Progetto DeFi: Attacco tramite Manipolazione degli Oracle

Laurea Magistrale in Informatica

Corso di Blockchain and Cryptocurrencies

Nitesh Kumawat
nitesh.kumawat@studio.unibo.it

14 aprile 2025

Indice

Capitolo 1

Introduzione Generale

Il settore della **Finanza Decentralizzata** (DeFi) ha introdotto un'era di innovazione senza precedenti, offrendo servizi finanziari aperti, trasparenti e accessibili a chiunque disponga di una connessione internet. Tuttavia, la **rapida evoluzione** e la natura complessa di questi protocolli hanno esposto nuove superfici di attacco, tra cui una delle più critiche è la **manipolazione degli** oracoli di prezzo. Gli oracoli sono servizi di terze parti che forniscono dati esterni, come i tassi di cambio degli asset, alle blockchain, e la loro integrità è **fondamentale per** il corretto funzionamento dei protocolli che ne dipendono.
Questo progetto si focalizza sull'analisi e la **dimostrazione pratica** di attacchi di manipolazione degli oracoli, una vulnerabilità significativa che può portare a **perdite finanziarie** sostanziali. Attraverso l'implementazione di due scenari distinti, il progetto esplora come un aggressore possa **sfruttare le** debolezze intrinseche degli oracoli di prezzo, in particolare quelli basati su **Automated Market** Maker (AMM) con liquidità limitata o che utilizzano prezzi spot.
Il primo scenario analizza un attacco in un ambiente **controllato e** didattico, utilizzando un semplice protocollo di prestito (`SimpleLender.sol`) che si affida a un AMM semplificato (`SimpleAMM.sol`) come oracolo di prezzo. Questo caso di studio illustra, passo dopo passo, come la **bassa liquidità** di un pool possa essere sfruttata per alterare artificialmente il prezzo di un asset e contrarre prestiti non **sufficientemente collateralizzati**.
Il secondo scenario eleva l'analisi a un contesto più **realistico**, simulando un attacco su un fork della mainnet di Ethereum. In questo caso, l'attacco viene orchestrato sfruttando un **flash loan** da un protocollo consolidato come Aave per **manipolare il** prezzo spot di una pool di liquidità di `Uniswap V2`, ingannando un **protocollo di** prestito vittima e sottraendone i fondi.
Attraverso la combinazione di **analisi teorica** e implementazione pratica, il progetto mira a fornire una **comprensione approfondita** delle meccaniche, dei rischi e delle implicazioni di sicurezza legati agli oracoli di prezzo nell'ecosistema DeFi.

1.1 Obiettivo

L'obiettivo primario di questo progetto è **dimostrare in** modo pratico la vulnerabilità dei protocolli di finanza decentralizzata (DeFi) che si affidano a oracoli di prezzo on-chain non robusti e, allo stesso tempo, **analizzare le** meccaniche di un attacco

di manipolazione del prezzo orchestrato tramite **flash loan**.
Per raggiungere tale scopo, il progetto si prefigge di **implementare e** documentare un attacco completo contro un protocollo di prestito semplificato, `SimpleLender.sol`, che utilizza un Automated Market Maker, `SimpleAMM.sol`, come fonte di prezzo. Questa dimostrazione mostrerà come un **agente malevolo** possa manipolare il tasso di cambio ETH/USDC attraverso uno scambio di grande entità, **sfruttare il** prezzo artificialmente gonfiato per depositare una garanzia sopravvalutata, e infine **contrarre un** prestito sovradimensionato rispetto al valore reale della garanzia per generare un profitto illecito.
Successivamente, l'analisi si estenderà a un ambiente più **complesso e** realistico, replicando un attacco simile su un fork della mainnet di Ethereum. Questo scenario comporterà l'utilizzo di un **flash loan** da un protocollo reale come Aave per ottenere il capitale necessario, la **manipolazione del** prezzo spot di una pool di liquidità di `Uniswap V2`, e lo **sfruttamento di** un protocollo di prestito la cui logica di valutazione della garanzia si basa direttamente su questo oracolo di prezzo spot vulnerabile.
Attraverso queste analisi, il progetto intende **evidenziare le** debolezze architetturali che rendono i protocolli vulnerabili, come la **dipendenza da** oracoli di prezzo spot e l'assenza di meccanismi di mitigazione come i `Time-Weighted Average Prices (TWAP)`. In definitiva, lo scopo è fornire una **chiara illustrazione** dei rischi associati alla **composabilità dei** protocolli DeFi e dell'importanza cruciale di implementare soluzioni di oracoli sicure e resistenti alla manipolazione.

Capitolo 2

Metodologia

La metodologia di questo progetto è stata strutturata per **dimostrare in** modo pratico e incrementale la vulnerabilità dei protocolli di finanza decentralizzata (DeFi) che si affidano a oracoli di prezzo on-chain non sicuri. L'approccio si è articolato in due **fasi principali**, ciascuna rappresentata da un caso di studio analizzato nei capitoli precedenti.

2.1 Fase 1: Simulazione di un Attacco in un Ambiente Controllato

La prima fase si è concentrata sulla **creazione di** un ecosistema DeFi minimale e controllato per isolare e dimostrare i meccanismi fondamentali di un attacco di manipolazione dell'oracolo.

Sviluppo dei Contratti Intelligenti Sono stati implementati in linguaggio Solidity tre contratti principali:
`SimpleAMM.sol`: Un **Automated Market** Maker (AMM) semplificato, che funge da oracolo di prezzo vulnerabile a causa della sua dipendenza diretta dalle riserve di liquidità.
`SimpleLender.sol`: Un protocollo di **prestito di** base che utilizza `SimpleAMM` come unica fonte per il prezzo degli asset, rendendolo il target dell'attacco.
`Attacker.sol`: Il contratto che **orchestra l'**attacco, eseguendo una sequenza di operazioni (swap, deposito, prestito) in un'unica transazione atomica per simulare un flash loan.

Setup dell'Ambiente È stato utilizzato un **ambiente di** sviluppo locale Hardhat per il deployment e il testing. Un token ERC20 di test (`TestUSDC.sol`) è stato creato per fornire la **liquidità iniziale** e simulare il collaterale.

Esecuzione della Prova L'attacco è stato eseguito tramite uno script di test (es. `OracleAttack.test.js`) che ha **simulato l'**intero flusso: l'attaccante prende un

"flash loan" (inviando ETH alla funzione di attacco), manipola il prezzo nell'AMM, deposita il collaterale nel lender, prende in prestito un importo gonfiato e infine ripaga il prestito iniziale, realizzando un profitto.

2.2 Fase 2: Analisi di un Attacco su un Fork della Mainnet

La seconda fase ha avuto l'obiettivo di **validare la** stessa tipologia di attacco in un ambiente più complesso e realistico, sfruttando le infrastrutture esistenti della mainnet di Ethereum.

Utilizzo di un Fork della Mainnet È stato impiegato il **framework Hardhat** per creare un fork locale della mainnet di Ethereum. Questo ha permesso di **interagire con** i protocolli DeFi reali e consolidati come Aave (per i flash loan) e Uniswap V2 (come AMM e oracolo di prezzo).

Sviluppo dei Contratti per l'Integrazione `LendingProtocol.sol`: Un nuovo contratto vittima è stato sviluppato per interagire direttamente con indirizzi e interfacce della mainnet, utilizzando una pool di `Uniswap V2` come oracolo di prezzo per il valore WETH/USDC.
`Attack.sol`: Il contratto dell'attaccante è stato **adattato per** interfacciarsi con il `IPool` di `Aave V3` per richiedere un vero flash loan e con il `IUniswapV2Router02` per eseguire gli swap di manipolazione.

Orchestrazione tramite Script L'intero processo è stato gestito tramite script JavaScript:
`deploy.js`: Si è occupato del deployment dei contratti (`LendingProtocol` e `Attack`) sul fork della mainnet. Una parte fondamentale di questo script è stata l'**impersonificazione di** un "whale account" per finanziare il contratto `LendingProtocol` con una quantità significativa di USDC, rendendo l'attacco realizzabile.
`attack.js`: Ha agito da trigger, invocando la funzione `attack()` sul contratto `Attack.sol` deployato, avviando così la sequenza di operazioni che includeva il deposito iniziale, la richiesta del flash loan da Aave, la manipolazione della pool Uniswap, il prestito dal `LendingProtocol` e il rimborso del flash loan.
Questo duplice approccio metodologico ha permesso non solo di comprendere la **teoria dietro** la manipolazione degli oracoli, ma anche di dimostrarne la **fattibilità pratica** con strumenti e protocolli standard del settore.

Capitolo 3

Attacco Oracle Price Manipulation

Questo progetto implementa e dimostra un attacco di **manipolazione** dell'oracolo di prezzo contro un semplice protocollo di prestito decentralizzato (`SimpleLender.sol`) che utilizza un **Automated Market** Maker (`SimpleAMM.sol`) come oracolo di prezzo per il valore ETH/USDC. L'attacco sfrutta la **bassa liquidità** dell'AMM per manipolare il prezzo di scambio durante una singola transazione (simulando un flash loan).
L'attaccante esegue le seguenti fasi: scambia una grande quantità di ETH per USDC nell'AMM per far salire **artificialmente** il prezzo di USDC rispetto a ETH, deposita l'USDC ricevuto nel contratto di prestito come **garanzia**, prende in prestito una quantità massimale di ETH basata sul prezzo manipolato dell'oracolo, e infine ripaga il flash loan, realizzando un **profitto significativo** dall'ETH preso in prestito.
Il contratto `Attacker.sol` orchestra questa sequenza di azioni in un'unica transazione.

3.1 Architettura del Protocollo

Il protocollo in questione è composto principalmente da due **contratti intelligenti**: `SimpleAMM.sol` e `SimpleLender.sol`, ai quali si aggiunge un token ERC20 di test (`TestUSDC.sol`). Questi contratti sono stati progettati per simulare un ambiente **DeFi basilare** di scambio e prestito.

3.1.1 Il Contratto `SimpleAMM`

Il contratto `SimpleAMM.sol` è un'implementazione semplificata di un **Automated Market** Maker (AMM), il cui scopo principale è facilitare lo **scambio** tra Ether (ETH) e un token ERC20 designato come USDC. La sua **architettura essenziale** lo rende uno strumento didattico efficace per comprendere le dinamiche di base di un AMM, ma allo stesso tempo evidenzia chiaramente le **vulnerabilità intrinseche** che possono emergere in sistemi con **liquidità limitata** e oracoli di prezzo **on-chain non sofisticati**.
All'interno della sua struttura, `SimpleAMM` mantiene un riferimento all'indirizzo del token USDC con cui opera, memorizzato nella variabile di stato pubblica `USDCAddress`. Questo indirizzo viene **configurato una sola volta** al momento della **deployment** del contratto tramite il suo costruttore, garantendo che l'AMM interagisca sempre con il token corretto.

La **gestione della** liquidità è fondamentale per qualsiasi AMM, e `SimpleAMM` offre funzioni pubbliche per interrogare i **bilanci correnti** di entrambi gli asset nel suo pool. La funzione `balanceETH()` restituisce la **quantità di** Ether detenuta dal contratto, mentre`balanceUSDC()` interroga il saldo dei token USDC chiamando il metodo `balanceOf()` sull'indirizzo del token stesso.
Le **funzioni di** prezzo all'interno di `SimpleAMM` sono di particolare rilevanza, poiché agiscono implicitamente come l'**oracolo di** prezzo per altri contratti, come vedremo con `SimpleLender`. Questi prezzi sono calcolati **dinamicamente e direttamente** basandosi sui rapporti delle riserve attuali nel pool.
Nello specifico, `priceUSDCETH()` calcola il valore di un singolo USDC in termini di ETH, utilizzando la formula `(balanceETH() * 1e18) / balanceUSDC()`, dove `1e18` scala il risultato per riflettere **18 decimali**, assumendo una compatibilità simile a ETH. Al contrario, `priceETHUSDC()` determina il prezzo di un ETH in USDC con la formula `(balanceUSDC() / balanceETH()) * 1e18`, anch'essa **scalata per** precisione.
Prima di eseguire uno **scambio effettivo**, gli utenti possono avvalersi delle **funzioni di** stima per prevedere l'ammontare che riceverebbero.

`getEstimatedEthForUSDC(uint256 amountFrom)` calcola l'ETH stimato per una data quantità di USDC, applicando la formula `(balanceETH() * amountFrom) / (balanceUSDC() + amountFrom)`. Similmente,

`getEstimatedUSDCForEth(uint256 amountFrom)` stima l'USDC che si otterrebbe per una quantità specifica di ETH tramite `(balanceUSDC() * amountFrom) / (balanceETH() + amountFrom)`. Queste stime sono basate sul **modello a** prodotto costante (`x*y=k`), un **principio comune** negli AMM.

Il **cuore operativo** del contratto è la funzione

`swap`, che consente agli utenti di effettuare **scambi reali**. Questa funzione è

`payable`, permettendo l'invio di ETH al contratto per gli scambi da ETH a USDC. Il meccanismo di swap distingue tra lo scambio di USDC per ETH e viceversa. Se l'utente scambia USDC, il contratto calcola l'ETH da ricevere, trasferisce gli USDC in entrata nel proprio bilancio tramite

`transferFrom` e invia l'ETH all'utente. Se invece l'utente scambia ETH, la funzione verifica che l'ETH inviato (

`msg.value`) corrisponda all'**importo dichiarato**, calcola l'USDC da ricevere e lo trasferisce all'utente. Il **risultato dello** swap viene sempre restituito. Il contratto include anche una funzione

`receive()`, che gli permette di accettare ETH **direttamente**, essenziale per la **gestione della** liquidità del pool.

La **natura semplificata** di `SimpleAMM` lo rende **intrinsecamente vulnerabile**, specialmente in condizioni di bassa liquidità. Grandi operazioni di scambio possono causare un **significativo slittamento** del prezzo, deviando notevolmente dal prezzo atteso. Più criticamente, la dipendenza dei prezzi dalle **riserve attuali** del pool apre la porta alla **manipolazione del** prezzo. Un attaccante con **capitale sufficiente**, o sfruttando un **flash loan**, può eseguire scambi voluminosi per alterare **artificialmente le** riserve e, di conseguenza, i prezzi riportati dalle funzioni dell'oracolo. Questa vulnerabilità è al **centro dell'**attacco dimostrato nel progetto, evidenziando come AMM con **liquidità ridotta** possano essere sfruttati come **oracoli di** prezzo inaffidabili.

3.1.2 Il Contratto `SimpleLender`

Il contratto `SimpleLender.sol` incarna un **protocollo di** prestito decentralizzato di base. La sua funzione principale è quella di permettere agli utenti di depositare token USDC come **garanzia** e, in cambio, prendere in prestito Ether (ETH). La semplicità della sua implementazione delle logiche di prestito e la sua dipendenza da un oracolo di prezzo esterno lo rendono un **caso di studio** illuminante per comprendere sia le meccaniche fondamentali del prestito on-chain sia le vulnerabilità intrinseche in tali sistemi.
A livello **architetturale**, `SimpleLender` gestisce alcune variabili di stato cruciali per le sue operazioni. `USDCAddress` e `ammAddress` memorizzano rispettivamente l'indirizzo del token USDC, accettato come garanzia, e l'indirizzo del contratto `SimpleAMM` che funge da oracolo di prezzo. Inoltre, `collateralizationRatio` è un **parametro configurabile**, espresso in basis points (dove 10000 equivale al 100%), che stabilisce il rapporto tra il valore della garanzia e l'ammontare massimo prestabile. Infine, una mappatura `USDCdeposits` tiene traccia della quantità totale di USDC che ogni utente ha depositato come garanzia all'interno del contratto. Tutte queste variabili vengono **inizializzate e** configurate al momento della deployment del contratto tramite il suo costruttore.
La funzionalità di deposito è gestita dalla funzione `depositUSDC(uint256 amount)`. Per poter depositare, gli utenti devono aver precedentemente **concesso autorizzazione** al contratto `SimpleLender` a spendere i loro token USDC. Una volta autorizzato, il contratto trasferisce l'importo specificato di USDC dal mittente al proprio indirizzo utilizzando `IERC20(USDCAddress).transferFrom()`. Successivamente, l'ammontare depositato viene registrato e aggiunto al saldo `USDCdeposits` dell'utente, **aggiornando la** sua garanzia disponibile.
Il cuore del **meccanismo di** valutazione della garanzia risiede nella funzione `getPriceUSDCETH()`. Questa funzione recupera il prezzo corrente di 1 USDC espresso in ETH. Tuttavia, un punto critico, esplicitamente commentato nel codice sorgente, è la sua vulnerabilità: `"External call to AMM used as price oracle"`. Ciò significa che `SimpleLender` si affida **direttamente e senza** filtri alla funzione `priceUSDCETH()` fornita dal contratto `SimpleAMM`. L'assenza di meccanismi di **validazione aggiuntivi**, come l'uso di oracoli multipli, medie temporali ponderate (TWAP) o ritardi, rende questo protocollo estremamente **suscettibile a** manipolazioni del prezzo dell'oracolo, specialmente se l'AMM di riferimento ha una bassa liquidità.
Basandosi sul prezzo dell'oracolo, la funzione `maxBorrowAmount()` calcola l'importo massimo di ETH che il mittente può prendere in prestito. È importante notare una **limitazione intrinseca**, evidenziata nel codice stesso: `"Does not take into consideration any existing borrows (collateral already used)"`. Questo implica che il calcolo si basa sull'intera quantità di USDC depositata dall'utente, ignorando qualsiasi prestito ETH già attivo che potrebbe aver già consumato parte di quella garanzia. Il calcolo procede recuperando i USDC depositati dall'utente, convertendoli in un valore equivalente in ETH utilizzando il prezzo dell'oracolo, e quindi applicando il `collateralizationRatio` (diviso per 10000 per convertire i basis points in una percentuale) per determinare l'**ammontare massimo** di ETH prestabile.
Infine, la funzione `borrowETH(uint256 amount)` consente all'utente di prelevare ETH dal contratto come prestito. Prima di procedere, viene eseguito un controllo

per assicurarsi che l'importo richiesto non superi il `maxBorrowAmount()` calcolato. Anche qui, il codice sottolinea la stessa limitazione menzionata in precedenza: `"Does not take into consideration any existing borrows"`. Se il controllo di validità passa, il contratto tenta di trasferire l'ETH all'utente tramite una chiamata diretta `msg.sender.call{value: amount}`, **verificandone il** successo. Come la maggior parte dei contratti DeFi che gestiscono ETH, `SimpleLender` include anche una funzione `receive()` che gli consente di accettare **depositi diretti** di Ether, cruciale per mantenere le sue riserve di prestito.
In sintesi, `SimpleLender.sol` serve come **modello didattico** per un protocollo di prestito, ma la sua architettura semplificata e, in particolare, la sua **dipendenza diretta** e non filtrata da un oracolo di prezzo on-chain come `SimpleAMM`, lo rendono estremamente vulnerabile in contesti reali. Le **omissioni nella** gestione del debito esistente e l'assenza di meccanismi di **liquidazione o** di interessi accentuano ulteriormente la sua natura di esempio per lo studio di vulnerabilità, piuttosto che di un protocollo pronto per la produzione.

3.2 Contratti Ausiliari: `TestUSDC.sol` e `Attacker.sol`

Per completare il quadro dell'architettura del protocollo e dimostrare concretamente la **vulnerabilità**, il progetto si avvale di due **contratti ausiliari** fondamentali: `TestUSDC.sol` e `Attacker.sol`.
Il contratto `TestUSDC.sol` è una realizzazione diretta e **semplificata di** un token ERC20. La sua funzione principale, al di là delle operazioni standard di un token, è la capacità di **coniare nuovi** token e assegnarli a indirizzi specifici tramite la funzione `mint(address receiver, uint256 amount) external`. Questo meccanismo è cruciale per il **setup dell'**ambiente di test, permettendo di distribuire facilmente USDC agli account necessari per simulare scenari di **liquidità nell'**AMM e di collateralizzazione nel protocollo di prestito.
Il vero **motore dell'**exploit è il contratto `Attacker.sol`, il cui ruolo è quello di orchestrare l'intero attacco di **manipolazione dell'**oracolo. Questo contratto è stato progettato per eseguire la sequenza di azioni necessarie a sfruttare le vulnerabilità in `SimpleLender` e `SimpleAMM` all'interno di un'unica **transazione atomica**. Questa esecuzione singola e indivisibile è fondamentale per simulare l'efficienza e l'impatto di un `"flash loan"`, dove il capitale viene **preso in** prestito e ripagato all'interno dello stesso blocco transazione.
L'attacco viene concretamente implementato attraverso la funzione `executeAttack(address amm, address usdc, address lender) external payable` di `Attacker.sol`. Questa funzione, essendo `payable`, è in grado di ricevere un importo iniziale di ETH, che simula il capitale di un flash loan, e che sarà poi **rimborsato al** termine dell'operazione.
Il **flusso dell'**attacco segue una sequenza ben definita, come dettagliato anche nel file di test `OracleAttack.test.js`. Inizia con l'acquisizione di capitale: la funzione `executeAttack` viene chiamata con un valore ETH allegato (nel test, 2 ETH), che rappresenta il **"flash loan"** iniziale. Una volta che l'ETH è disponibile, l'attaccante procede alla **manipolazione del** prezzo all'interno del `SimpleAMM`. Viene eseguito uno **swap significativo**: una grande quantità dell'ETH appena acquisito (i 2 ETH del flash loan) viene scambiata per USDC. Dato che il `SimpleAMM` è inizializzato con

una **liquidità relativamente** bassa (ad esempio, 1 ETH e 3000 USDC nel test), questo swap massiccio altera drasticamente il rapporto ETH/USDC all'interno del pool. Di conseguenza, il prezzo di 1 USDC, come riportato dalla funzione `priceUSDCETH()` del `SimpleAMM`, viene **artificialmente gonfiato**, passando da un valore iniziale di circa 0.00033 ETH a 0.003 ETH dopo lo swap.
Subito dopo questa manipolazione, l'USDC ricevuto dallo swap (ottenuto a un prezzo **vantaggioso grazie** all'alterazione del mercato) viene immediatamente approvato e depositato nel contratto `SimpleLender` come garanzia. Il `SimpleLender` registra questo deposito, e a causa del prezzo di USDC ora **artificialmente elevato**, il valore della garanzia dell'attaccante risulta enormemente amplificato.
Con una **garanzia apparentemente** elevata, l'attaccante procede a richiedere l'ammontare massimo di ETH che il `SimpleLender` gli consente di prendere in prestito. A causa della **lettura del** prezzo manipolato dall'oracolo, il `maxBorrowAmount` calcolato dal lender sarà significativamente superiore a quanto sarebbe stato con il prezzo reale. Nel test, ciò permette all'attaccante di **prelevare 4.8** ETH.
Una volta completato il prelievo di ETH, l'attaccante esegue l'ultimo passaggio cruciale: il ripagamento del **"flash loan"** iniziale. I 2 ETH presi in prestito all'inizio vengono **restituiti all'**indirizzo del chiamante della funzione `executeAttack`. Il successo di questo trasferimento è verificato. La **differenza tra** l'ETH prelevato dal lender e l'ammontare ripagato per il flash loan costituisce il **profitto netto** dell'attaccante. Nel contesto del test, questo si traduce in un **guadagno netto** di 2.8 ETH per l'attaccante. L'intero ciclo si svolge in un'unica **transazione atomica**, rendendo l'attacco istantaneo e resistente a interventi esterni, e dimostrando efficacemente come la **fiducia in** oracoli di prezzo non robusti, specialmente quelli derivati da AMM con **liquidità limitata**, possa esporre i protocolli DeFi a **rischi finanziari** sostanziali.

Capitolo 4

Analisi di un Attacco Flash Loan e Manipolazione dell'Oracolo su un Protocollo di Prestito Decentralizzato (Fork Mainnet)

Questa **analisi tecnica** esamina un'implementazione di attacco di **manipolazione del** prezzo on-chain, mirato a un contratto di protocollo di prestito simulato, e modellato sulla **vulnerabilità degli** oracoli di prezzo spot in ambienti DeFi.
Il **fulcro dell'**attacco risiede nello sfruttamento di un oracolo di prezzo che deriva il valore del collaterale direttamente dalle riserve di una pool Uniswap V2, rendendolo suscettibile a manipolazioni attraverso **volumi di** trading significativi.
L'attaccante inizializza l'operazione **depositando una** quantità di WETH come collaterale nel protocollo vittima.
Successivamente, viene eseguito un **flash loan** da Aave di un'ingente quantità di WETH.
Questi fondi vengono immediatamente utilizzati per eseguire uno `swapExactTokensForTokens` sulla pool Uniswap WETH/USDC, **alterando artificialmente** il rapporto delle riserve e, di conseguenza, il prezzo spot di WETH rispetto a USDC.
Con il prezzo di WETH **temporaneamente gonfiato**, l'attaccante invoca la funzione `borrow` sul protocollo di prestito vulnerabile, ottenendo un prestito di USDC **significativamente maggiore** rispetto al valore reale del collaterale depositato prima della manipolazione.
La funzione `getCollateralValueInUsd` del protocollo vittima, che calcola il valore massimo prendibile in prestito, si basa sul **prezzo spot** manipolato dall'oracolo.
Per completare l'attacco e rimborsare il flash loan, l'attaccante **converte una** parte degli USDC ottenuti in prestito nuovamente in WETH tramite Uniswap, **ripristinando le** riserve della pool e rimborsando il debito più il premium ad Aave.
L'infrastruttura di deployment utilizza **Hardhat con** forking della mainnet di Ethereum, consentendo l'interazione con contratti e account reali (come le pool Aave e Uniswap e una `"whale"` USDC per il finanziamento iniziale del protocollo vittima) in un ambiente **simulato ma** realistico.

4.1 Architettura del Protocollo

Il protocollo si articola in due **componenti principali**: il `LendingProtocol` (vittima) e il contratto `Attack` (attaccante), interagenti con **infrastrutture DeFi** consolidate sulla mainnet di Ethereum, simulate tramite forking.
Il `LendingProtocol` è un sistema di **prestito semplificato** che consente agli utenti di depositare WETH come collaterale e di prendere in prestito USDC. La sua architettura include `collateralToken` (WETH), `debtToken` (USDC) e un `priceOracle`. La vulnerabilità intrinseca risiede nella funzione `getCollateralValueInUsd`, che determina il **valore del** collaterale interrogando direttamente il **prezzo spot** da una pool `UniswapV2Pair (WETH/USDC)`. Questa dipendenza da un oracolo spot la rende **suscettibile a** manipolazioni.
Il contratto `Attack` è progettato per **orchestrare l'**exploit. La sua logica si basa sull'interazione con un `IPool (Aave V3)` per richiedere flash loan e con un `IUniswapV2Router02` per eseguire swap che **manipolano il** prezzo. Il flusso dell'attacco prevede il deposito di WETH nel `LendingProtocol`, seguito da un **flash loan** di WETH da Aave. Parte del WETH ottenuto viene scambiata con USDC tramite Uniswap per **manipolare il** prezzo dell'oracolo. Successivamente, l'attaccante sfrutta il **prezzo manipolato** per ottenere un prestito sottocollateralizzato di USDC dal `LendingProtocol`. Infine, per rimborsare il flash loan, una porzione degli USDC viene **riconvertita in** WETH tramite Uniswap, **ripristinando il** prezzo, e il debito verso Aave viene saldato.

4.2 `Attack.sol`

Il contratto `Attack.sol` è un **componente cruciale** nell'exploit, orchestrando una **complessa sequenza** di operazioni per manipolare un oracolo di prezzo spot e drenare fondi dal `LendingProtocol` vulnerabile. La sua architettura è progettata per interagire con diverse componenti chiave dell'ecosistema DeFi: `Aave V3` per i flash loan, `Uniswap V2 Router` per gli scambi di token, e il `LendingProtocol` vittima.
La funzione `attack()` è il **punto di** ingresso per l'esecuzione dell'exploit. Inizia **depositando un** ammontare predefinito di WETH (2 ether) come collaterale nel `LendingProtocol` vittima. Questo **stabilisce la** posizione iniziale dell'attaccante. Successivamente, viene richiesta una flash loan di 30 WETH da Aave. Questa quantità è **strategicamente scelta** per essere sufficiente a influenzare significativamente le riserve della pool Uniswap WETH/USDC.
Il **cuore dell'**attacco si svolge all'interno della callback `executeOperation()`, che viene invocata da Aave una volta erogato il flash loan. Qui, l'attaccante esegue le seguenti fasi:
Manipolazione del Prezzo: I 30 WETH ricevuti tramite flash loan vengono immediatamente scambiati con USDC sulla pool Uniswap WETH/USDC tramite la funzione `swapExactTokensForTokens` del router. Questo **swap massiccio** distorce il rapporto delle riserve nella pool, **gonfiando artificialmente** il prezzo di WETH rispetto a USDC, come percepito dall'oracolo spot del `LendingProtocol`.
Sfruttamento della Vulnerabilità: Con il prezzo di WETH **temporaneamente aumentato**, l'attaccante invoca la funzione `borrow` sul `LendingProtocol` vittima, richiedendo 5000 USDC. A causa del prezzo manipolato letto dall'**oracolo vul-**

nerabile del `LendingProtocol`, il sistema calcola erroneamente che il collaterale depositato dall'attaccante ha un valore molto più alto, **consentendo l'**erogazione di un prestito sottocollateralizzato.
Ripristino e Rimborso: Per ripagare il flash loan, l'attaccante converte gli USDC ottenuti dal prestito in WETH tramite un altro swap su Uniswap, **ripristinando in** gran parte il rapporto delle riserve alla sua condizione pre-attacco. Infine, il debito totale verso Aave (ammontare del flash loan più il premium) viene ripagato utilizzando il WETH appena riconvertito.
Il contratto `Attack` sfrutta quindi la **fungibilità dei** token e la **composabilità dei** protocolli DeFi per orchestrare una **complessa sequenza** di transazioni atomiche che garantiscono il profitto dall'operazione, evidenziando le **insicurezze derivanti** dall'affidamento a oracoli di prezzo spot.

4.3 `LendingProtocol.sol`

Il contratto `LendingProtocol.sol` rappresenta la **componente vulnerabile** del sistema, fungendo da protocollo di **prestito semplificato**. La sua architettura è basata su token standard ERC20 e un'interfaccia `Uniswap V2`, evidenziando le comuni **interazioni nei** sistemi DeFi.
Il contratto dichiara tre variabili di stato `immutable`: `collateralToken`, `debtToken` e `priceOracle`. Queste vengono inizializzate nel costruttore con gli indirizzi dei token `WETH`, `USDC` e della pool `WETH/USDC` di Uniswap V2, rispettivamente. È presente una `mapping` denominata `collateralBalances` per tenere traccia dei **saldi del** collaterale depositato da ciascun utente. Un'altra **costante cruciale** è `LOAN_TO_VALUE (LTV)`, fissata al 70%, che determina la **percentuale massima** del valore del collaterale che può essere presa in prestito.
La funzione `deposit(uint256 _amount)` permette agli utenti di trasferire WETH al contratto, **incrementando il** loro saldo di collaterale registrato nella `collateralBalances`. Il token viene trasferito dal `msg.sender` all'indirizzo del contratto `LendingProtocol` utilizzando `transferFrom`, richiedendo quindi una previa **approvazione del** token da parte dell'utente.
La **vulnerabilità critica** risiede nella funzione `borrow(uint256 _borrowAmount)` e nel suo dipendente `getCollateralValueInUsd(address _user)`. Quando un utente tenta di prendere in prestito, il protocollo calcola il valore del collaterale depositato invocando `getCollateralValueInUsd`. Questa funzione, etichettata esplicitamente come **"oracolo vulnerabile"**, recupera il **prezzo spot** di WETH rispetto a USDC direttamente dalle riserve (`reserve0`, `reserve1`) della `priceOracle` di Uniswap V2. Il calcolo del prezzo viene effettuato assumendo un **rapporto fisso** di decimali tra `reserve0` (USDC, 6 decimali) e `reserve1` (WETH, 18 decimali), e il **valore finale** del collaterale in USD viene derivato moltiplicando la quantità di collaterale dell'utente per questo prezzo spot.
Il `maxBorrowable` viene quindi calcolato applicando l'LTV al `collateralValueInUsd`. Una `require` assicura che l'ammontare richiesto (_borrowAmount) non superi questo massimo. Se la condizione è soddisfatta, il debito in USDC viene **erogato all'**utente tramite `debtToken.transfer(msg.sender, _borrowAmount)`. La dipendenza esclusiva da un oracolo di prezzo spot, **senza alcuna** misura di mitigazione contro la manipolazione (come la media mobile temporale o l'uso di oracoli decen-

tralizzati resistenti alla manipolazione), rende il `LendingProtocol` **suscettibile a** flash loan attacks che alterano temporaneamente il rapporto delle riserve nella pool di liquidità.

4.4 `deploy.js` e `attack.js`

Gli script `deploy.js` e `attack.js` sono **strumenti fondamentali** nell'ecosistema Hardhat che permettono l'orchestrazione e l'esecuzione dell'attacco di **manipolazione del** prezzo. Insieme ai contratti Solidity, essi formano un **ambiente di** test completo per dimostrare la vulnerabilità.
Il file `deploy.js` è responsabile del **dispiegamento (deployment)** dei contratti intelligenti sulla blockchain simulata (o reale). Inizia ottenendo un'istanza del **deployer**. Successivamente, deploya il `LendingProtocol`, il contratto vittima, fornendo gli indirizzi del `WETH`, dell'`USDC` e della pool `Uniswap WETH/USDC`. Una sezione cruciale di `deploy.js` è dedicata al **finanziamento del** `LendingProtocol` con USDC. Questo viene realizzato "impersonando" un indirizzo di `whale` sulla mainnet, consentendo allo script di trasferire una **quantità significativa** di USDC (100.000 USDC, 6 decimali) al contratto `LendingProtocol`. Questo passo è vitale per **simulare un** ambiente realistico in cui il protocollo vittima ha fondi sufficienti per **erogare prestiti**. Infine, `deploy.js` deploya il contratto `Attack`, passando l'indirizzo del `LendingProtocol` appena deployato, insieme agli indirizzi della pool `Aave` e del router `Uniswap`.
Una volta che i contratti sono stati deployati e il `LendingProtocol` è stato finanziato, entra in gioco lo script `attack.js`. Questo script è il **trigger dell'**attacco. Esso ottiene un'istanza del contratto `Attack` all'indirizzo **pre-configurato**. La sua funzione principale è invocare il metodo `attack()` sul contratto `Attack`, **avviando l'intera** sequenza di manipolazione del prezzo e di sfruttamento della vulnerabilità. Per simulare l'azione iniziale dell'attaccante che deposita collaterale, la chiamata alla funzione `attack()` include un **valore di** 2 Ether, che viene utilizzato dal contratto `Attack` per avviare il deposito di WETH nel `LendingProtocol`.
In sintesi, `deploy.js` **prepara il** terreno deployando e configurando i contratti e finanziando il protocollo vittima, mentre `attack.js` agisce come il **comando di** esecuzione che innesca la logica complessa definita in `Attack.sol` per sfruttare la vulnerabilità del `LendingProtocol`. Insieme, questi script facilitano una **dimostrazione pratica** e controllata di un attacco flash loan con **manipolazione dell'**oracolo.

Capitolo 5

Conclusioni

L'analisi e la **dimostrazione pratica** di questo attacco di manipolazione del prezzo evidenziano una **vulnerabilità fondamentale** nei protocolli DeFi che si affidano a oracoli di prezzo spot, in particolare quelli che derivano i prezzi direttamente dalle pool di liquidità di **Automated Market** Maker (AMM) come `Uniswap V2`. Il `LendingProtocol` simulato, pur essendo semplificato, incapsula un **difetto di** design critico: la sua funzione `getCollateralValueInUsd` utilizza un prezzo istantaneo (spot price) ottenuto dalle riserve della pool `WETH/USDC`. Questa dipendenza rende il protocollo **suscettibile a** "flash loan attacks", in cui un attaccante può temporaneamente distorcere il prezzo di un asset e sfruttare tale discrepanza.
L'attacco orchestrato tramite il contratto `Attack.sol` dimostra chiaramente come un **flash loan** da un protocollo come Aave possa essere utilizzato per manipolare il mercato. L'attaccante converte un ingente ammontare di WETH in USDC, **alterando significativamente** il rapporto delle riserve nella pool Uniswap. Questo porta l'oracolo del `LendingProtocol` a valutare erroneamente il collaterale dell'attaccante a un prezzo **gonfiato**, permettendo l'erogazione di un **prestito sottocollateralizzato** in USDC. La reversibilità dello swap consente all'attaccante di **ripristinare la** liquidità della pool e rimborsare il flash loan, consolidando il profitto derivante dagli USDC ottenuti illecitamente.
Questa dimostrazione sottolinea l'importanza critica di **oracoli di** prezzo robusti e resistenti alla manipolazione nei protocolli DeFi. Strategie di mitigazione includono l'utilizzo di oracoli basati su **medie temporali** ponderate (TWAP - Time-Weighted Average Price), che sono meno suscettibili a manipolazioni istantanee, o l'integrazione con oracoli **decentralizzati e** aggregati (come Chainlink) che derivano i prezzi da molteplici fonti on-chain e off-chain. La **sicurezza dei** fondi in un protocollo di prestito dipende direttamente dall'accuratezza e dalla resilienza del suo oracolo dei prezzi. Senza adeguate **misure contro** la manipolazione degli oracoli spot, i protocolli DeFi rimarranno vulnerabili a exploit che possono portare a significative **perdite finanziarie**.

Bibliografia

[1] Qin, K., Zhou, L., Gamito, P., Jovanovic, P., & Gervais, A. (2021). *An Empirical Study of DeFi Liquidations: Incentives, Risks, and Instabilities*. In ACM Internet Measurement Conference (IMC). https://doi.org/10.1145/3487552.3487811

[2] Qin, K., Zhou, L., & Gervais, A. (2020). *Attacking the DeFi Ecosystem with Flash Loans for Fun and Profit*. https://arxiv.org/abs/2003.03810

[3] Chainlink Labs. (2020). *Chainlink Price Feeds: Decentralized Oracle Networks*. https://blog.chain.link/chainlink-price-feeds-for-defi/

[4] Uniswap Labs. (2021). *Uniswap V2 Core and TWAP Oracle Documentation*. https://docs.uniswap.org/contracts/v2/concepts/core-concepts/oracles

[5] Brown, J. (2021). *Flash Loan Attacks Explained*. ConsenSys Blog. https://consensys.net/blog/security/flash-loan-attacks-explained/

[6] Lesaege, C., & Co. (2022). *Oracles in DeFi: Comparative Analysis and Security Implications*. DeFi Research Reports.

[7] Daian, P., Goldfeder, S., Kell, T., et al. (2019). *Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges*. USENIX Security Symposium. https://arxiv.org/abs/1904.05234

[8] Reiffers-Masson, A., Rusu, V., & Stainer, J. (2021). *Formal Verification of DeFi Protocols: A Survey*. https://arxiv.org/abs/2101.07769

[9] Gudgeon, L., Perez, D., Harz, D., Gervais, A., & Livshits, B. (2020). *The Decentralized Financial Crisis: Attacking DeFi*. In 2020 IEEE European Symposium on Security and Privacy Workshops. https://doi.org/10.1109/EuroSPW51379.2020.00026

[10] MChain Research Group. (2023). *Cross-Platform Oracle Risk Models and Governance Solutions in DeFi*. Whitepaper, MChain Labs.

[11] Chen, M., Monnot, B., Perez, D., & Livshits, B. (2021). *A Systematization of Knowledge: DeFi Attacks*. arXiv:2106.06389 [cs.CR]. https://arxiv.org/pdf/2106.06389.pdf

[12] Research Imperium. (2021). *SoK DeFi Attacks Dataset*. GitHub repository. https://github.com/Research-Imperium/SoKDeFiAttacks

[13] Research Imperium. (2021). *2020 bZx Flash Loan Attack JSON Description*. In SoK DeFi Attacks Dataset. https://github.com/Research-Imperium/SoKDeFiAttacks/blob/main/dataset/attacks/2020_bzx_flashloan_attack_1.json