ELTE
EÖTVÖS LORÁND
UNIVERSITY

EÖTVÖS LORÁND UNIVERSITY - FACULTY OF INFORMATICS

CYBERSECURITY LAB I REPORT
COURSE CODE: IPM-22FKBSCLAB1

Smart Contracts Vulnerabilities

Student name:
LEMTAFFAH Khalil Abdellah
Supervisor:
Mr. SERES István András

Professor:
DR. LIGETI Péter

January 15, 2023

Contents

1 Abstract

First of all, the name of this project was changed from "Privacy-Enhancing Technologies For Cryptocurrencies" to "Smart Contracts Vulnerabilities", since that's the focus of our lab and it's a more broader and interesting topic.

The goal of this project is to have a complete overview of the techniques that are built on top cryptocurrencies to enhance financial privacy. These techniques include but are not limited to coin mixing, stealth addresses, confidential transactions, CoinSwaps, etc. Ideally, we could develop a novel privacy-enhancing technique that is covert in the sense that no one could tell just by looking at the blockchain that people are issuing transactions to enhance their privacy. The end goal would be to design a CoinSwap protocol for Ethereum. A new research or vulnerability discovery is not expected from this lab because one of this project's goals is to get to know these notions and branches of cryptocurrencies better.

2 Introduction

Many resources were used to learn different technologies that are related to the blockchain world, especially the Ethereum distributed ledger. As said above in the abstract section, the main aim of the lab is to know more about the technologies and learn different concepts, such as developing a smart contract and deploying it on the Ethereum blockchain.

A timeframe of 3 months was given to learn the new concepts and apply them in a lab, at the same time document them in a report and conclude with a 30 minutes presentation at the end of the semester.

3 Goal

The goal of this lab is to develop a smart contract from scratch, using the programming language Solidity, intentionally include 3 - 5 bugs in the code, then security audit the contract using a web3 scanner, and mitigate those bugs to have a secure smart contract finally, that can be later successfully deployed to the Ethereum Mainnet.

The whole purpose of this report is the learn how to avoid some serious vulnerabilities that smart contract developers always tend to forget about.

ethereum

Figure 1: Ethereum logo

4 The blockchain world

4.1 What is the blockchain?

It's amazing how computers transitioned from a centralized network to a decentralized one, where we can write and hold thousands of records in a matter of minutes, sometimes in seconds. These records contain a cryptographic hash of the previous computer that was in the line, a timestamp, and transaction data. These computers combined together form what we call a block, and a big amount of these blocks, communicating together form a blockchain or a distributed ledger.

So when we are dealing with a lot of information and data transactions, we start to think about the integrity of the exchanged information, so that it's going to be hard for the outsiders and bad vectors to alter it and perform a blockchain outage. Finally, the integrity and chronological order of the blockchain are enforced with cryptography. In this manner, we can say that the blockchain is resistant to data alteration.

4.2 Ethereum

One of the most famous decentralized blockchains in web3 is the Ethereum platform, which is open-source and runs under the command of applications that are programmed to avoid any possibility of downtime, fraud, third-party interference, and so on. These programs are called smart contracts, and they are developed in one of the programming languages, Solidity for example.

We can also use Ethereum to build a whole application and deploy it on the Ethereum blockchain. Since these applications run in a decentralized way, they are called decentralized applications (or dApps in short). These dApps can contain whatever the brain can imagine, and we can also include coin transactions and make them interact with real currency.

4.3 Smart contracts

The transactions on the blockchain are done following an agreement between buyers and sellers, this agreement plays the role of a smart contract, which is a self-executing contract with some terms. Those terms are lines of code written by developers who can program in the Solidity language. For instance, the contract code resides on the Ethereum blockchain network.

Solidity is a statically-typed, contract-oriented programming language for implementing smart contracts, these contracts run on an Ethereum Virtual Machine (EVM) which takes the compiled bytecode, and executes it. Solidity is the most popular and simplest language for writing smart contracts on the Ethereum platform, which is going to be the scope of this lab.

5 Methodology

The given timeframe for learning Ethereum technologies, applying it, and building a smart contract, with learning its security best practices and trying to break it was a big challenge to surpass. Thanks to the freely available content on the internet, an intermediate level of knowledge was built in this period, where a lot of websites and tutorials were consumed and used as a resource.

5.1 Learning phase

A methodology and a plan were needed to start with, in order to keep the deadline and practice as much as possible. The first step was to create a GitHub repository [1] and invite my supervisor to it, in order to keep track of my progress and check out the commits. The repository was private and will be public after presenting the lab.

5.1.1 Resources

After creating the repository I was looking for a blog post written by a previous researcher/developer to guide me through the best resources, and I found this one [2] which demonstrates some great resources about how to get started into learning smart contracts development in Solidity language, and how to practice its security vulnerabilities, as well as some bug bounty platforms to earn money for potentially found bugs in the wild.

5.1.2 CryptoZombies

In order to digest information in a fast way, a "gamified" or "fun" platform was needed to learn something new for the first time. Luckily, the blog post mentioned that there is a platform to learn Solidity basics which is called CryptoZombies [3].

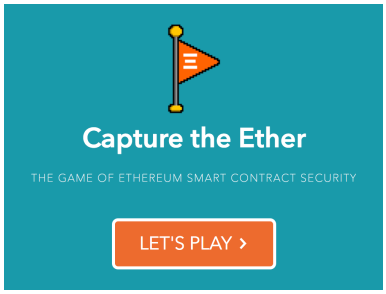
CryptoZombies is a free, online course that teaches the basics of smart contract programming in Solidity. It is designed to be fun and educational, with a gamified experience that allows users to learn by building their own blockchain-based game. The course covers the fundamentals of Solidity, including data types, functions, and control structures, as well as more advanced topics like contract inheritance and the Ethereum Virtual Machine (EVM). By the end of the course, users will have built their own fully functional ERC-721 non-fungible token that they can use to represent their very own in-game CryptoZombie.

5.1.3 Ethereum Tutorials

The organization of Ethereum created a tutorials section [4] where you can learn a lot of technologies and concepts related to the Ethereum blockchain. This website was used to learn how to create a hello world smart contract in Solidity language, and deploy it to a test network using Alchemy.

[1] https://github.com/splint3rsec/CybersecLab
[2] https://blog.yeswehack.com/yeswerhackers/getting-started-smart-contract-bug-bounty/
[3] https://cryptozombies.io/
[4] https://ethereum.org/en/developers/tutorials/

5.2 Smart contracts Security



While learning the basics, it turns out that smart contract security is the most important aspect of the whole field. Where developers should take careful attention to the business logic, and try to not make small mistakes that will lead to expensive costs in the future.

In that manner, security training was needed to learn the security vulnerabilities, so that they're going to be replicated in the lab's smart contract. In the blockchain, we can also find a lot of Capture The Flag (CTF) platforms, where we can practice real-world scenarios and hunt for smart contract vulnerabilities for education purposes. One of the used platforms to learn Solidity security were CaptureTheEther [5] and Ethernaut [6] by Open Zeppelin, which is an open-source framework for building secure smart contracts.

In addition to playing CTFs, we can also refer to the SWC Registry [7] and read about the different smart contract weaknesses, their implementation, and the ways to mitigate those vulnerabilities.

5.2.1 Quote on smart contracts security

One of the marking quotes on the field of securing smart contracts, which explains in a nutshell how developers should actually deal with it, and consider it seriously:

> *"smart contract code is unforgiving. Every bug can lead to monetary loss. You should not treat smart contract programming the same way as a general-purpose programming. Writing DApps in Solidity is not like creating a web widget in JavaScript. Rather, you should apply rigorous engineering and software development methodologies, as you would in aerospace engineering or any similar unforgiving discipline. Once you "launch" your code, there's little you can do to fix any problems."*
>
> - Andreas M. Antonopoulos, Gavin Wood, *Mastering Ethereum*[8]

5.3 Application

After taking note of all the concepts and digesting the basics, I decided to start applying the knowledge of creating a vulnerable smart contract using Solidity, as well as detecting its security bugs and trying to fix them at the end.

The whole process seems to be very complex to finish, because just developing a smart contract can take a lot of time, nevertheless making it vulnerable and trying to fix it. But a simple smart contract is going to be used.

[5] https://capturetheether.com/
[6] https://ethernaut.openzeppelin.com/
[7] https://swcregistry.io/
[8] https://github.com/ethereumbook/ethereumbook

6 The smart contract

6.0.1 Idea

The idea of our smart contract is to create a decentralized ticket handling marketplace, where the users can perform a lot of things with it, such as:

- Buying / Selling tickets
- Participating in a lucky draw where they can win a free ticket
- Giving a ticket to another user

As mentioned before, the contract is not going to be complex and/or implement hard concepts, due to the time restriction. For such, we will just call normal methods and keep the algorithm and the business flow simple.

6.0.2 Attacking the contract

After creating thee contract, we will implement pitfalls to make it more vulnerable to external attacks, such as when another smart contract interacts with `ticketsMarketplace`, it is going to be easy for it to loot tokens and drain money from the dApp. Implemented bugs are but not limited to:

- Reentrancy [9]
- Integer Over/Underflow [10]
- Unprotected SELFDESTRUCT Instruction [11]

6.0.3 Used static analyzers

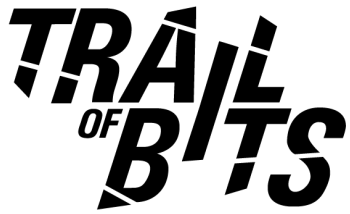


Figure 2: Trail of Bits logo

In this part of the lab, we will use two famous static analyzers: Slither[12] and Mythril [13].

Those tools are Solidity static analyzers and fuzzers that can check the code for common vulnerabilities, from known sources such as the SWC registry or contributions from other blockchain security researchers.

One of the best blockchain tools are created and handled by Trail Of Bits [14] Organization, which have a subgroup that handles everything related to blockchain security, called crytic. [15] But still, even with these tools, we will still have to perform a dynamic analysis manually and check out if they missed some parts or not. Follow up in the Conclusion section.

[9] `https://swcregistry.io/docs/SWC-107`
[10] `https://swcregistry.io/docs/SWC-101`
[11] `https://swcregistry.io/docs/SWC-106`
[12] `https://github.com/crytic/slither`
[13] `https://github.com/ConsenSys/mythril`
[14] `https://github.com/trailofbits`
[15] `https://github.com/crytic`

6.1 Source Code

6.1.1 Implementation

This is the implementation of the smart contract of our idea. Imported contracts are available on Github[16].

```
1  // SPDX-License-Identifier: MIT
2
3  /**
4  * @dev The ticketsMarketplace is a contract that plays the role of a
       stocks market
5  * in a decentralized way, where the buyers can take a certain amount of
       tickets
6  * depending on their market value and the buyer's budget. As well as a
       person can sell
7  * his ticket later for a higher/lower price. It depends on the ticket
       value...
8  * There's also a lucky draw function where each user may win a ticket
       and withdraw it
9  * after a certain amount of time, decided by a parameter entered by the
       address owner.
10 */
11
12 pragma solidity ^0.8.0;
13
14 import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
15 import "@openzeppelin/contracts/token/ERC721/extensions/ERC721Enumerable
       .sol";
16
17 contract ticketsMarketplace is ERC721 {
18
19     uint randNonce = 1337;
20     uint _modulus = 10**5;
21     bool canTakePrize = false;
22
23     struct Ticket {
24         address owner;
25         bool isForSale;
26         uint value;
27     }
28
29     constructor() public ERC721("Ticket", "TIC") {}
30
31     Ticket[] public tickets;
32
33     function supportsInterface(bytes4 interfaceId) public view override
           returns (bool) {
34         return super.supportsInterface(interfaceId);
35     }
36
37     mapping (uint => Ticket) public tokens;
38     mapping (uint => uint) public ticketValues;
39     mapping (address => uint) userBalance;
40
41     event ticketSold(uint ticketId);
```

[16] https://github.com/OpenZeppelin/openzeppelin-contracts/tree/master/contracts

```
42      event ticketListed(uint ticketId);
43      event luckyDrawWinner(address);
44
45      function buyTicket(uint ticketId, address buyer) public payable {
46          Ticket storage ticket = tokens[ticketId];
47          uint balance = address(this).balance; //Intended bug
48          require(ticket.isForSale, "This ticket is not for sale.");
49          require(balance >= ticketValues[ticketId], "You have
                insufficient amount of money.");
50          ticket.isForSale = false;
51          ticket.owner = msg.sender;
52          _transfer(msg.sender, buyer, ticketId);
53          emit ticketSold(ticketId);
54      }
55
56      function sellTicket(uint ticketId, uint val) public {
57          require(ownerOf(ticketId) == msg.sender, "Sorry, you can't sell
                this ticket.");
58          Ticket storage ticket = tokens[ticketId];
59          ticket.isForSale = true;
60          ticket.value = val;
61          ticketValues[ticketId] = ticket.value;
62          emit ticketListed(ticketId);
63      }
64
65      function transferTicket(uint ticketId, address newOwner) public {
66          require(ownerOf(ticketId) == msg.sender, "You are not allowed to
                transfer the ticket.");
67          _transfer(msg.sender, newOwner, ticketId);
68      }
69
70      function random() internal returns (uint) {
71          randNonce += 1; //intentional
72          return uint(keccak256(abi.encodePacked(block.timestamp, msg.
                sender, randNonce))) % _modulus;
73      }
74
75      function getIndex(address walletAddress) view private returns (uint)
            {
76          for(uint i = 0; i < tickets.length; i++) {
77              if(tickets[i].owner == walletAddress) {
78                  return i;
79              }
80          }
81      }
82
83      function withdraw(address payable walletAddress) payable public {
84          uint i = getIndex(walletAddress);
85          require(address(this).balance >= tickets[i].value, "Insufficient
                contract balance to pay the ticket value");
86          require(msg.sender == tickets[i].owner);
87          (bool success, ) = msg.sender.call{value:userBalance[msg.sender
                ]}("");
88          userBalance[msg.sender] = 0;
89      }
90
```

```
91      function withdrawPrize(address payable walletAddress) payable public
            {
92          withdraw(walletAddress); // bug, anyone can call it.
93      }
94
95      function luckyDraw(uint duration) public {
96          require(duration > 0, "The duration can't be 0");
97          address[] memory ticketHolders = new address[](balanceOf(address
                (0)));
98          for (uint i = 0; i < ticketHolders.length; i++) {
99              ticketHolders[i] = ownerOf(i);
100         }
101         uint randomIndex = random();
102         address winner = ticketHolders[randomIndex];
103         require(block.timestamp > duration);
104         payable(winner).transfer(tickets[getIndex(winner)].value);
105         withdrawPrize(payable(winner));
106         emit luckyDrawWinner(winner);
107     }
108
109     function killMe() public {
110         selfdestruct(payable(msg.sender));
111     }
112
113 }
```

Once our smart contract was created and developed, we can then go ahead and perform the security analysis phase, where we will run the previously mentioned tools, and we will dynamically analyze the contract, so that potential bugs are going to be spotted before deploying the contract into an Ethereum network. Goerli test network was used in the lab.

6.1.2 Contract visualisation

In order to have a graphical representation of the contract, we can use Surya [17] by Consensys, and export a graph like the following:

[17] https://github.com/ConsenSys/surya

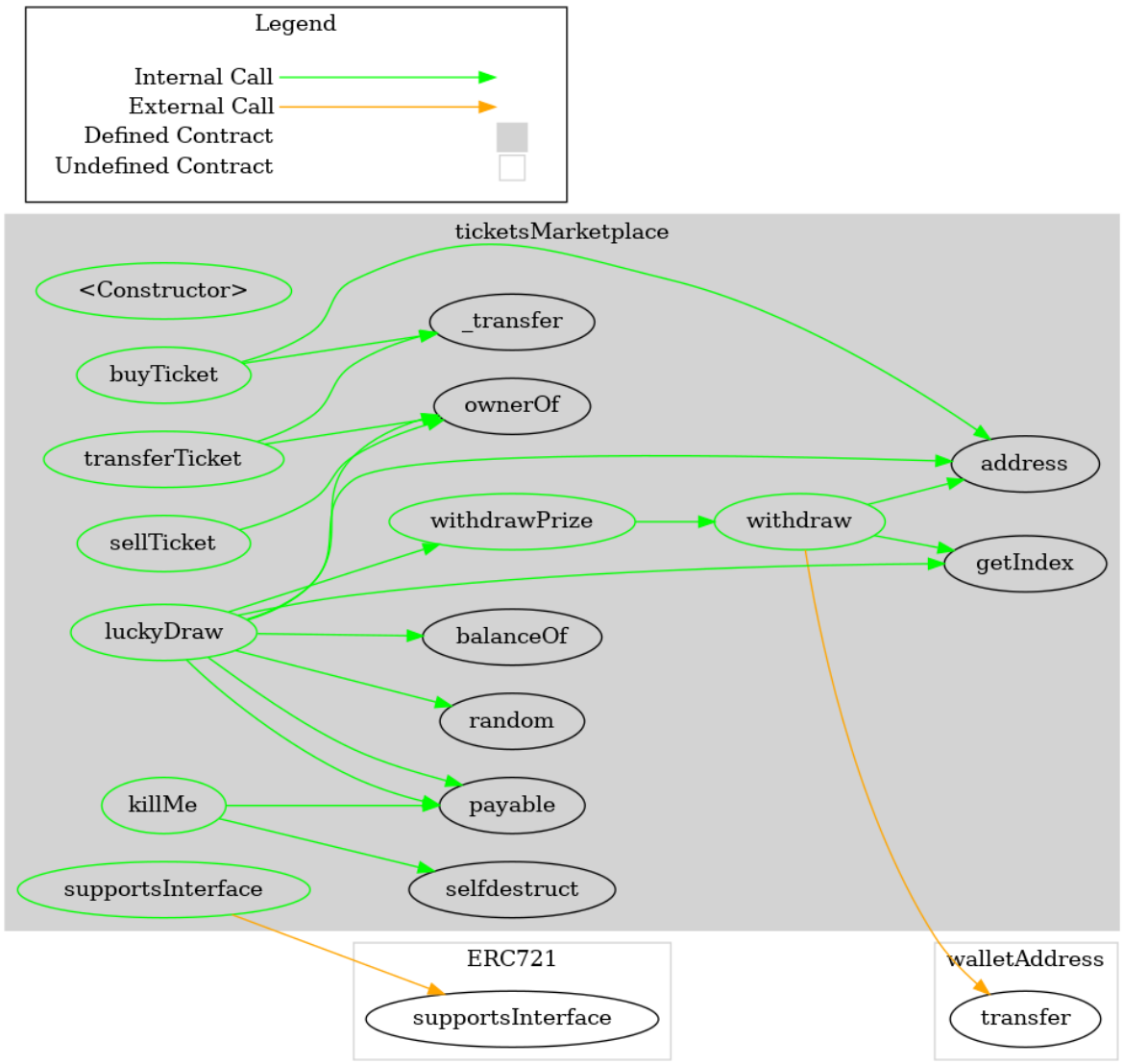


Figure 3: ticketsMarketplace Surya graph

You will notice that the graph isn't that complex, since the goal of this lab is definitely not to develop a decentralized application and deploy it on the Ethereum mainnet, but instead to learn Solidity skills and know its security aspects.

Certainly, this contract includes most of the Solidity functionalities and implementations, as well as ERC tokens (ERC721).

7 Implemented vulnerabilities

7.1 Usage of insecure Pseudo Random Number Generator

`keccak256 Keccak256 (SHA-3)` is considered to be a very secure hashing algorithm, and it is not currently known to be breakable. Keccak-256 was selected as the winner of the National Institute of Standards and Technology (NIST) competition to become the new SHA-3 standard, this competition was designed to find a new hash function that would be resistant to both known and future attacks.

That being said, no cryptographic algorithm is completely unbreakable and it's always possible that new attack methods will be discovered in the future. However, as of now, Keccak256 is considered to be a very secure hashing algorithm and is widely used in the blockchain and cryptocurrency industry.

Sometimes with high computation powers, a candidate can calculate the outcome of a function and be the first to submit the answer in the blockchain, in order to get the final prize. Also, usage of predicted variables such as `block.timestamp`, `now` and `blockhash` are insecure to use.

The `ticketsMarketplace` contract uses this vulnerability in the following code snippet:

```
1 function random() internal returns (uint) {
2     randNonce += 1; //intentional
3     return uint(keccak256(abi.encodePacked(block.timestamp, msg.sender,
          randNonce))) % _modulus;
4 }
```

The problem here is that `block.timestamp` is a bad variable to use for a modulo, since these can be influenced by miners to some extent so they should be avoided.

For more information: SWC-120 [18]

7.2 Reentrancy

One of the most dangerous and unnoticed vulnerabilities in the blockchain world is reentrancy. A very famous hack on the DAO oraganization happened on 2016 [19], where a black hat hacker exploited this vulnerability to drain $150M worth of money, right after 3 months after launching the project.

Our contract implements this vulnerability, where an attacker can recall a withdraw function, to get all the tokens in the smart contract, without resetting its balance to 0.

```
1 function withdraw(address payable walletAddress) payable public {
2         uint i = getIndex(walletAddress);
3         require(address(this).balance >= tickets[i].value, "Insufficient
              contract balance to pay the ticket value");
4         require(msg.sender == tickets[i].owner);
5         (bool success, ) = msg.sender.call{value:userBalance[msg.sender
              ]}("");
6         userBalance[msg.sender] = 0;
7 }
```

[18] https://swcregistry.io/docs/SWC-120
[19] https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee

In order to mitigate this issue, we have to always change the state before calling the value for the `msg.sender`. In this case, an attacker can't drain tokens from the smart contract.

For more information: SWC-107 [20]

7.3 Integer Overflow - Underflow

Very easy to spot, but needs some time to get exploited, an attacker have to either overflow or underflow a variable in order to receive the initial state of the variable.

In our case, we used it like the following:

```
1 function random() internal returns (uint) {
2     randNonce += 1; //intentional
3     return uint(keccak256(abi.encodePacked(block.timestamp, msg.sender,
          randNonce))) % _modulus;
4 }
```

We can simply mitigate the bug by importing an external library from OpenZeppelin called `SafeMath`[21] and use its functions for arithmetic operations, since they include a check for mathematical overflows and underflows.

For more information: SWC-101 [22]

7.4 Unprotected SELFDESTRUCT function

Sometimes developers forget to add an access control on functions that contain a SELFDETRUCT call. In this manner, an attacker from the outside world can easily call that function and it will be shut down, as well as all the tokens that this contract holds will be transferred to the attacker's wallet.

The vulnerable snippet in the contract:

```
1 function killMe() public {
2         selfdestruct(payable(msg.sender));
3 }
```

For fixing this vulnerability, we can add a `onlyOwner` modifier to restrict access to the function, so that it won't be called by someone other than the contract's owner.

For more information: SWC-106 [23]

7.5 Reflection

Of course, these are not the only intentional bugs, where we only used our business logic thinking to produce those vulnerabilities, but on the other hand, there are undiscovered ones that neither we nor the static analyzers can determine.

The next section is gonna be about using the previously mentioned tools and their results.

[20] https://swcregistry.io/docs/SWC-107
[21] https://docs.openzeppelin.com/contracts/3.x/api/math
[22] https://swcregistry.io/docs/SWC-101
[23] https://swcregistry.io/docs/SWC-106

8 Contract auditing & results

A best practice for smart contract developers is to not rely on only one tool to be 100% sure, instead, developers should use more than 2 tools or preferably invest in a professional smart contracts security auditing team, to perform this operation and provide the developers team with a complete report, that contains the discovered bugs and the future potential ones.

8.1 Mythril

One of the best tools to spot bugs in smart contracts, Mythril is a security analysis tool for EVM bytecode, that gives a detailed overview about the bug class and its severity.

Unfortunately, this tool wasn't able to spot all the bugs that we mentioned earlier, and the results were shown to the console after ∼5 minutes of the execution, which is considered slow in the computer science world.

The vulnerability that Mythril was able to spot is the `Unprotected SELFDESTRUCT function`, which is a very obvious one compared to the other ones present in the smart contract. This makes us ask a lot of questions, one of them is maybe the smart contract wasn't developed in a way that makes the user interacts with it, in other words, while developing the contract, the bugs implementation was the only concern to solve, not the logic of the decentralized application and how to make it deployment ready.

```
┌──(venv)(splinter㉿DESKTOP-AASFCIM)-[~/ELTE/cyseclab/playground]
└─$ myth a contract.sol
==== Unprotected Selfdestruct ====
SWC ID: 106
Severity: High
Contract: ticketsMarketplace
Function name: killMe()
PC address: 4645
Estimated Gas Usage: 209 - 304
Any sender can cause the contract to self-destruct.
Any sender can trigger execution of the SELFDESTRUCT instruction to destroy
eview the transaction trace generated for this issue and make sure that app
--------------------
In file: contract.sol:111

selfdestruct(payable(msg.sender))

--------------------
Initial State:

Account: [CREATOR], balance: 0x1, nonce:0, storage:{}
Account: [ATTACKER], balance: 0x1, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: , decoded_data: , value: 0x0
Caller: [ATTACKER], function: killMe(), txdata: 0xb603cd80, value: 0x0
```

Figure 4: Mythril output on ticketsMarketplace

8.2 Slither

Slither on the other hand is only specialized for Solidity language, and it runs a suite of vulnerability detectors, prints visual information about contract details, and provides an API to easily write custom analyses.

It shows the output in a colorful way, such that the red ones are very highly classified and should be fixed as soon as possible, yellow ones are a security risk and should be considered, and lastly, green ones are treated as warnings, fixing them will be good for the contract on the long run.

When we ran it on our contract, it took less than 5 seconds to come up with the results and show us many more spotted bugs than Mythril, such as `SELFDESTRUCT function`, `Reentrancy` and `Insecure usage of PRNG`.

Also, Slither includes a link to their GitHub repository, where they explain each single bug class, an example, and how to fix it.

```
ticketsMarketplace.random() (contract.sol#70-73) uses a weak PRNG: "uint256(keccak256(bytes)(abi.encodePacked(block.timestamp
,msg.sender,randNonce))) % _modulus (contract.sol#72)"
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#weak-PRNG

Reentrancy in ticketsMarketplace.withdraw(address) (contract.sol#83-90):
        External calls:
        - (success) = msg.sender.call{value: userBalance[msg.sender]}() (contract.sol#88)
        External calls sending eth:
        - walletAddress.transfer(tickets[i].value) (contract.sol#87)
        - (success) = msg.sender.call{value: userBalance[msg.sender]}() (contract.sol#88)
        State variables written after the call(s):
        - userBalance[msg.sender] = 0 (contract.sol#89)
        ticketsMarketplace.userBalance (contract.sol#39) can be used in cross function reentrancies:
        - ticketsMarketplace.withdraw(address) (contract.sol#83-90)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities

ticketsMarketplace.killMe() (contract.sol#110-112) allows anyone to destruct the contract
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#suicidal

ticketsMarketplace.tickets (contract.sol#31) is never initialized. It is used in:
        - ticketsMarketplace.getIndex(address) (contract.sol#75-81)
        - ticketsMarketplace.withdraw(address) (contract.sol#83-90)
        - ticketsMarketplace.luckyDraw(uint256) (contract.sol#96-108)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-state-variables
```

Figure 5: Slither output on ticketsMarketplace

8.3 Personal review

Personally, I would prefer to use Slither to spot bugs in Solidity smart contracts, than using Mythril. Because Slither is written and open-sourced by one of the big pioneers in the blockchain industry (Trail of Bits), and it is only focused on the Solidity programming language. On the other hand, Mythril will be more useful in the case of other programming languages that run on EVM-compatible blockchains, such as Hedera, Quorum, Vechain, Roostock, Tron and others...

Consensys recommends smart contracts developers to check out their security tools bundle [24] which is going to be useful for spotting bugs and avoiding them.

[24] `https://github.com/muellerberndt/awesome-mythx-smart-contract-security-tools`

9 Fixing the vulnerabilities

In order to fix the vulnerabilities, we can just refer to the links mentioned in the tool's output, or we go to the SWC registry and read more about the vulnerability class, then try to implement the fix. This is what our `ticketsMarketplace` contract looks like after the fixes:

```
1  // SPDX-License-Identifier: MIT
2
3  /**
4  * @dev The ticketsMarketplace is a contract that plays the role of a
       stocks market
5  * in a decentralized way, where the buyers can take a certain amount of
       tickets
6  * depending on their market value and the buyer's budget. As well as a
       person can sell
7  * his ticket later for a higher/lower price. It depends on the ticket
       value...
8  * There's also a lucky draw function where each user may win a ticket
       and withdraw it
9  * after a certain amount of time, decided by a parameter entered by the
       address owner.
10 */
11
12 pragma solidity ^0.8.0;
13
14 import "./node_modules/@openzeppelin/contracts/token/ERC721/ERC721.sol";
15 import "./node_modules/@openzeppelin/contracts/token/ERC721/extensions/
       ERC721Enumerable.sol";
16 import "./node_modules/@openzeppelin/contracts/utils/math/SafeMath.sol";
17 import "./node_modules/@openzeppelin/contracts/access/Ownable.sol";
18
19 contract ticketsMarketplace is ERC721, Ownable {
20
21     using SafeMath for uint256;
22
23     uint randNonce = 1337;
24     uint _modulus = 10**5;
25     bool canTakePrize = false;
26
27     struct Ticket {
28         address owner;
29         bool isForSale;
30         uint value;
31     }
32
33     Ticket admin;
34
35     function setAdmin() public {
36         admin = Ticket(address(this), false, 100);
37     }
38     Ticket[] public tickets;
39
40     constructor() public ERC721("Ticket", "TIC") {}
41
42
```

```
43      function supportsInterface(bytes4 interfaceId) public view override
            returns (bool) {
44          return super.supportsInterface(interfaceId);
45      }
46
47      mapping (uint => Ticket) public tokens;
48      mapping (uint => uint) public ticketValues;
49      mapping (address => uint) userBalance;
50
51      event ticketSold(uint ticketId);
52      event ticketListed(uint ticketId);
53      event luckyDrawWinner(address);
54
55      function buyTicket(uint ticketId, address buyer) public payable {
56          Ticket storage ticket = tokens[ticketId];
57          uint balance = address(this).balance; //Intended bug
58          require(ticket.isForSale, "This ticket is not for sale.");
59          require(balance >= ticketValues[ticketId], "You have
                insufficient amount of money.");
60          ticket.isForSale = false;
61          ticket.owner = msg.sender;
62          _transfer(msg.sender, buyer, ticketId);
63          emit ticketSold(ticketId);
64      }
65
66      function sellTicket(uint ticketId, uint val) public {
67          require(ownerOf(ticketId) == msg.sender, "Sorry, you can't sell
                this ticket.");
68          Ticket storage ticket = tokens[ticketId];
69          ticket.isForSale = true;
70          ticket.value = val;
71          ticketValues[ticketId] = ticket.value;
72          emit ticketListed(ticketId);
73      }
74
75      function transferTicket(uint ticketId, address newOwner) public {
76          require(ownerOf(ticketId) == msg.sender, "You are not allowed to
                transfer the ticket.");
77          _transfer(msg.sender, newOwner, ticketId);
78      }
79
80      function random() internal returns (uint) {
81          uint32 max = 100000;
82          randNonce = randNonce.add(1);
83          uint256 salt =  block.timestamp * randNonce;
84          uint256 x = salt * 100 / max;
85          uint256 y = salt * block.number / (salt % 5);
86          uint256 seed = block.number / 3 + (salt % 300) + y;
87          uint256 h = uint256(blockhash(seed));
88          // Random number between 1 and max
89          return uint256((h / x)) % max + 1;
90      }
91
92      function getIndex(address walletAddress) private returns (uint) {
93          tickets[0] = admin;
94          for(uint i = 1; i < tickets.length; i++) {
95              if(tickets[i].owner == walletAddress) {
```

```
96                  return i;
97              }
98          }
99      }
100
101     function withdraw(address payable walletAddress) payable public {
102         uint i = getIndex(walletAddress);
103         require(address(this).balance >= tickets[i].value, "Insufficient
                contract balance to pay the ticket value");
104         require(msg.sender == tickets[i].owner);
105         userBalance[msg.sender] = 0;
106         walletAddress.transfer(tickets[i].value);
107     }
108
109     function withdrawPrize(address payable walletAddress) payable public
            {
110         withdraw(walletAddress); // bug, anyone can call it.
111     }
112
113     function luckyDraw(uint duration) public {
114         require(duration > 0, "The duration can't be 0");
115         address[] memory ticketHolders = new address[](balanceOf(address
                (0)));
116         for (uint i = 0; i < ticketHolders.length; i++) {
117             ticketHolders[i] = ownerOf(i);
118         }
119         uint randomIndex = random();
120         address winner = ticketHolders[randomIndex];
121         require(block.timestamp > duration);
122         payable(winner).transfer(tickets[getIndex(winner)].value);
123         withdrawPrize(payable(winner));
124         emit luckyDrawWinner(winner);
125     }
126
127     function killMe() public onlyOwner {
128         selfdestruct(payable(msg.sender));
129     }
130
131 }
```

10 Conclusion

Smart contracts security play a very big role in the blockchain world, and a small mistake may lead to critical consequences that will cost the maintainers a lot of money and resources. Millions of dollars were stolen from different cryptocurrencies in the past few years, which makes the companies look for and invest in the best security auditors to prevent similar losses in the future.

Using static analyzers and relying on them isn't always a solution, sometimes those tools can miss the most critical bugs in the contract, and not warn about it. That's why it is recommended to perform a dynamic analysis before pushing code and deploying the smart contract. That was noticed after comparing the tools execution (Mythril and Slither), then only getting 1 vulnerability warning compared to 5 vulnerabilities on the other hand. Still, that's not everything and a manual code review is needed for the contract.

To conclude, this lab was a very interesting topic to dig in and learn more about cryptocurrencies and how they work, especially the Ethereum blockchain. A basic to intermediate level of Smart contracts development using Solidity language was gained during this lab, as well as their security implementations and how to spot potential bugs, either using tools for static analysis, or dynamically analyze it with logical thinking and test deployments.