National College of Ireland

How to Improve Security of Smart Contracts written in Solidity in Blockchain by Detecting Reentrancy Vulnerability

MSc Research Project
MSc in Cyber Security

Rahul --
Student ID: 20243804

School of Computing
National College of Ireland

Supervisor: Prof. Michael Prior

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Rahul --		
Student ID:	20243804		
Programme:	MSc in Cyber Security	**Year:**	2021-22
Module:	MSc Research Project/Internship		
Supervisor:	Prof. Michael Prior		
Submission Due Date:	15/08/2022		
Project Title:	How to Improve Security of Smart Contracts written in Solidity in Blockchain by Detecting Reentrancy Vulnerability		
Word Count:	6125 **Page Count:** 29		

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Rahul --

Date: 15/08/2022

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

How to Improve Security of Smart Contracts written in Solidity in Blockchain by Detecting Reentrancy Vulnerability

Rahul --
20243804

Abstract

As we know that the use of Blockchain is growing and so is the use of solidity a Programming language used for creating agreements on the Ethereum platform but is it really safe to use solidity. In this paper, I have researched on the main vulnerability of the programming language used for Smart contracts in the Ethereum environment (i.e., solidity) which is reentrancy, and my research proposal is that I have tried to provide a novel solution/ verification reentrancy detection tools. I have verified two important reentrancy vulnerability detection tools, Slither and Mythril, in searching for a new and ingenious solution. Since the tools were written in a programming language version that is outdated and throwing errors, I debugged the code of these two tools and some other tools too. I also experimented with brownie and python console on how to build a smart contract, deploy them and interact with other smart contracts to learn how the smart contracts work. I verified Slither and Mythril on different platforms, Kali Linux, Ubuntu, and Windows OS with various different datasets.

1 Introduction

As we know in that the use of Blockchain is growing and so is the use of solidity a Programming language used for creating agreements on the Ethereum platform but is it safe to use solidity. In this paper, I have researched on the main vulnerability of the programming language used for Smart contracts in Ethereum environment (i.e., solidity) which is reentrancy, and my research proposal is that I have tried to provide a novel solution to this existing vulnerability by merging other solutions for this vulnerability. I have verified two important reentrancy vulnerability detection tools, Slither and Mythril, in searching for a new and ingenious solution. Since the tools were written in a programming language version that is outdated and throwing errors, I debugged the code of these two tools and some other tools too. I also experimented with brownie and python console on how to build a smart contract, deploy them and interact with other smart contracts to learn how the smart contracts work. I verified Slither and Mythril on different platforms, Kali Linux, Ubuntu, and Windows OS with various different datasets.
Solidity (Dannen, 2017) is one of the most prominent statically typed programming languages used on the Ethereum environment to create smart contracts, its name was coined by Gavin Wood in 2014 and was developed in the near future by a team created for solidity which was a part of Blockchain project. Unlike ECMAScript, Solidity uses static typing and has variable return types, so it is familiar to programmers.

Also, the environment in which solidity is used i.e., Ethereum is a frontrunner of web 3.0 (Groce, 2019) and one of the largest blockchain networks. Regardless of the fact that Ethereum is a commonly utilized cryptocurrency that allows for secure currency transactions. There's a whole number of apps that make use of the Blockchain's distributed nature to handle activities.
The motive behind the research on the chosen topic is because solidity, a statically typed programming language first designed in 2015, has risen in popularity because multiple inheritance is provided by it, stores numerous variable functions for arranging and representing smart contracts. if the third party is involved the operations are very reliable and safe, and it can be used to create a significant number of smart contracts. Blockchain, a relatively older technology with little progress in the sector, has recently become one of the most significant breakthroughs in cryptocurrency. People have recognized the actual potential of the Blockchain network as a result of the advancements in this industry. In spite of smart contracts' many advantages, they can still be vulnerable. One example is that attacks can happen while the contracts are still in development. Solidity smart contracts are vulnerable to the Reentrancy attack, this vulnerability occurs when unanticipated behaviors can be exploited to the project's detriment. If a function calls an untrusted contract externally, it is considered a reentrancy attack. In an attempt to drain funds, the untrustworthy contract recursively calls the original function. The attacker can continuously drain the contract's funds if the contract fails to update its state before sending money. The reentrancy exploit could take place whenever a smart contract calls another smart contract externally. When an EVM is called by a smart contract, the execution responsibility of the EVM is transferred from the smart contract that is calling to the one calling (as shown in Figure 1 indicated by the green arrow). There is therefore a danger except if the smart contract executing the call is familiar with the code of a smart contract being called. Depending on the call to the smart contract, the external code could be used in any way it wishes. Smart contracts that have been called can include actions such as calling back to the original smart contracts (as shown in Figure 1 indicated by the red arrow).

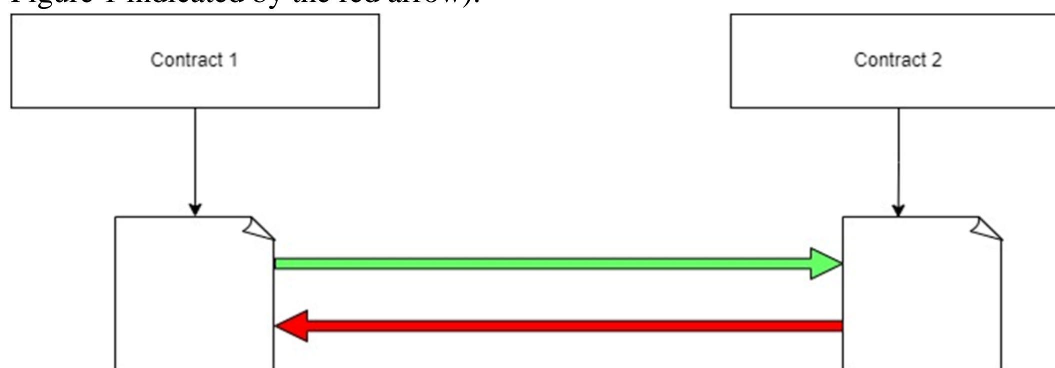

Figure 1: Transaction of contracts

During DAO attack, 60 million US dollars were lost as a result of Reentrancy, which makes it a very harmful vulnerability.

Structure of the paper. The rest of this paper is organised as follows:

1) Section 2 talks about the Literature Review

2) Section 3 talks about the Research Methodology
3) Section 4 talks about the Design
4) Section 5 talks about the Implementation
5) Section 6 talks about the Evaluation
6) Section 7 talks about Conclusion and Future Work
7) Section 8 includes link to the Video demonstration
8) Section 9 includes References

2 Related Work

Since we know that Reentrancy is a very risky vulnerability in the Blockchain's Smart contracts as the information might be used by a malignant smart contract to build a "fabricated fallback function" to perform malicious operations in the initial smart contract, there should be a solution for it to avoid this vulnerability (Tantikul, 2020). Many reentrancy detection coding techniques and tools have been innovated to know if there is a reentrancy vulnerability in a smart contract programmed code or not by researchers amongst which one of the first detection static tools was Oyente (Luu, 2016) which was successful at some levels to find reentrancy, symbolic run on EVM bytecode is used and also it was the basis for some other tools such as Oyente, but it had too many false positives and false negatives. After this, other researchers also presented other static tools for detecting and revealing reentrancy in smart contracts in the Blockchain which were Slither, Securify, and Mythril out of which the work of group researched which worked on Slither was the best of all the static detection tools as it had less FPs and FNs and more accurate than others.
One of the most notable and accurate detection tool ReDefender (Pan, 2021) was proposed and innovated by a group of researchers which made a significant improvement when compared to other static tools such as Oyente as it had the minimum number of false positives and false negatives making it more accurate and efficient.

2.1 Detecting Reentrancy

For detection of reentrancy many different tools use different rules, such as Slither (Feist, 2019) presented by TrailOfBits which is a tool that uses a rule which should be met by the condition given in Figure below (Figure 2) for reentrancy to be possible.

$$\left(r(var_g) \lor w(var_g)\right) > externCall > w(var_g) \Rightarrow \text{reentrancy} \quad (1)$$

Figure 2: Slither's rule

In the above-given figure (Figure 2), Read and write operations are indicated by r() and w() respectively, varg(g) represents a public variable. In the control flow of a program, ">" indicates the order of execution. Payment functions that are externally called except send() and transfer() are described in the extrnCall property. According to this stipulation, in a case

where there are sequence processes to the very same accessible parameter, reentrancy may occur when a call is made to an external payment function.
This principle also gives False negatives and false positives as a result which makes it less effective. Now we will look at Securify tool which has better effectiveness than Slither.
A tool presented by group of researchers named as Securify (Tsankov, 2018), the way this tools works is that it first takes in the smart contract's source code as its initial input and to proceed with the analysis, the files are then compiled into EVM bytecode. Then the first set of patterns captures the conditions that a contract must satisfy in order to violate security properties and the second set illustrates the conditions needed for a contract to violate security properties. After that stackless representation in the static-single assignment of the EVM bytecode given as initial input then it examines the contract after decompiling it to find semantic rules underlying all of its actions, including information and control-flow dependencies, securify then refers to a set of adherence patterns and cybersecurity violations once semantic data have been acquired finally when it detects a breach pattern. Finally, it gives the output of the command that causes a breach pattern to match. It has some limitations such as overflows are not identified by it, every function in the smart contract is reachable so it assumes, and it also gives false positives and false negatives same as the case is with the Slither tool, but it is more precise than Slither.
Oyente is one of the first tools for detecting reentrancy presented by Melonport AG which has very similar static procedural rules to that of Slither and Securify. So, it has same limitations as Slither and Security making it less effective. Here False negative is a case where the tool misses a reentrancy activity because smart contract static tools typically miss analysis of some important but suspicious program paths, such as cross-function or cross-contract call chains which is the case for all above-mentioned tools. (Luu, 2016). Also, by extending Oyente to check for integer bugs in smart contracts, Oyente researchers have developed Osiris and it has a similar accuracy rate as Oyente (Torres, 2018).
SolSaviour is a mechanism for mending and retrieving existing faulty smart contracts by reallocating repaired contracts and transitioning the initial condition of previous smart contracts to modified ones (Li, 2021). Some other researchers also suggested a runtime hook mechanism for syncing and evaluating the Ethereum contract data's existing operations (Lin, 2020).
A static analysis tool called SmartCheck, innovated by SmartDec, uses patterns to detect weaknesses and suspicious coding methods and it analyzes Solidity source code from a lexical and syntactical perspective, but it is very less precise when compared to Slither tool (Tikhomirov, 2018).
Mythril is a tool innovated by ConsenSys that among the other security issues detects reentrancy, using taint path evalutaion (Mueller, 2018). The Mythril tool cannot identify issues in an application's business logic, since it is designed to find common vulnerabilities. A Mythril executor is not well-suited to exploring all possible states of a program, because often it is not able to do so. But its detection accuracy is better than above given three tools as found by researchers using analysis tools (Mueller, 2021).
A group of researchers developed a framework depending on the ABI requirements of the smart contracts undergoing evaluation, this structure constructs a harmful contract and

evaluates the smart contract communication to accurately notify reentrancy vulnerability. But this framework is only theoretical and not made in real time (Fatima Samreen, 2020).
A group of researchers built a machine learning model that analyzes transaction data and classifies them as benign or harmful based on features extracted from the data. Its accuracy is low, uses machine learning (Eshghie, 2021). Another group of researchers also use deep learning to make a framework for detecting reentracy vulnerability, but runtime execution is not analysed by this framework (P. Qian, 2020).
Some researchers employed a Datalog-based framework to discover reentrancy vulnerability in the real time of Ethereum and contrasted the findings to those of other tools, which gave much less false positives (Tang, 2021). While some other researchers (Cecchetti, 2021) proposed a process to maintain security that allows smart contracts to secure their important elements while keeping the descriptive potential of safe kinds of reentrancy.
With TrailOfBits' (which also developed Slither) Manticore tool, symbolic execution is used to identify possible reentrancy vulnerabilities within EVM bytecode and self-destruct areas, but it is less accurate than Slither, Securify, and Mythril (Mossberg, 2019).
One of the recently developed tools by researchers for detecting reentracy is ReDefender tool and ReGuard tool which is based on fuzz testing it is different from the above tools in a way that it is dynamic and not static and In ReDefender, the source code from uploaded contracts is pre-processed for fuzz testing, a fuzzing engine will provide the input for the fuzzing process, in the construction of an agent contract, all contracts are targeted for interaction and assault, fuzzing inputs collect runtime information while they are executed, a malicious reentrancy is detected by analyzing the execution log (Pan, 2021). The ReGuard tool works on a similar workflow, as it was developed using fuzzing. To perform fuzz tests on smart contracts, ReGuard generates diverse but random transactions iteratively. It then automatically discovers vulnerability reentrancy, based on the execution logs. and it also uses fuzz testing as its base (Liu, 2018). The limitations of this tool are,

- The fuzzing is not narrowed down which creates unnecessary redundancy,
- The execution timeouts are not dynamically determined which makes it a little less effective,

but it still provides more efficiency than the other 3 static tools, as it can do the analysis of smart contracts dynamically. Also, these tools are more effective than SmartInspect tool (Bragagnolo, 2018) which provides better visualization.
The Clairvoyance Smart Contract static analysis tool (Xue, 2020) detects reentrancy vulnerabilities by identifying infeasible paths across functions and contracts. The workflow of Clairvoyance is that “msg.sender” is first checked to see whether it is part of an authorized list of addresses or contracts, or whether it has permission to send. Next, it is checked for modification or initialization before the external call of the tainted address or object, afterwards, implement the above given two principles to the function's auto-defined modifiers. Verify that the execution lock is present, and finally, verify that the pattern “checks-effects-interactions” exists. This tool has more True positives than other static tools, but it still has many false positives too (Ye, 2020).
In this section, I have talked about the related works and their strengths and limitations and in the next section, I will be doing an analysis of these tools.

2.2 Analysis of Tools

The research gap or niche in the above-related works is that all the above-listed tools still give some false positives and false negatives which I will try to improve in my novel solution, the static tools take quite a bit of time to do the detection analysis on smart contracts my proposed solution will be able to do it faster, the dynamic tools that used fuzzing techniques have not got it narrowed down I will make it possible in my proposed solution through the technique of static analysis building CFG with this I will be able to avoid redundancy in the execution of paths, and use path coverage rates to dynamically determine execution timeouts to be more accurate.

3 Research Methodology

By running the instalation code, Brownie console was installed successfully installed. Brownie will be installed in .local/bin in home directory by default. Due to the fact that some of the things that Brownie maintains shouldn't be up in the GitHub repository, I usually make a project in the subdirectory I added to the gitignore file, then symlink to the contracts and tests that will be needed/used. Upon running the essential commands for compiling and deploying a smart contract, Brownie console downloads Solidity compiler and runs it. If contracts are unchanged, Brownie console does not recompile them by default, but it can be forced it using --all flag. As soon as Brownie console is launched on a machine, it will start looking for an Ethereum client running on that machine. Now the smart contract is deployed when we run the command "Counter.deploy()". On running a specific command it will be shown that the newly created object has methods read and increment. These are used in our smart contract to increment the value of counter by 1. Then a command is run to send and recieve Ether between two user accounts. After that, the account is saved. Tests are written in Python and then our smart contract is tested using the Brownie test command to get to the desired outcome (christianb93, 2021).
I collected dataset from different pages of Github which are listed below,

- https://github.com/smartbugs/SolidiFI-benchmark/tree/master/buggy_contracts/Re-entrancy
- https://solidity-by-example.org/hacks/re-entrancy/

For doing the same thing with web3 I first import the web3 library, then I establish a communication with the Ganache server when version string is called. Then, I checked balance of the account that I created with Brownie, also the address and key of the user account is copied by me for later use. The ether is then transferred to and called back between the user account that I created and alice. Finally, we communicated with the smart contract upon creating the smart contract application binary interface (christianb93, 2021).
For the verification of Slither tool I first installed the dependencies that it required which are Python 3 and solidity compiler, the solidity compiler mentioned in the source link is wrong which is solcjs but I had to install the solc-select for the tool to run successfully. Then, I installed slither analyser and analysed a smart contract from a dataset mentioned below to check for the efficiency and working of Slither, which then generated the output in the text form in the terminal (trailofbits, 2022).

For the verification of the Mythril tool I first installed the tool using the pip3 command of python 3. Upon successful completion of installation of the tool I ran the analysis on a solidity smart contract file and it ran successfully, which then generated the output in the text form in the terminal (ConsenSys, 2020).
Upon comparing both the tools we observe that Mythril takes more time than Slither but accuracy of Mythril is more than that of Slither tool.

4 Design Specification

Brownie (christianb93, 2021):
The Brownie framework targets the Ethereum Virtual Machine and is based on Python. Its functionalities include,

- Using Pytest, you can check the range via traces. Utilize stack-trace analysis to analyse test ranges when writing unit tests in Python.
- Hypothesis-based testing of properties and states.
- One can set custom error strings and create tracebacks using Python-style tools.
- Quickly interact with your project with the built-in console.
- EthPM packages are supported.
- Create automated workflows for deploying smart contracts onto blockchains and initializing or integrating them.
- For quick testing in a local environment or interaction with your smart contracts on the mainnet, write scripts or use the console.
- To help you pinpoint the issue quickly, you will receive detailed information when a transaction reverts.

Brownie makes it very easy to build, deploy and interact with smart contracts and testing for vulnerabilities in the smart contracts, and Solidity and Vyper are fully supported.

Web3 (christianb93, 2021):
The web3.py library allows you to interact with Ethereum using Python such as interacting with a smart contract that was created using Brownie upon defining the smart contract App Binary Interface. The architecture of web3.py consists of 4 phases is given in the Figure (Figure 3) below,

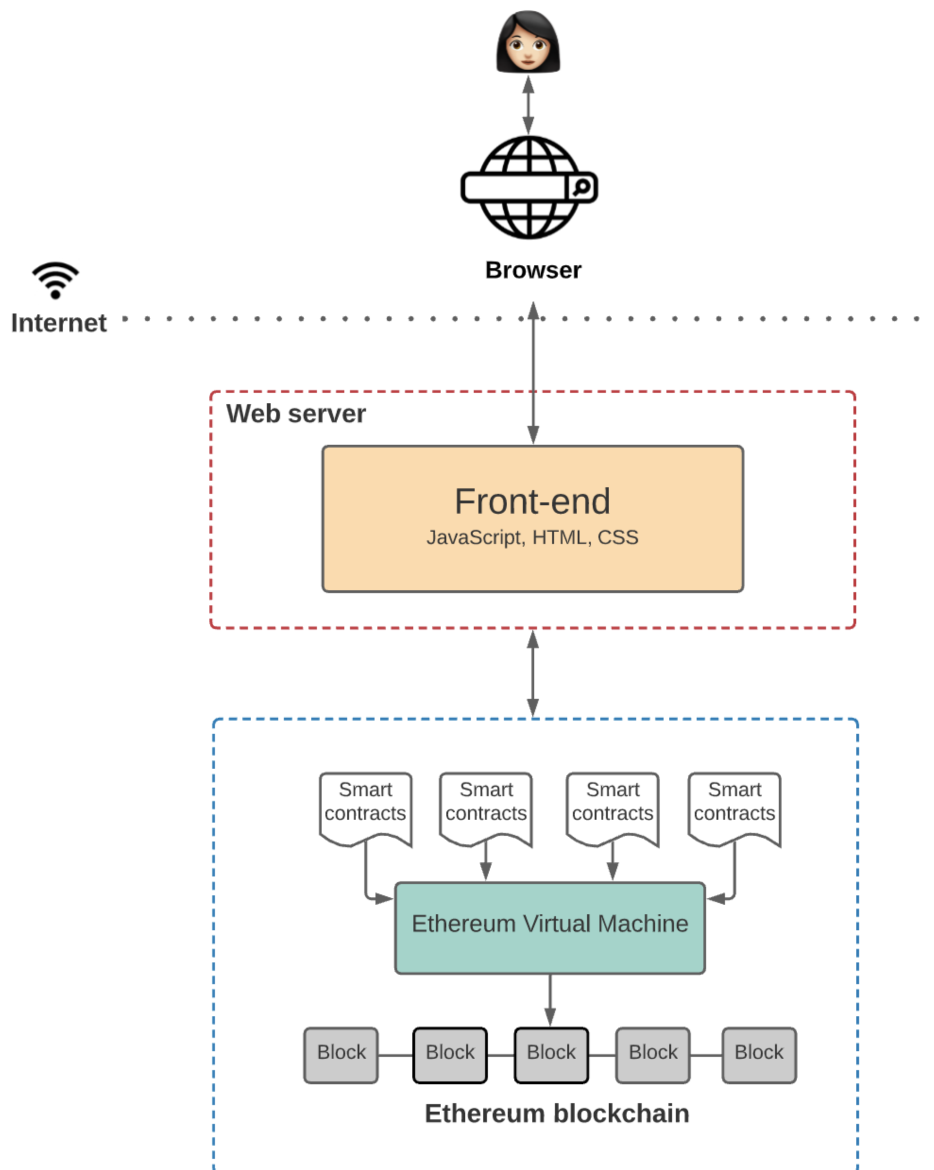

Figure 3: Architecture of Web3

1. Blockchain: Everyone can access and write to a blockchain, which is designed as a state machine. Thus, this system belongs to the entire network collectively rather than to a single entity. Furthermore, only additional information may be put to the Ethereum blockchain; current data cannot be modified there.
2. Smart Contracts: Anyone can examine the application logic contained in a smart contract because the Ethereum blockchain stores the smart contract script.
3. Ethereum Virtual Machine (EVM): Unlike high-level languages like Solidity and, the EVM does not understand high-level languages like them. In order for the EVM to execute the high-level language, you must compile it down into bytecode.
4. Frontend: As a final step, let's take a look at the front end. Additionally, it communicates with the contract logic defined in the app logic, as we already mentioned. There is a little more complexity to the interaction between both the front end and contracts than seems in the figure above.

Slither (Feist, 2019):
Due to its adaptability, Slither is the perfect foundation for studying smart contract code and allowing a wide range of applications. Current uses of the framework include,

- Automated vulnerability identification makes it possible to find a wide range of smart contract issues without human input.
- Automated code optimisation detection makes it possible to find code improvements that the compiler overlooks.
- Code understanding makes it possible to assist in the analysis of the codebase, printers summarise and display the data from smart contracts.
- Assisted code review makes it possible for clients to communicate with Slither using API.

The architecture of Slither is depicted in the image (Figure 4) below,

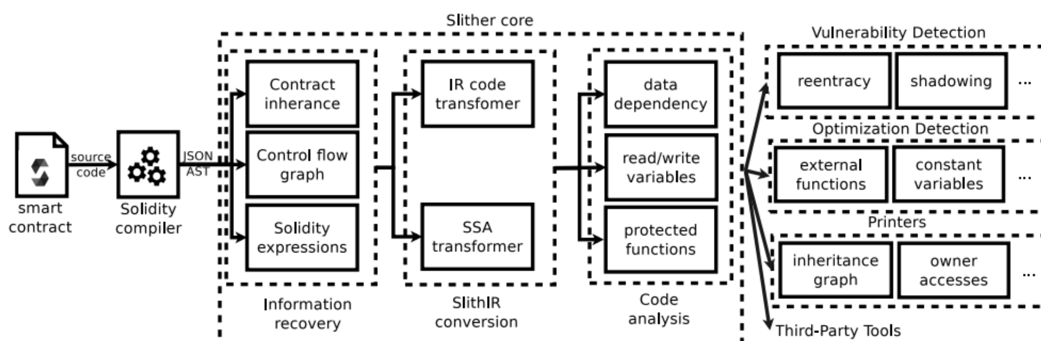

Figure 4: Architecture of Slither

A multistage static analysis procedure is used by Slither to analyse contracts. First, a Solidity compiler is used to generate a Solidity Abstract Syntax Tree (AST) from the smart contract source code. This AST is used as an initial input by Slither. After that, Slither first retrieves essential smart contract information such as inheritance graphs, expression lists, and control flow graphs (CFGs). The entire smart contract code is then converted into SlithIR, Slither's internal representation language. For the computation of various code analyses, SlithIR makes use of static single assessment (SSA). After that is complete, Slither finally starts the actual code analysis. It calculates a group of pre-determined analyses that give essential information to other modules. All of the above-mentioned stages are summarized in the figure above.

Mythril (ConsenSys, 2020):
The Mythril tool analyses EVM bytecode for security issues. This tool detects security issues in contracts developed for Ethereum, Quorum, Hedera, Vechain, Tron, Roostock, and other EVM-compatible blockchains. To detect various forms of security issues, it uses symbolic execution (SMT solving) and taint analysis. To prune the search space and to look for values that allow exploiting smart contracts, ConsenSys uses concolic analysis, taint analysis, and control flow verification of the EVM bytecode.

5 Implementation

Slither:
Slither is implemented on the smart contracts of a dataset which are taken from the following link,
https://github.com/smartbugs/SolidiFI-benchmark/tree/master/buggy_contracts/Re-entrancy
The dataset contains 50 files that are written in solidity language and have a re-entrancy vulnerability. This vulnerability has been induced in all the smart contracts present in the dataset. In the screenshots below I have shown the analysis of slither analyser on a smart contract taken from this dataset. I did have to change the version of the solidity compiler as for slither to run successfully the version of the solidity compiler and the version in which the smart contract is written should be the same. In the code of the smart contracts that I tested I also tried modifying the code such as changing the functions of the code but most of the time the code compiled with errors.
Continuous (Feist, 2019) integration is supported as well as developer toolboxes. AST parsing of the smart contract under examination requires the latest version of the Solidity compiler, which doesn't have many dependencies.
Mythril:
Mythril is implemented on the smart contracts of a dataset which are taken from the following link,
https://solidity-by-example.org/hacks/re-entrancy/
This smart contract code has been modified by me such as I increased the number of ethers being transferred as given in the solidity file and then I used the mythril to analyse the smart contact file and it successfully did the analysis with the output of re-entrancy vulnerability. The mythril tool only works for the smart contracts written in solidity version >=0.8. So the latest version of the solidity compiler needs to be downloaded for the mythril tool to run successfully.
Both the tools slither and mythril gives the expected output as both tools identify the re-entrancy vulnerability in the dataset/smart contracts file written in solidity.

6 Evaluation

I used Brownie and Web3 (christianb93, 2021) (christianb93, 2021) modules that are included in the Python 3 package to do the following,

- Created a smart contract (or the smart contracts present in the ganache server can also be utilized).
- Deployed the smart contract.
- Know the details about logs which are the subjects, data, and address, the interpretation of logs i.e., events and information related to the smart contract such as the block number, the gas price, the gas limit, the gas utilized, and even a complete execution trace down to the level of individual commands.
- Append new user accounts to the key store once they are created.
- Managed accounts are viewed by getting a list of them.
- With a given user account, signed a transaction.

- A private key of user account is used to import an account (also locking and unlocking of user accounts).
- Finally ran tests for finding vulnerabilities with Brownie and in Web3 created smart contract ABI for interaction with the smart contract.

After that, I ran tests for re-entrancy detection using Slither tool on the dataset mentioned above and verified the efficiency of the tool, the results that I got upon running the tests are about the vulnerabilities present in the code in text form in the terminal, this tool tests each line of the smart contract dataset code written in solidity language.

Finally, I ran tests for re-entrancy detection using Mythril tool on the smart contract file mentioned above and verified the efficiency of the tool, the results that I got upon running the tests are about the vulnerabilities present in the code in text form in the terminal, this tool tests each line of the smart contract dataset code written in solidity language.

6.1 Brownie

The results that I got from testing using brownie is shown in the figures (Figure 5, 6, 7, 8 and 9) below,

```
Ubuntu 64-bit - VMware Workstation 16 Player (Non-commercial use only)
Player
Activities   Terminal   Aug 15 05:57
ubuntu@ubuntu-virtual-machine: ~/nft-bootcamp/tmp
ubuntu@ubuntu-virtual-machine:~$ cd nft-bootcamp
ubuntu@ubuntu-virtual-machine:~/nft-bootcamp$ cd tmp
ubuntu@ubuntu-virtual-machine:~/nft-bootcamp/tmp$ brownie compile
Brownie v1.19.0 - Python development framework for Ethereum

Project has been compiled. Build artifacts saved at /home/ubuntu/nft-bootcamp/t
mp/build/contracts
ubuntu@ubuntu-virtual-machine:~/nft-bootcamp/tmp$ brownie console
Brownie v1.19.0 - Python development framework for Ethereum

TmpProject is the active project.

Launching 'ganache-cli --port 8545 --gasLimit 12000000 --accounts 10 --hardfork
 istanbul --mnemonic brownie'...
Brownie environment is ready.
>>> counter = Counter.deploy({"from": accounts[0]});
Transaction sent: 0x612007a9a263036fd2a5cfa00dbae17f8aecdcb0c3d542fe13490dc31f4
59eae
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 0
  Counter.constructor confirmed   Block: 1   Gas used: 117127 (0.98%)
  Counter deployed at: 0x3194cBDC3dbcd3E11a07892e7bA5c3394048Cc87

>>> dir(Counter)
[abi, at, bytecode, decode_input, deploy, get_method, get_verification_info, in
fo, publish_source, remove, selectors, signatures, topics, tx]
>>> counter.read()
0
Show Applications r.increment()
Transaction sent: 0x8d9d6cc869fce1566c733e4b0eb50f953250f98c8b18376a4fac8ef2899
```

Ubuntu 64-bit - VMware Workstation 16 Player (Non-commercial use only)

Player

Activities   Terminal   Aug 15 05:58

ubuntu@ubuntu-virtual-machine: ~/nft-bootcamp/tmp

```
Transaction sent: 0x8d9d6cc869fce1566c733e4b0eb50f953250f98c8b18376a4fac8ef2899
0069e
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 1
  Counter.increment confirmed   Block: 2   Gas used: 44684 (0.37%)

>>> counter.read()
1
>>> txn.logs
[AttributeDict({'logIndex': 0, 'transactionIndex': 0, 'transactionHash': HexByt
es('0x8d9d6cc869fce1566c733e4b0eb50f953250f98c8b18376a4fac8ef28990069e'), 'bloc
kHash': HexBytes('0x8bb826067135ce1e96f84f3d810ac432aec022417dd6976d54234090050
7eef5'), 'blockNumber': 2, 'address': '0x3194cBDC3dbcd3E11a07892e7bA5c3394048Cc
87', 'data': '0x000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000001', 'topics': [
HexBytes('0x64f50d594c2a739c7088f9fc6785e1934030e17b52f1a894baec61b98633a59f'),
 HexBytes('0x00000000000000000000000066ab6d9362d4f35596279692f0251db635165871')
], 'type': 'mined'})]
>>> txn.events
{'Increment': [OrderedDict([('sender', '0x66aB6D9362d4F35596279692F0251Db635165
871'), ('oldValue', 0), ('newValue', 1)])]}
>>> txn.info()
Transaction was Mined
---------------------
Tx Hash: 0x8d9d6cc869fce1566c733e4b0eb50f953250f98c8b18376a4fac8ef28990069e
From: 0x66aB6D9362d4F35596279692F0251Db635165871
To: 0x3194cBDC3dbcd3E11a07892e7bA5c3394048Cc87
Value: 0
Function: Counter.increment
Block: 2
```

Ubuntu 64-bit - VMware Workstation 16 Player (Non-commercial use only)

Player

Activities   Terminal   Aug 15 05:59

ubuntu@ubuntu-virtual-machine: ~/nft-bootcamp/tmp

```
Gas Used: 44684 / 12000000 (0.4%)

Events In This Transaction
--------------------------
└── Counter (0x3194cBDC3dbcd3E11a07892e7bA5c3394048Cc87)
    └── Increment
        ├── sender: 0x66aB6D9362d4F35596279692F0251Db635165871
        ├── oldValue: 0
        └── newValue: 1

>>> web3.eth.get_accounts()
  File "<console>", line 1, in <module>
AttributeError: 'Eth' object has no attribute 'get_accounts'
>>> web3.eth._get_accounts()
["0x66aB6D9362d4F35596279692F0251Db635165871", "0x33A4622B82D4c04a53e170c638B94
4ce27cffce3", "0x0063046686E46Dc6F15918b61AE2B121458534a5", "0x21b42413bA931038
f35e7A5224FaDb065d297Ba3", "0x46C0a5326E643E4f71D3149d50B48216e174Ae84", "0x807
c47A89F720fe4Ee9b8343c286Fc886f43191b", "0x844ec86426F076647A5362706a04570A5965
473B", "0x23BB2Bb6c340D4C91cAa478EdF6593fC5c4a6d4B", "0xA868bC7c1AF08B8831795FA
C946025557369F69C", "0x1CEE82EEd89Bd5Be5bf2507a92a755dcF1D8e8dc"]
>>> accounts
[<Account '0x66aB6D9362d4F35596279692F0251Db635165871'>, <Account '0x33A4622B82
D4c04a53e170c638B944ce27cffce3'>, <Account '0x0063046686E46Dc6F15918b61AE2B1214
58534a5'>, <Account '0x21b42413bA931038f35e7A5224FaDb065d297Ba3'>, <Account '0x
46C0a5326E643E4f71D3149d50B48216e174Ae84'>, <Account '0x807c47A89F720fe4Ee9b834
3c286Fc886f43191b'>, <Account '0x844ec86426F076647A5362706a04570A5965473B'>, <A
ccount '0x23BB2Bb6c340D4C91cAa478EdF6593fC5c4a6d4B'>, <Account '0xA868bC7c1AF08
B8831795FAC946025557369F69C'>, <Account '0x1CEE82EEd89Bd5Be5bf2507a92a755dcF1D8
e8dc'>]
```

Ubuntu 64-bit - VMware Workstation 16 Player (Non-commercial use only)

Activities Terminal Aug 15 05:59

ubuntu@ubuntu-virtual-machine: ~/nft-bootcamp/tmp

```
>>> me = accounts.add()
mnemonic: 'globe champion welcome cry defy total moment great oblige screen coc
onut dirt'
>>> alice = accounts.add()
mnemonic: 'resemble program flavor broken crazy brush spice bicycle puzzle loop
 adult since'
>>> accounts[0].transfer(to=me.address, amount=1000);
Transaction sent: 0xbe900403c3af50ef2447a8e37b15234c34124e70dfa46eaa9487a156d25
a1d63
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 2
  Transaction confirmed   Block: 3   Gas used: 21000 (0.18%)

<Transaction '0xbe900403c3af50ef2447a8e37b15234c34124e70dfa46eaa9487a156d25a1d6
3'>
>>> txn = {
  "from": me.address,
  "to": alice.address,
  "value": 10,
  "gas": 21000,
  "gasPrice": 0,
  "nonce": me.nonce
}
>>> txn_signed = web3.eth.account.signTransaction(txn, me.private_key)
>>> web3.eth.send_raw_transaction(txn_signed.rawTransaction)
HexBytes('0x828fc4f4bb4f82c3bc81cf4c46e954e4858a2912c80bccef240019c285f9b337')
>>> alice.balance()
10
>>> me.save("myAccount")
  File "<console>", line 1, in <module>
```

Ubuntu 64-bit - VMware Workstation 16 Player (Non-commercial use only)

Activities Terminal Aug 15 06:01

ubuntu@ubuntu-virtual-machine: ~/nft-bootcamp/tmp

```
ubuntu@ubuntu-virtual-machine:~/nft-bootcamp/tmp$ brownie compile
Brownie v1.19.0 - Python development framework for Ethereum

Project has been compiled. Build artifacts saved at /home/ubuntu/nft-bootcamp/t
mp/build/contracts
ubuntu@ubuntu-virtual-machine:~/nft-bootcamp/tmp$ brownie test tests/test_Count
er.py
Brownie v1.19.0 - Python development framework for Ethereum

============================= test session starts =============================
platform linux -- Python 3.10.4, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /home/ubuntu/nft-bootcamp/tmp
plugins: eth-brownie-1.19.0, web3-5.29.1, xdist-1.34.0, forked-1.4.0, hypothesi
s-6.27.3
collected 2 items

Launching 'ganache-cli --port 8545 --gasLimit 12000000 --accounts 10 --hardfork
 istanbul --mnemonic brownie'...

tests/test_Counter.py ..                                                [100%]

============================= 2 passed in 10.07s ==============================
Terminating local RPC client...
ubuntu@ubuntu-virtual-machine:~/nft-bootcamp/tmp$
```

We can see from the above image that it gives all the information about the smart contract such as the coverage report, methods invoked, and the gas used. The report of vulnerabilities can be viewed through Brownie GUI.

6.2 Web3

The results that I got from using Web3 on the smart contract account created with Brownie is shown in the image (Figure 10, 11, 12, 13, and 14) below,

```
Ubuntu 64-bit - VMware Workstation 16 Player (Non-commercial use only)
Player
Activities    Terminal    Aug 15 06:09
ubuntu@ubuntu-virtual-machine: ~/nft-bootcamp/tmp
ubuntu@ubuntu-virtual-machine:~/nft-bootcamp/tmp$ python3
Python 3.10.4 (main, Jun 29 2022, 12:14:53) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import web3
>>> w3 = web3.Web3(web3.HTTPProvider('http://127.0.0.1:8545'))
>>> w3.clientVersion
'EthereumJS TestRPC/v2.13.1/ethereum-js'
>>> me = w3.eth._get_accounts()[0]
>>> w3.eth.get_balance(me)
100000000000000000000
>>> alice = "0x7D72Df7F4C7072235523A8FEdcE9FF6D236595F3"
>>> value = w3.toWei(10, "ether")
>>> txn = {
...    "from": me,
...    "to": alice,
...    "value": value,
...    "gas": 21000,
...    "gasPrice": 0
... }
>>> txn_hash = w3.eth.send_transaction(txn)
>>> w3.eth.wait_for_transaction_receipt(txn_hash)
AttributeDict({'transactionHash': HexBytes('0x4cd093620b653b61801751922fa76137d
4398b8bdc73694e076bc534d9b13104'), 'transactionIndex': 0, 'blockHash': HexBytes
('0xbbd4e7d507fbb13fca71bef959997a022dc13bc998e54bbcdfceda2e51bbf6f5'), 'blockN
umber': 2, 'from': '0x66aB6D9362d4F35596279692F0251Db635165871', 'to': '0x7D72D
f7F4C7072235523A8FEdcE9FF6D236595F3', 'gasUsed': 21000, 'cumulativeGasUsed': 21
000, 'contractAddress': None, 'logs': [], 'status': 1, 'logsBloom': HexBytes('0
x000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Ubuntu 64-bit - VMware Workstation 16 Player (Non-commercial use only)

Player

Activities Terminal Aug 15 06:10

ubuntu@ubuntu-virtual-machine: ~/nft-bootcamp/tmp

```
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000')})
>>> w3.eth.get_balance(me)
90000000000000000000
>>> w3.eth.get_balance(alice)
10000000000000000000
>>> nonce = w3.eth.get_transaction_count(alice)
>>> refund = {
...   "from": alice,
...   "to": me,
...   "value": value,
...   "gas": 21000,
...   "gasPrice": 0,
...   "nonce": nonce
... }
>>> key = "0x5777ee3ba27ad814f984a36542d9862f652084e7ce366e2738ceaa0fb0fff350"
>>> signed_txn = w3.eth.account.sign_transaction(refund, key)
>>> txn_hash = w3.eth.send_raw_transaction(signed_txn.rawTransaction)
>>> w3.eth.wait_for_transaction_receipt(txn_hash)
AttributeDict({'transactionHash': HexBytes('0x142dd036990c22583c6e012db6619a2b5
b20bfadd5db6300c282f975f6fce202'), 'transactionIndex': 0, 'blockHash': HexBytes
('0xea2d95feb49f0d87e0fb3219b4533709d5c118bc5df7823bcc15a9d7d99e1c99'), 'blockN
umber': 3, 'from': '0x7D72Df7F4C7072235523A8FEdcE9FF6D236595F3', 'to': '0x66aB6
D9362d4F35596279692F0251Db635165871', 'gasUsed': 21000, 'cumulativeGasUsed': 21
000, 'contractAddress': None, 'logs': [], 'status': 1, 'logsBloom': HexBytes('0
x0000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Ubuntu 64-bit - VMware Workstation 16 Player (Non-commercial use only)

Player

Activities Terminal Aug 15 06:10

ubuntu@ubuntu-virtual-machine: ~/nft-bootcamp/tmp

```
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000')})
>>> w3.eth.get_balance(me)
100000000000000000000
>>> w3.eth.get_balance(alice)
0
>>> abi = [
...     {
...         'anonymous': False,
...         'inputs': [
...             {
...                 'indexed': True,
...                 'internalType': "address",
...                 'name': "sender",
...                 'type': "address"
...             },
...             {
...                 'indexed': False,
...                 'internalType': "uint256",
...                 'name': "oldValue",
...                 'type': "uint256"
...             },
...             {
...                 'indexed': False,
...                 'internalType': "uint256",
```

Ubuntu 64-bit - VMware Workstation 16 Player (Non-commercial use only)

Player

Activities Terminal Aug 15 06:11

ubuntu@ubuntu-virtual-machine: ~/nft-bootcamp/tmp

```
...                 'type': "uint256"
...             }
...         ],
...         'name': "Increment",
...         'type': "event"
...     },
...     {
...         'inputs': [],
...         'name': "increment",
...         'outputs': [],
...         'stateMutability': "nonpayable",
...         'type': "function"
...     },
...     {
...         'inputs': [],
...         'name': "read",
...         'outputs': [
...             {
...                 'internalType': "uint256",
...                 'name': "",
...                 'type': "uint256"
...             }
...         ],
...         'stateMutability': "view",
...         'type': "function"
...     }
... ]
>>> address = "0x3194cBDC3dbcd3E11a07892e7bA5c3394048Cc87"
>>> counter = w3.eth.contract(address=address, abi=abi)
```

Ubuntu 64-bit - VMware Workstation 16 Player (Non-commercial use only)

Player

Activities Terminal Aug 15 06:11

ubuntu@ubuntu-virtual-machine: ~/nft-bootcamp/tmp

```
>>> counter = w3.eth.contract(address=address, abi=abi)
>>> counter.functions.read().call()
0
>>> txn_hash = counter.functions.increment().transact({"from": me})
>>> w3.eth.wait_for_transaction_receipt(txn_hash)
AttributeDict({'transactionHash': HexBytes('0xea1deb22b1cb377c383897c250a436e7c
f7c26d82ddf98c3c1d6108880ef4eeb'), 'transactionIndex': 0, 'blockHash': HexBytes
('0x0d222c42c739a1c7ac3ca453ba117a2610da235b15f9b6371088f9a17dc3f604'), 'blockN
umber': 4, 'from': '0x66aB6D9362d4F35596279692F0251Db635165871', 'to': '0x3194c
BDC3dbcd3E11a07892e7bA5c3394048Cc87', 'gasUsed': 44684, 'cumulativeGasUsed': 44
684, 'contractAddress': None, 'logs': [AttributeDict({'logIndex': 0, 'transacti
onIndex': 0, 'transactionHash': HexBytes('0xea1deb22b1cb377c383897c250a436e7cf7
c26d82ddf98c3c1d6108880ef4eeb'), 'blockHash': HexBytes('0x0d222c42c739a1c7ac3ca
453ba117a2610da235b15f9b6371088f9a17dc3f604'), 'blockNumber': 4, 'address': '0x
3194cBDC3dbcd3E11a07892e7bA5c3394048Cc87', 'data': '0x0000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000001', 'topics': [HexBytes('0x64f50d594c2a739c7088f9fc6785e
1934030e17b52f1a894baec61b98633a59f'), HexBytes('0x00000000000000000000000066ab
6d9362d4f35596279692f0251db635165871')], 'type': 'mined'})], 'status': 1, 'logs
Bloom': HexBytes('0x0000000000000000000000000000400000000000000000080000000000000
0000000000000000000000000000000000000000000000000001000000000000000000000000000
0000000000000000000000000000000000000000000000000002000000000000000000000000000
0000000000000000000000200000000000000000040000000000000000000000000000000000000
0000000000000000000000000000000100000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000080000000000
0000020000000000000000000000000000000000000000000000000000')})
>>> counter.functions.read().call()
1
>>> 
```

We can see from the above image that the methods specified in the contracts ABI can be used with Web3 such as to read and increment the value of Counter an example of interaction with the smart contract.

6.3 Slither

The results that I got from testing using Slither analyser is shown in the figure (Figure 15, 16, 17, 18, and 19) below,

Kali-Linux-2021.3-vmware-amd64 - VMware Workstation 16 Player (Non-commercial use only)

Player

kali@kali: ~ 09:10 PM

File Actions Edit View Help

```
Variable PHO.lastPlayer_re_ent30 (buggy_4.sol#105) is not in mixedCase
Variable PHO.jackpot_re_ent30 (buggy_4.sol#106) is not in mixedCase
Variable PHO.balances_re_ent8 (buggy_4.sol#114) is not in mixedCase
Variable PHO.redeemableEther_re_ent39 (buggy_4.sol#121) is not in mixedCase
Variable PHO.balances_re_ent36 (buggy_4.sol#130) is not in mixedCase
Variable PHO.counter_re_ent35 (buggy_4.sol#163) is not in mixedCase
Variable PHO.userBalance_re_ent40 (buggy_4.sol#175) is not in mixedCase
Variable PHO.userBalance_re_ent33 (buggy_4.sol#189) is not in mixedCase
Variable PHO.not_called_re_ent27 (buggy_4.sol#223) is not in mixedCase
Variable PHO.balances_re_ent31 (buggy_4.sol#241) is not in mixedCase
Variable PHO.not_called_re_ent13 (buggy_4.sol#256) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions

Reentrancy in PHO.bug_re_ent20() (buggy_4.sol#41-47):
        External calls:
        - ! (msg.sender.send(1000000000000000000)) (buggy_4.sol#43)
        State variables written after the call(s):
        - not_called_re_ent20 = false (buggy_4.sol#46)
Reentrancy in PHO.bug_re_ent27() (buggy_4.sol#224-230):
        External calls:
        - ! (msg.sender.send(1000000000000000000)) (buggy_4.sol#226)
        State variables written after the call(s):
        - not_called_re_ent27 = false (buggy_4.sol#229)
Reentrancy in PHO.buyTicket_re_ent23() (buggy_4.sol#88-93):
        External calls:
        - ! (lastPlayer_re_ent23.send(jackpot_re_ent23)) (buggy_4.sol#89)
        State variables written after the call(s):
        - jackpot_re_ent23 = address(this).balance (buggy_4.sol#92)
        - lastPlayer_re_ent23 = msg.sender (buggy_4.sol#91)
Reentrancy in PHO.buyTicket_re_ent30() (buggy_4.sol#107-112):
        External calls:
        - ! (lastPlayer_re_ent30.send(jackpot_re_ent30)) (buggy_4.sol#108)
        State variables written after the call(s):
        - jackpot_re_ent30 = address(this).balance (buggy_4.sol#111)
        - lastPlayer_re_ent30 = msg.sender (buggy_4.sol#110)
Reentrancy in PHO.callme_re_ent14() (buggy_4.sol#96-102):
        External calls:
        - ! (msg.sender.send(1000000000000000000)) (buggy_4.sol#98)
        State variables written after the call(s):
        - counter_re_ent14 += 1 (buggy_4.sol#101)
Reentrancy in PHO.callme_re_ent35() (buggy_4.sol#164-170):
        External calls:
        - ! (msg.sender.send(1000000000000000000)) (buggy_4.sol#166)
        State variables written after the call(s):
        - counter_re_ent35 += 1 (buggy_4.sol#169)
Reentrancy in PHO.callme_re_ent7() (buggy_4.sol#77-83):
        External calls:
        - ! (msg.sender.send(1000000000000000000)) (buggy_4.sol#79)
        State variables written after the call(s):
        - counter_re_ent7 += 1 (buggy_4.sol#82)
Reentrancy in PHO.claimReward_re_ent32() (buggy_4.sol#50-56):
        External calls:
```

Type here to search 27°C ENG 06:40 15-08-2022

Kali-Linux-2021.3-vmware-amd64 - VMware Workstation 16 Player (Non-commercial use only)

Player

kali@kali: ~ 09:11 PM

File Actions Edit View Help

```
        External calls:
        - msg.sender.transfer(transferValue_re_ent32) (buggy_4.sol#54)
        State variables written after the call(s):
        - redeemableEther_re_ent32[msg.sender] = 0 (buggy_4.sol#55)
Reentrancy in PHO.claimReward_re_ent39() (buggy_4.sol#122-128):
        External calls:
        - msg.sender.transfer(transferValue_re_ent39) (buggy_4.sol#126)
        State variables written after the call(s):
        - redeemableEther_re_ent39[msg.sender] = 0 (buggy_4.sol#127)
Reentrancy in PHO.claimReward_re_ent4() (buggy_4.sol#68-74):
        External calls:
        - msg.sender.transfer(transferValue_re_ent4) (buggy_4.sol#72)
        State variables written after the call(s):
        - redeemableEther_re_ent4[msg.sender] = 0 (buggy_4.sol#73)
Reentrancy in PHO.withdrawFunds_re_ent31(uint256) (buggy_4.sol#242-247):
        External calls:
        - require(bool)(msg.sender.send(_weiToWithdraw)) (buggy_4.sol#245)
        State variables written after the call(s):
        - balances_re_ent31[msg.sender] -= _weiToWithdraw (buggy_4.sol#246)
Reentrancy in PHO.withdrawFunds_re_ent38(uint256) (buggy_4.sol#60-65):
        External calls:
        - require(bool)(msg.sender.send(_weiToWithdraw)) (buggy_4.sol#63)
        State variables written after the call(s):
        - balances_re_ent38[msg.sender] -= _weiToWithdraw (buggy_4.sol#64)
Reentrancy in PHO.withdraw_balances_re_ent36() (buggy_4.sol#131-134):
        External calls:
        - msg.sender.send(balances_re_ent36[msg.sender]) (buggy_4.sol#132)
        State variables written after the call(s):
        - balances_re_ent36[msg.sender] = 0 (buggy_4.sol#133)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-4

Variable PHO.balances_re_ent31 (buggy_4.sol#241) is too similar to PHO.balances_re_ent36 (buggy_4.sol#130)
Variable PHO.balances_re_ent31 (buggy_4.sol#241) is too similar to PHO.balances_re_ent38 (buggy_4.sol#59)
Variable PHO.balances_re_ent36 (buggy_4.sol#130) is too similar to PHO.balances_re_ent38 (buggy_4.sol#59)
Variable PHO.jackpot_re_ent23 (buggy_4.sol#87) is too similar to PHO.jackpot_re_ent30 (buggy_4.sol#106)
Variable PHO.lastPlayer_re_ent23 (buggy_4.sol#86) is too similar to PHO.lastPlayer_re_ent30 (buggy_4.sol#105)
Variable PHO.not_called_re_ent20 (buggy_4.sol#40) is too similar to PHO.not_called_re_ent27 (buggy_4.sol#223)
Variable PHO.redeemableEther_re_ent32 (buggy_4.sol#49) is too similar to PHO.redeemableEther_re_ent39 (buggy_4.sol#121)
Variable PHO.claimReward_re_ent32().transferValue_re_ent32 (buggy_4.sol#53) is too similar to PHO.claimReward_re_ent39().transferValue_re_ent39 (buggy_4.sol#125)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#variable-names-are-too-similar

PHO.constructor() (buggy_4.sol#142-162) uses literals with too many digits:
        - saleAmount = toWei(1000000000) (buggy_4.sol#148)
PHO.constructor() (buggy_4.sol#142-162) uses literals with too many digits:
        - evtAmount = toWei(200000000) (buggy_4.sol#149)
PHO.constructor() (buggy_4.sol#142-162) uses literals with too many digits:
        - teamAmount = toWei(800000000) (buggy_4.sol#150)
PHO.constructor() (buggy_4.sol#142-162) uses literals with too many digits:
        - _totalSupply = toWei(2000000000) (buggy_4.sol#151)
PHO.transfer(address,uint256) (buggy_4.sol#200-222) uses literals with too many digits:
        - balances[msg.sender] - value < toWei(600000000) (buggy_4.sol#209)
PHO.transfer(address,uint256) (buggy_4.sol#200-222) uses literals with too many digits:
```

Type here to search 27°C ENG 06:41 15-08-2022

We can see from the above figure that it shows all the lines of code of the smart contract chosen from above given dataset where the possibility of reentrancy vulnerability tool is detected.

6.4 Mythril

The results that I got from testing using Mythril is shown in the figure (Figure 20, 21, 22, 23, 24, 25, 26, and 27) below,

```
Arbitrary senders other than the contract creator can profitably extract Ether from the contract account. Verify the business logic carefully and make sure that appropriate security controls are in place to prevent unexpected loss of fun
ds.
--------------------
In file: 3.sol:64

etherStore.withdraw()

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

Account: [CREATOR], balance: 0x400000000003, nonce:0, storage:{}
Account: [ATTACKER], balance: 0x3f25f4b8c589c0000, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: 000…000deadbeefdeadbeefdeadbeefdeadbeefdeadbeef000…000, value: 0x0
Caller: [SOMEGUY], function: attack(), txdata: 0x9e5faafc, value: 0xde0b6b3a7640000
Caller: [ATTACKER], function: unknown, txdata: 0x01010101, value: 0x0

==== External Call To User-Supplied Address ====
SWC ID: 107
Severity: Low
Contract: EtherStore
Function name: withdraw()
PC address: 368
Estimated Gas Usage: 7278 - 62222
A call to a user-supplied address is executed.
An external message call to an address specified by the caller is executed. Note that the callee account might contain arbitrary code and could re-enter any function within this contract. Reentering the contract in an intermediate state
may lead to unexpected behaviour. Make sure that no state modifications are executed after this call and/or reentrancy guards are in place.
--------------------
In file: 3.sol:42

msg.sender.call{value: bal}("")

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

Account: [CREATOR], balance: 0x8004000000372b, nonce:0, storage:{}
```

```
Account: [CREATOR], balance: 0x8004000000372b, nonce:0, storage:{}
Account: [ATTACKER], balance: 0x10000000000000004, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: , value: 0x0
Caller: [ATTACKER], function: deposit(), txdata: 0xd0e30db0, value: 0x1
Caller: [ATTACKER], function: withdraw(), txdata: 0x3ccfd60b, value: 0x0

==== External Call To User-Supplied Address ====
SWC ID: 107
Severity: Low
Contract: Attack
Function name: attack()
PC address: 498
Estimated Gas Usage: 4888 - 74205
A call to a user-supplied address is executed.
An external message call to an address specified by the caller is executed. Note that the callee account might contain arbitrary code and could re-enter any function within this contract. Reentering the contract in an intermediate state
may lead to unexpected behaviour. Make sure that no state modifications are executed after this call and/or reentrancy guards are in place.
--------------------
In file: 3.sol:70

etherStore.deposit{value: 1 ether}()

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

Account: [CREATOR], balance: 0x411c0800000000000, nonce:0, storage:{}
Account: [ATTACKER], balance: 0x0, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: 000…000deadbeefdeadbeefdeadbeefdeadbeefdeadbeef000…000, value: 0x0
Caller: [CREATOR], function: attack(), txdata: 0x9e5faafc, value: 0xde0b6b3a7640000

==== Unprotected Ether Withdrawal ====
SWC ID: 105
Severity: High
```

Kali-Linux-2021.3-vmware-amd64 - VMware Workstation 16 Player (Non-commercial use only)

```
Contract: Attack
Function name: attack()
PC address: 498
Estimated Gas Usage: 4888 - 74205
Any sender can withdraw Ether from the contract account.
Arbitrary senders other than the contract creator can profitably extract Ether from the contract account. Verify the business logic carefully and make sure that appropriate security controls are in place to prevent unexpected loss of fun
ds.
────────────────
In file: 3.sol:70

etherStore.deposit{value: 1 ether}()

────────────────
Initial State:

Account: [CREATOR], balance: 0x809511009081a, nonce:0, storage:{}
Account: [ATTACKER], balance: 0x4003156589c0100, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: 000000000000000000000000deadbeefdeadbeefdeadbeefdeadbeefdeadbeef000000…000000, value: 0x0
Caller: [SOMEGUY], function: attack(), txdata: 0x9e5faafc, value: 0xde0b6b3a7640000
Caller: [ATTACKER], function: attack(), txdata: 0x9e5faafc, value: 0xde0b6b3a7640000

=== State access after external call ===
SWC ID: 107
Severity: Medium
Contract: Attack
Function name: attack()
PC address: 524
Estimated Gas Usage: 4888 - 74205
Read of persistent state following external call
The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.
────────────────
In file: 3.sol:71

etherStore
```

Kali-Linux-2021.3-vmware-amd64 - VMware Workstation 16 Player (Non-commercial use only)

```
────────────────
Initial State:

Account: [CREATOR], balance: 0x49ce5d6d3cce10000, nonce:0, storage:{}
Account: [ATTACKER], balance: 0x0, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: 000000000000000000000000deadbeefdeadbeefdeadbeefdeadbeefdeadbeef000000…000000, value: 0x0
Caller: [CREATOR], function: attack(), txdata: 0x9e5faafc, value: 0xde0b6b3a7640000

=== State access after external call ===
SWC ID: 107
Severity: Medium
Contract: EtherStore
Function name: withdraw()
PC address: 556
Estimated Gas Usage: 7278 - 62222
Write to persistent state following external call
The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.
────────────────
In file: 3.sol:45

balances[msg.sender] = 0

────────────────
Initial State:

Account: [CREATOR], balance: 0x400000020d1f89, nonce:0, storage:{}
Account: [ATTACKER], balance: 0x100008, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: , value: 0x0
Caller: [ATTACKER], function: deposit(), txdata: 0xd0e30db0, value: 0x1
Caller: [ATTACKER], function: withdraw(), txdata: 0x3ccfd60b, value: 0x0
```

Kali-Linux-2021.3-vmware-amd64 - VMware Workstation 16 Player (Non-commercial use only)

```
Caller: [ATTACKER], function: withdraw(), txdata: 0×3ccfd60b, value: 0×0

==== External Call To User-Supplied Address ====
SWC ID: 107
Severity: Low
Contract: Attack
Function name: attack()
PC address: 627
Estimated Gas Usage: 4888 - 74205
A call to a user-supplied address is executed.
An external message call to an address specified by the caller is executed. Note that the callee account might contain arbitrary code and could re-enter any function within this contract. Reentering the contract in an intermediate state
may lead to unexpected behaviour. Make sure that no state modifications are executed after this call and/or reentrancy guards are in place.
--------------------
In file: 3.sol:71

etherStore.withdraw()

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

Account: [CREATOR], balance: 0×411c0800000000000, nonce:0, storage:{}
Account: [ATTACKER], balance: 0×0, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: 000000000000000000000000deadbeefdeadbeefdeadbeefdeadbeefdeadbeef000000000000…
…0000000000000000, value: 0×0
Caller: [CREATOR], function: attack(), txdata: 0×9e5faafc, value: 0×de0b6b3a7640000

==== Multiple Calls in a Single Transaction ====
SWC ID: 113
Severity: Low
Contract: Attack
Function name: attack()
PC address: 627
Estimated Gas Usage: 4888 - 74205
Multiple calls are executed in the same transaction.
This call is executed following another call within the same transaction. It is possible that the call never gets executed if a prior call fails permanently. This might be caused intentionally by a malicious callee. If possible, refactor
 the code such that each transaction only executes one external call or make sure that all callees can be trusted (i.e. they're part of your own codebase).
```

Type here to search 27°C ENG 06:38 15-08-2022

Kali-Linux-2021.3-vmware-amd64 - VMware Workstation 16 Player (Non-commercial use only)

```
etherStore.withdraw()

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

Account: [CREATOR], balance: 0×10000002c0000811, nonce:0, storage:{}
Account: [ATTACKER], balance: 0×0, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: 000000000000000000000000000000000000…
…0000000000000000, value: 0×0
Caller: [CREATOR], function: attack(), txdata: 0×9e5faafc, value: 0×de0b6b3a7640000

==== State access after external call ====
SWC ID: 107
Severity: Medium
Contract: Attack
Function name: attack()
PC address: 627
Estimated Gas Usage: 4888 - 74205
Write to persistent state following external call
The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy
lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.
--------------------
In file: 3.sol:71

etherStore.withdraw()

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

Account: [CREATOR], balance: 0×49ce5d6d3cce10000, nonce:0, storage:{}
Account: [ATTACKER], balance: 0×0, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: 000000000000000000000000deadbeefdeadbeefdeadbeefdeadbeefdeadbeef000000000000…
```

Type here to search 27°C ENG 06:38 15-08-2022

We can see from the above figure that it shows all the lines of code of the smart contract file where the possibility of reentrancy vulnerability tool is detected.

6.5 Discussion

I have verified two important reentrancy vulnerability detection tools, Slither and Mythril, in searching for a new and ingenious solution. Since the tools were written in a programming language version that is outdated and throwing errors, I debugged the code of these two tools and some other tools too. I also experimented with brownie and python console on how to build a smart contract, deploy them and interact with other smart contracts to learn how the smart contracts work. I verified Slither and Mythril on different platforms, Kali Linux, Ubuntu, and Windows OS with various different datasets. Slither was found to be taking less time than Mythril in detecting reentrancy vulnerability on the dataset of smart contracts. The link to installing the solidity compiler given in the source link for Slither is wrong it is the solcjs installed through npm which will not work for the slither analyzer, for slither we need solc-select and I found that I download the wrong solidity compiler after it threw error a few times. Also, we have to give administrative privileges to the terminal/ command prompt on which the slither is installed for it to access the directory to be stored. The Slither and Mythril could have been built for even analysing all the smart contracts with pragma solidity less than 0.8 version or the latest version so the need to upgrade every code written in older versions of solidity need not be upgraded to latest version. Mythril tool should have been built to take less time in analysing a given smart contract code. Slither might act as a compiler by converting SlithIR to EVM or bytecode. SlithIR's performance could be improved with optimization. The intermediate representation can be used to implement symbolic execution or bounded model checking, making validation for vulnerability identifiers and worst-case gas price assessment easily accessible. Both tools should be modified to be implemented on a whole dataset at once.

7 Conclusion and Future Work

In this research project I have tried to provide a novel solution/ verification reentrancy detection tools. I have verified two important reentrancy vulnerability detection tools, Slither and Mythril, in searching for a new and ingenious solution. Since the tools were written in a programming language version that is outdated and throwing errors, I debugged the code of these two tools and some other tools too. I also experimented with brownie and python console on how to build a smart contract, deploy them and interact with other smart contracts to learn how the smart contracts work. I verified Slither and Mythril on different platforms, Kali Linux, Ubuntu, and Windows OS with various different datasets.
Testing with Brownie gives all the information about the smart contract such as the coverage report, methods invoked, and the gas used. The report of vulnerabilities can be viewed through Brownie GUI. I ran tests for re-entrancy detection using Slither and Mythril tool on the dataset mentioned above and verified the efficiency of the tool, the results that I got upon running the tests are about the vulnerabilities present in the code in text form in the terminal, this tool tests each line of the smart contract dataset code written in solidity language. Both tools shows all the lines of code of the smart contract file where the possibility of reentrancy vulnerability tool is detected.

For this project I have used Brownie and web3 to understand the logic of smart contract implementation and I have verified the efficiency of two tools Slither and Mythril with some debugging and modification. If I get a chance to work on this project again, I will develop and implement a tool of my own which would detect the reentrancy with more efficiency than the Slither and Mythril tool also, I will make it so that the false positives and false negatives are less than all other tools that are already present. Finally, I will build such that it is able to analyse all versions of solidity code and all the smart contracts present in the dataset at once to be more efficient.

8 References

Bragagnolo, S. a. (2018). SmartInspect: solidity smart contract inspector. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)* (pp. 9-18). Campobasso, Italy: IEEE.

Cecchetti, E. a. (2021, 05 26). Compositional Security for Reentrant Applications. *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 1249-1267.

christianb93. (2021, 08 18). *Fun with Solidity and Brownie*. Retrieved from LeftAsExercise: https://leftasexercise.com/2021/08/18/fun-with-brownie/

christianb93. (2021, 08 22). *Using web3.py to interact with an Ethereum smart contract*. Retrieved from LeftAsExercise: https://leftasexercise.com/2021/08/22/using-web3-py-to-interact-with-a-smart-contract/

ConsenSys. (2020, 03 23). *Mythril*. Retrieved from GitHub: https://github.com/ConsenSys/mythril

Dannen, C. (2017). Solidity Programming. In T. Green (Ed.), *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain* (pp. 69-88). Brooklyn, New York, USA: Apress Media.

Eshghie, M. a. (2021). *Dynamic Vulnerability Detection on Smart Contracts Using Machine Learning*. New York, NY, USA: Association for Computing Machinery.

Fatima Samreen, N. a. (2020). Reentrancy Vulnerability Identification in Ethereum Smart Contracts. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)* (pp. 22-29). Ryerson University, Toronto, ON, Canada: IEEE.

Feist, J. a. (2019). Slither: A Static Analysis Framework for Smart Contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software*

Engineering for Blockchain (WETSEB) (pp. 8-15). Montreal, QC, Canada: IEEE Press.
Groce, A. a. (2019). What are the Actual Flaws in Important Smart Contracts (And How Can We Find Them)? In J. Bonneau (Ed.), *Financial Cryptography and Data Security* (Vol. abs/1911.07567, pp. 634-653). New York City, NY, USA: Springer.
Li, Z. a. (2021). *SolSaviour: A Defending Framework for Deployed Defective Smart Contracts.* New York, NY, USA: Association for Computing Machinery.
Lin, W.-T. a.-W. (2020). *Runtime Hook on Blockchain and Smart Contract Systems.* New York, NY, USA: Association for Computing Machinery.
Liu, C. a. (2018). *ReGuard: Finding Reentrancy Bugs in Smart Contracts.* New York, NY, USA: Association for Computing Machinery.
Luu, L. a.-H. (2016). Making Smart Contracts Smarter. (pp. 254-269). New York, NY, USA: Association for Computing Machinery.
Mossberg, M. a. (2019). Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 1186-1189). New York City, USA: IEEE Press.
Mueller, B. (2021, 08 06). *Mythril Documentation.* Retrieved 04 03, 2022, from https://readthedocs.org/projects/mythril-classic/downloads/pdf/master/
P. Qian, Z. L. (2020). Towards Automated Reentrancy Detection for Smart Contracts Based on Sequential Models. *IEEE Access, 8*, 19685-19695.
Pan, Z. a. (2021). ReDefender: A Tool for Detecting Reentrancy Vulnerabilities in Smart Contracts Effectively. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)* (pp. 915-925). Jiangsu Province, P.R. China: IEEE.
Tang, Y. a. (2021). Rethinking of Reentrancy on the Ethereum. In *2021 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)* (pp. 68-75). HongKong, China: IEEE.
Tantikul, P. a. (2020). Exploring Vulnerabilities in Solidity Smart Contract. *Proceedings of the 6th International Conference on Information Systems Security and Privacy (ICISSP 2020)*, pp. 317-324.
Tikhomirov, S. a. (2018). *SmartCheck: Static Analysis of Ethereum Smart Contracts.* New York, NY, USA: Association for Computing Machinery.
Torres, C. F. (2018). Osiris: Hunting for integer bugs in ethereum smart contracts. New York, NY, USA: Association for Computing Machinery.
trailofbits. (2022, 04 21). *Slither*. Retrieved from GitHub: https://github.com/crytic/slither#how-to-install
Tsankov, P. a.-C. (2018). *Securify: Practical Security Analysis of Smart Contracts.* New York, NY, USA: ACM.
Xue, Y. a. (2020). *Cross-Contract Static Analysis for Detecting Practical Reentrancy Vulnerabilities in Smart Contracts.* New York, NY, USA: Association for Computing Machinery.
Ye, J. a. (2020). *Clairvoyance: Cross-contract Static Analysis for Detecting Practical Reentrancy Vulnerabilities in Smart Contracts.* New York, NY, USA: Association for Computing Machinery.