dichos contratos. Se añaden dos nuevos personajes, *Ponzi* y *Madoff.* El personaje de *Ponzi* es el propietario del contrato atacante y es el único que puede interactuar con él para mandarle ciertas órdenes.

Caso de prueba: Ataque de reentrada
Personajes involucrados: Sabina, Rosalía, Manolo Escobar, Ponzi, Madoff
Producto: barcos
Prima producto: 5 ETH.

Personaje	*Acción*	*Resultado*	ETH_0	ETH_f	*Fondo*	*Reserva*
Sabina	Añadir recibo en barcos. Pago de 5 ETH.	Recibo añadido.	100	95	5	0
Rosalía	Añadir recibo en barcos. Pago de 5 ETH.	Recibo añadido.	100	95	10	0
Manolo Escobar	Añadir recibo en barcos. Pago de 5 ETH.	Recibo añadido.	100	95	15	0
Ponzi mediante contrato	Añadir recibo en barcos. Pago de 5 ETH.	Recibo añadido	100	95	20	0
Madoff	Añadir recibo en barcos. Pago de 5 ETH.	Recibo añadido	100	95	25	0
Ponzi mediante contrato	Declarar siniestro con importe 1 ETH	Siniestro añadido	-	-	24	1
Madoff	Declarar siniestro con importe 24 ETH	Siniestro añadido	.	.	0	25
Madoff	Votar positivamente en siniestro de Ponzi	Voto añadido	-	-	-	-
-	Actualizar fechas para que el período de votación haya terminado.	Fechas actualizadas	-	-	-	-
Ponzi mediante contrato	Reclamar importe del siniestro con reentrada. Llamar sucesivamente 15 veces	Fondos de reserva agotados	95	110	0	10
Ponzi	Transferir ETH de su contrato hacia su cuenta	ETH robados y transferidos	-	-	-	-

Tabla 12 prueba de ataque de reentrada

Esta prueba se implementa en la prueba automática *Insurance.attack.test.js* presente en repositorio del código. La prueba finaliza correctamente y esto demuestra que el ataque por reentrada es explotable en la primera versión de la aseguradora descentralizada.

Respecto a la inspección manual del código mediante la propia experiencia del desarrollador, se ha detectado que existe una potencial vulnerabilidad que ninguna herramienta ha podido encontrar. Se trata del **desbordamiento de votos.**

Al intentar reclamar un siniestro, los votos de éste se suman a una variable de tipo ***int8***. Esto se realiza para saber si el siniestro se da por reclamado (suma de votos igual o superior a 0), o rechazado (suma inferior a 0).

El problema se encuentra en que esta variable puede ser desbordada con facilidad, ya que sólo admite el rango de valores [-128,127]. Un atacante podría intentar desbordar el contador justo en el momento exacto para que la suma pasara de -128 a 0, y por lo tanto cambiar el resultado de la votación.

La solución a esta vulnerabilidad pasa por utilizar el tipo ***int256***, que tiene 256 bits y por lo tanto requiere un total de 2^255 votos para poderse desbordar, inviable a efectos prácticos.

Bastionado final

El bastionado final del contrato tiene como objetivo mitigar todas las posibles amenazas detectadas anteriormente. La siguiente tabla resume todas las vulnerabilidades detectadas y aquellas acciones que se llevan a cabo, con su correspondiente justificación.

Vulnerabilidad	Nivel	Acciones propuestas	Justificación	Detectado por
Reentrancy en *claimSinister* y en *addReceipt*	High	No utilizar *msg.sender.call* ya que no tiene límite de gas y se puede explotar. Utilizar la función *msg.transfer* que tiene un límite de gas establecido. Ajustar las variables de estado **antes** de realizar la transferencia para que una posible reentrada acabe en excepción.	Se elimina por completo la posibilidad de realizar la reentrada. Se escribirá otro caso de prueba para verificar que así sea.	Slither, Remix
Uninitialized local variables en *getPolicyByOwner AndProduct*	Medium	Inicializar las variables de forma explícita.	-	Slither
Desbordamiento de votos	Medium	Utilizar un tipo de datos de mayor rango para evitar el desbordamiento.	-	Developer
Uso de *block. timestamp*	Low	No se realiza ninguna acción.	Los posibles segundos de margen de manipulación que un minero pueda conseguir (30 s.) no son suficientes para alterar el aspecto troncal del contrato. [13]	Slither, Remix
Pragma compile	Low	Definir una versión concreta del compilador en vez de un rango de versiones.	-	Slither, Mythx
Missing events en *setTimestamp*	Low	No se realiza ninguna acción.	Es una función únicamente disponible para pruebas.	Slither
Boolean equality	Low	Corregir todos los *require* para simplificarlos.	-	Slither
Dead code	Low	Eliminar función huérfana.	-	Slither
Naming-conventions	Info	Cambiar los nombres a *mixedCase*	-	Slither
Public functions should be external	Low	Cambiar la visibilidad de las funciones recomendadas a *external*	El modificador *external* no consume tanto gas en la llamada.	Slither
State variable visibility not set.	Low	Cambiar la visibilidad de la variable *insuranceOwner* a *private*.	Aunque esta variable debe ser eliminada en producción.	Mythx

Tabla 13 acciones para mitigar las vulnerabilidades

La versión final del contrato se puede encontrar en el fichero *InsuranceHardened.sol* del repositorio de código. La prueba *Insurance.attack.failed.test.js* intenta realizar sin éxito el mismo ataque de reentrada explicado anteriormente sobre el contrato bastionado. El intento de reentrada termina en una excepción que revierte el estado de la *blockchain.*

Posteriores análisis con todas las herramientas no reportan ninguna de las vulnerabilidades corregidas.

Futuras mejoras

La funcionalidad de la aseguradora descentralizada, aunque bastante limitada, es perfectamente desplegable en una *blockchain* real. Aunque esto queda fuera del alcance del proyecto, sí que resulta interesante comentar aquellas posibles mejoras que garantizan aún más la seguridad del contrato.

La primera brecha de seguridad más obvia es el **uso de las pruebas de verificación** de un siniestro. Estas pruebas consisten en una URL y en un *hash* de verificación. Este mecanismo puede suponer un vector de entrada a los usuarios que puedan abrir la URL, ya que esta podría contener recursos maliciosos.

El contrato desplegado en la red de *Ethereum* tampoco sería capaz de verificar que el contenido de la URL coincida con el hash, ya que como se ha comentado, el resultado de una llamada a una función es verificada por todos los mineros de forma conjunta. Si el recurso **URL no está disponible** o es distinto para diversos mineros, esta verificación falla. Por no hablar de la posibilidad de que se descarguen los recursos maliciosos en los computadores de los mineros.

Adicionalmente, tampoco sería posible certificar que las evidencias de un siniestro realmente pertenezcan a las fechas descritas por el asegurado. Nadie puede asegurar que las fotografías, vídeos… sean relativas a las fechas anunciadas.

Los **oráculos** son entidades centralizadas que pueden proporcionar recursos externos a la ejecución de un contrato en una *blockchain* determinista.

Los contratos pueden realizar peticiones a estos oráculos, que serán los encargados de contactar con el recurso externo y proporcionarlo. Es importante mencionar que los oráculos son capaces de lanzar las peticiones en el **futuro**, proporcionando un mecanismo para certificar que dicho recurso tenía tal contenido en ese instante de tiempo. [11]

Por lo tanto, en este caso, sería el oráculo el encargado de comprobar que las evidencias de un siniestro sean verídicas (que su fecha y contenido sean los que se anuncian), y que no presenten una amenaza.

Los oráculos, aunque solucionan el problema de obtener recursos externos a la *blockchain*, son una entidad centralizada. Este hecho se enfrenta con el paradigma de la descentralización de la red que persigue el proyecto de Ethereum.

Adicionalmente, también se debería ampliar el modelo de datos que la aseguradora mantiene. Este modelo no representa la realidad, ya que una póliza real tiene muchos más datos, como la matrícula del vehículo, el año de matriculación, la fecha de nacimiento del asegurado… Con estos datos se evalúa el riesgo y por lo tanto la prima que los asegurados deben pagar.

Por último, una reflexión final acerca del proyecto Ethereum y su futuro camino. En este proyecto se ha podido observar que el consumo de gas que se requiere para ejecutar un contrato supone un bajo incentivo para que sus capacidades puedan llegar a ser beneficiosas.

Se ha comprobado experimentalmente que, en pocas operaciones, se consumían unas ingentes cantidades de ETH para aquellas direcciones que deseaban llamar a ciertas operaciones, como la de abrir un siniestro, abrir una póliza…

Este consumo de gas se debe a que el coste de realizar operaciones es altísimo en relación con el valor actual de ETH. Los mismos casos de prueba llegan a consumir entre 1 y 2 ETH sólo en consumo de invocaciones. En el valor de mercado actual, esto puede suponer entre 2000 y 4000 euros.

Existen estudios [4] dedicados exclusivamente a tratar de analizar las mejores formas de ahorrar gas en el diseño de los contratos inteligentes. Esto denota que hay una preocupación importante acerca del coste que supone desplegar y mantener un contrato inteligente. Si el coste resulta demasiado alto, puede ser que un contrato tradicional tenga menores costes asociados que su contrapartida en una *blockchain* descentralizada.

Este, entre otros motivos, han impulsado el cambio de paradigma del proyecto Ethereum desde *proof-of-work* hacia *proof-of-stake.* La nueva versión de Ethereum será la 2.0 y se prevé que finalice en 2023.

En el *proof-of-stake*, los nuevos bloques son verificados por un minero seleccionado al azar, que ofrece una cantidad de ETH en reserva a cambio de validar el bloque. Si intenta manipular el bloque, otros verificadores tumbarán el proceso y la reserva de ETH se perderá para siempre. Si se llega a un consenso, entonces recibe ETH en forma de recompensa, además de desbloquear los fondos. Esto es muy diferente al paradigma de *proof-of-work*, dónde se invierte tiempo y poder computacional para minar nuevos bloques.

Esto supone un cambio radical en la filosofía de Ethereum, ya que las transacciones pueden dispararse hasta 100000 al segundo (actualmente 30 cada segundo) y la

energía requerida para minar nuevos bloques se reducirá al 99%. Esto también disminuye sustancialmente el gas requerido para todas las transacciones, lo que implica menores costes de despliegue i operación de contratos inteligentes. [3]

A partir de aquí, mi opinión (que es meramente especulativa). Hasta ahora, parte del valor económico de las criptomonedas esta basado en su coste de minado directamente proporcional al eléctrico que supone para los servidores y máquinas hardware de minado. A medida que pasa el tiempo los bloques son más difíciles de minar por lo que su coste energético aumenta.

Este cambio de paradigma implica que el valor de ETH ya no se respaldará en el coste energético, por lo que su precio (al cambio hacia monedas FIAT) podría caer en picado.

La única forma de que el proyecto crezca llegados a este punto es que realmente Ethereum sea capaz de crear valor significativo para las personas. Esto significa que sus contratos inteligentes y aplicaciones descentralizadas jugarán un papel **crucial** en el éxito o fracaso del proyecto.

Al desasociar el valor de ETH de su esfuerzo de minado, lo que podrá tener valor en ese punto será el valor de los contratos y aplicaciones que se puedan crear bajo la red de Ethereum.

Si la red consigue enfocar correctamente esta transición, la seguridad de los contratos será cada vez más crucial, ya que toda la red se respaldará en su capacidad para implementar contratos que aporten valor y que solucionen problemas reales. Por lo tanto, deberán ser seguros al máximo para no comprometer la credibilidad del proyecto.

Conclusiones

Las vulnerabilidades de los contratos inteligentes son muy distintas a las que presentan los sistemas tradicionales de arquitecturas centralizadas. Esto se debe a que su funcionamiento y posibles vectores de entrada difieren mucho entre los dos paradigmas.

El código de un contrato inteligente no se puede alterar una vez desplegado, ya que se encuentra materializado en la *blockchain*. Además, su ejecución es resultado de un consenso entre todos los mineros, por lo que puede, en menor o mayor grado, ser influido por los intereses de estos.

El conocimiento acerca de las vulnerabilidades que tienen los desarrolladores en el momento de diseñar e implementar el contrato es quizá el factor que más determina el grado de seguridad de este. Una falta de conocimiento acerca de las vulnerabilidades conocidas e históricas puede llegar a suponer un gran riesgo para la seguridad de los contratos.

La vulnerabilidad más peligrosa es la *reentrada*. Es fácil de pasar por alto y puede conllevar unos enormes daños a las personas que depositan su confianza (y dinero) en un contrato inteligente.

Respecto a las herramientas de auditoría de seguridad de los contratos, la realidad es que o bien muchas de ellas están desactualizadas, o bien no aportan suficiente información ni tienen la capacidad de analizar contratos tan complejos como el del proyecto.

En este aspecto, eso sí, las herramientas *open-source* destacan por encima de las de pago. Las herramientas *open-source* detectan más vulnerabilidades críticas y también disponen de una mejor documentación online. En particular, la mejor herramienta de auditoría entre todas las probadas es *slither*.

También es importante mencionar que los costes asociados al despliegue y ejecución de un contrato son muy altos. Esto supone un gran impedimento para desarrollar contratos complejos, ya que el simple hecho de realizar operaciones y almacenar memoria se paga a un alto precio.

El cambio de paradigma que supone la migración de Ethereum a su nueva versión 2.0 puede significar una reducción total de estos costes, que podría impulsar el desarrollo de contratos hacia nuevas dimensiones.

Si así resulta, la seguridad de los contratos jugará un papel crucial en asentar la confianza de la gente a las tecnologías de la Web3. Este proyecto aporta un ejemplo de cómo bastionar un contrato inteligente desde cero hasta una versión segura.

Glosario

Blockchain: cadena de bloques que contienen información. Nuevos bloques se pueden añadir secuencialmente cuando son validados por los mineros de la red. Normalmente se utilizan para almacenar transacciones entre los participantes de la red.

DAO: Organización descentralizada autónoma. Es una entidad que almacena fondos de inversores en formato de criptomonedas.

ETH (Ether): moneda oficial de la red Ethereum con la que se pueden realizar y recibir pagos.

Ethereum: red descentralizada que ofrece servicios entre punto y punto a través de su propia moneda, Ether.

EVM: Máquina Virtual de Ethereum. Es un software que puede ejecutar el código de más bajo nivel de la red Ethereum. Es una máquina virtual que se ejecuta en todos los nodos de la red descentralizada.

Gas: combustible necesario para operar la red de Ethereum. Es una fracción de la unidad ETH.

Hash: función unidireccional que tiene una cadena de longitud arbitraria de entrada y una cadena de longitud fija de salida. Una misma entrada siempre produce una misma salida. La entrada no se puede calcular (a priori) con únicamente la salida.

Póliza: contrato que representa la vigencia de un seguro dentro de un período determinado para un asegurado.

Prima de un seguro: aportación periódica a una póliza para que se debe aportar para que las coberturas de la póliza sean activas.

Proof-of-work: paradigma donde los nuevos bloques en la *blockchain* son minados mediante potencia computacional.

Proof-of-stake: paradigma donde los nuevos bloques son consensuados mediante el bloqueo de fondos de algún particular que está forzado a verificar el bloque correctamente o perderá los fondos bloqueados.

Siniestro: evento adverso cuyos costes pueden estar cubiertos por una póliza.

Smart Contract: contrato inteligente que implementa un acuerdo entre dos o más partes. Su ejecución es automática y no depende de terceros.

Turing completo: máquina que puede solucionar cualquier problema computacional, dado un tiempo y memoria infinitos.

Web3: nuevo paradigma de Internet que se basa en la construcción de servicios y aplicaciones basadas en tecnologías descentralizadas apoyadas en *blockchains*.

Bibliografía

[1] Atzei, N., Bartoletti, M., and Cimoli, T. 2017. A survey of attacks on ethereum smart contracts (sok). In International conference on principles of security and trust (pp. 164-186). Springer, Berlin, Heidelberg.

[2] Bit2Me Academy. n.d. *Smart Contracts: ¿Qué son, cómo funcionan y qué aportan? - Bit2Me Academy*. [online] Available at: <https://academy.bit2me.com/que-son-los-smart-contracts/> [Accessed 1 March 2022].

[3] Castor, A., 2022. Why Ethereum is switching to proof of stake and how it will work. [online] MIT Technology Review. Available at: <https://www.technologyreview.com/2022/03/04/1046636/ethereum-blockchain-proof-of-stake/> [Accessed 24 May 2022].

[4] Di Sorbo, A., Laudanna, S., Vacca, A., Visaggio, C. A., and Canfora, G. 2022. Profiling gas consumption in solidity smart contracts. *Journal of Systems and Software*, *186*, 111193.

[5] Dika, A. 2017. Ethereum smart contracts: Security vulnerabilities and security tools (Master's thesis, NTNU).

[6] Esra, S., 2018. ICO Smart contract Vulnerability: Short Address Attack. [online] Medium. Available at: <https://medium.com/huzzle/ico-smart-contract-vulnerability-short-address-attack-31ac9177eb6b> [Accessed 26 March 2022].

[7] ethereum.org. n.d. *Ethereum Virtual Machine (EVM) | ethereum.org*. [online] Available at: <https://ethereum.org/en/developers/docs/evm/> [Accessed 1 March 2022].

[8] ethereum.org. n.d. *Introduction to smart contracts | ethereum.org*. [online] Available at: <https://ethereum.org/en/developers/docs/smart-contracts/> [Accessed 1 March 2022].

[9] ethereum.org. n.d. Smart contract languages | ethereum.org. [online] Available at: <https://ethereum.org/en/developers/docs/smart-contracts/languages/> [Accessed 15 March 2022].

[10] ethereum.org. n.d. Ethereum Whitepaper | ethereum.org. [online] Available at: <https://ethereum.org/en/whitepaper/> [Accessed 19 March 2022].

[11] Fravoll. n.d. Solidity Patterns. [online] Available at: <https://fravoll.github.io/solidity-patterns/oracle.html> [Accessed 7 May 2022].

[12] GeeksforGeeks. 2020. Solidity - Mappings - GeeksforGeeks. [online] Available at: <https://www.geeksforgeeks.org/solidity-mappings/> [Accessed 2 April 2022].

[13] Goldberg, P., 2018. Smart Contract Best Practices Revisited: Block Number vs. Timestamp. [online] Medium. Available at: <https://medium.com/@phillipgoldberg/smart-contract-best-practices-revisited-block-number-vs-timestamp-648905104323> [Accessed 7 May 2022].

[14] Hein, N., 2022. *Solidity vs Vyper explained - step-by-step beginners guides | QuickNode*. [online] Quicknode.com. Available at: <https://www.quicknode.com/guides/vyper/solidity-vs-vyper> [Accessed 2 March 2022].

[15] Hiremath, O., 2020. Setting Up A Smart Contract Development Environment | Edureka. [online] Edureka. Available at: <https://www.edureka.co/blog/building-smart-contract-development-environment/> [Accessed 18 March 2022].

[16] Ibm.com. n.d. *What is Blockchain Technology? - IBM Blockchain | IBM*. [online] Available at: <https://www.ibm.com/topics/what-is-blockchain> [Accessed 2 March 2022].

[17] López, A., 2018. *¿Qué son los 'smart contracts'?*. [online] elpais.com. Available at: <https://elpais.com/retina/2017/12/22/tendencias/1513937575_114270.html> [Accessed 1 March 2022].

[18] Manning, A., 2018. Solidity Security: Comprehensive list of known attack vectors and common anti-patterns. [online] blog.sigmaprime.io. Available at: <https://blog.sigmaprime.io/solidity-security.html> [Accessed 3 April 2022].

[19] Nakamoto, S. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, 21260.

[20] Piqueras, E., 2020. The Landscape of Solidity Smart Contract Security Tools in 2020. [online] Kleros. Available at: <https://blog.kleros.io/the-landscape-of-solidity-smart-contract-security-tools-in-2020/> [Accessed 25 March 2022].

[21] Planet Compliance. n.d. *In a nutshell: The DAO attack and how to steal $60 million worth of cryptocurrency - Planet Compliance*. [online] Available at: <https://www.planetcompliance.com/nutshell-dao-steal-60-million-worth-cryptocurrency/> [Accessed 1 March 2022].

[22] Proebsting, T., 2018. Writing a Contract That Handles Time. [online] Programtheblockchain.com. Available at: <https://programtheblockchain.com/posts/2018/01/12/writing-a-contract-that-handles-time/> [Accessed 3 April 2022].

[23] River Financial. n.d. *What Are Bitcoin Smart Contracts? | River Learn - Bitcoin Technology*. [online] Available at: <https://river.com/learn/what-are-bitcoin-smart-contracts/> [Accessed 1 March 2022].

[24] Senren, N., 2020. 8 Security Vulnerabilities in Ethereum Smart Contracts that can now be easily avoided!. [online] Medium. Available at: <https://medium.com/coinmonks/8-security-vulnerabilities-in-ethereum-smart-contracts-that-can-now-be-easily-avoided-dcb7de37a64> [Accessed 17 March 2022].

[25] Szabo, N., 1997. *Nick Szabo -- The Idea of Smart Contracts*. [online] Web.archive.org. Available at: <https://web.archive.org/web/20060615044959/http://szabo.best.vwh.net/smart_contracts_idea.html> [Accessed 1 March 2022].

[26] Tantikul, P. and Ngamsuriyaroj, S. 2020. Exploring Vulnerabilities in Solidity Smart Contract. In *ICISSP* (pp. 317-324). [Accessed 1 March 2022].

[27] Terek, Y., 2019. Make HTTP Request Using Your Solidity Smart Contract. [online] Medium. Available at: <https://medium.com/coinmonks/make-http-request-using-your-solidity-smart-contract-4f7173bd391c> [Accessed 27 March 2022].

[28] Trail of Bits Blog. 2019. Avoiding Smart Contract “Gridlock” with Slither. [online] Available at: <https://blog.trailofbits.com/2019/07/03/avoiding-smart-contract-gridlock-with-slither/> [Accessed 5 May 2022].

[29] Wright, E., 2019. Ethereum Network Fees: Everything You Need to Know. [online] Atomicwallet.io. Available at: <https://atomicwallet.io/academy/ethereum-network-fees> [Accessed 25 March 2022].

[30] Wu, J., 2021. *Ethereum's History: From Zero to 2.0*. [online] wisdomtree.com. Available at: <https://www.wisdomtree.com/blog/2021-07-15/ethereums-history-from-zero-to-20> [Accessed 1 March 2022].