Ethereum Smart Contracts

Devansh Patel(40219664)
Information System Security
Concordia University
Montreal, Canada
pa_devan@live.concordia.ca

Darshit Gajjar(40224405)
Information System Security
Concordia University
Montreal, Canada
d_gajj@live.concordia.ca

Rushi Chandalia(40231722)
Information System Security
Concordia University
Montreal, Canada
ru_chan@live.concordia.ca

Kamal Sharma(40219383)
Information System Security
Concordia University
Montreal, Canada
kamal.sharma@mail.concordia.ca

Palak Kaur Sodhi(40205409)
Information System Security
Concordia University
Montreal, Canada
p_sod@live.concordia.ca

Sachin Verma(40231728)
Information System Security
Concordia University
Montreal, Canada
v_sachi@live.concordia.ca

Gulshan Joshi(40219040)
Information System Security
Concordia University
Montreal, Canada
gu_joshi@live.concordia.ca

Varsha Vivek(40238511)
Information System Security
Concordia University
Montreal, Canada
va_vivek@live.concordia.ca

***Abstract*— To provide flash loan to users, the Unstoppable and Naive Receiver contract was created. In this paper, we thoroughly examined both contracts and found a number of weaknesses that attackers could take advantage of. We discovered that the contract is particularly susceptible to denial-of-service and reentrancy attacks. We also suggested a number of mitigation techniques, such as building fail-safe methods and providing input validation, to address these issues. Overall, our findings emphasize the significance of carrying out exhaustive security evaluations and putting in place suitable security controls to guarantee the security of DeFi applications.**

Keywords—smart contracts, ethereum, blockchain, vulnerability, code analysis, exploitation.

I. Introduction

A smart contract is a self-executing commitment with integrated lines of code that define the parameters of the agreement between both the contract's counterparties. A smart agreement serves as a digital replica of the traditional paper contract that actively maintains and carries out its conditions. The smart contract is performed over a blockchain system, and the network's various computers, each contain a copy of the contract's script. This guarantees a safer and more open implementation and execution of the contract terms. Furthermore, since the code of a smart contract is checked by every member of the blockchain network, smart contracts do not need an intermediary to be validated. The cost to counterparties is diminished significantly by eliminating the intermediary from the deal. Blockchain technology serves as the foundation for the notion of smart contracts.

A blockchain is a distributed network made up of a continuously expanding list of records (blocks) connected by encryption. Unlike a traditional database, a blockchain technology does not have a single central location. Each machine on the network has permission to view the information that is recorded in the blockchain. The network is consequently less susceptible to mistakes or attacks. A data on one device cannot be changed in a blockchain without also updating the identical information on other computers in the network. With a blockchain, transactions are organised into blocks that are connected by a chain. Only once the preceding block is finished is a new block formed. Each block comprises a cryptographic hash of the preceding block and is presented in a linear chronological sequence.

There are multiple steps involved in working of smart contracts. The contract's parameters should initially be decided by the counterparties. The completed contractual requirements are then converted into a computer program. In essence, the program contains a variety of conditional statements that outline several circumstances for a potential future transaction. As a piece of code is written, it is copied across the blockchain's users and saved in the network. The code is then compiled and executed by all machines on the network. The appropriate transaction is carried out if a condition of the contract is met and has been confirmed by every user of the blockchain network.

Code analysis plays a major role in this process. Basically, code analysis investigates programming code to discover potential pitfalls or mistakes, assure compliance with coding standards, and guarantee quality. Code analysis can be done statically (viewing the program without running it) or dynamically (executing it and evaluating its performance). There are several tools and methodologies for doing code review.

II. Context

A. Common Types of Vulnerabilities in Ethereum Smart Contracts:

Smart contracts, compared to numerous other contracts, are mainly focused on monetary assets. As a result of the Blockchain's immutability, failures in smart contracts cannot be corrected after they've been implemented. Smart contract flaws might be harmful to security, as well as a tempting aim for suspicious computer hackers. In fact, regardless of whether there are outer manipulators, there is a risk of investment breakdown and economic difficulties in certain instances.

Here are some common vulnerabilities:

Reentrancy:

A risk in which an intruder can approach a contract feature recurrently before the preceding call has finished, likely to result in the intruder trying to execute their own script within the contract.

Integer overflow and underflow:

A form of security vulnerability in which a computation generates an outcome that is greater or less than the highest or lowest value that can be kept in a variable, resulting in malicious activity.

Unused variables:

Variables that are declared but not utilized can imply a coding error and may result in possible errors.

Uninitialized storage pointers:

A threat in which a memory pointer is not initialised prior to application, resulting in errors.

Function visibility:

Proclaimed public functions can be called by anybody, which includes hackers, which could also ultimately lead to security flaws.

Replay signatures attacks:

Signatures allow one account to transfer payments from another account to the blockchain. The actual account will sign a message, and the delivery account will transmit it to a smart contract so that the money transfer fees are paid by the shipping account rather than the main account.

Block gas limit vulnerability:

The block gas limit assists in preventing blocks from becoming too large. If a transfer of funds absorbs excessive gas, it's unlikely to fit in the block and will thus be ignored. Therefore, if information is maintained in arrays and then retrieved via loops over such arrays, the exchange may exhaust its gas and receive a refund. It might result in a denial-of-service (DoS) attack.

Using the Block Hash Function:

By using the block hash function, comparable to the timestamp reliance, is a strategy of attempts to hack smart contracts. It is not suggested that it be utilized for vital parts for the identical purpose as timestamp dependency: miners can modify such features and alter the funds that are withdrawn to their own benefit. This would be particularly apparent whenever the block cipher is employed as a generator of arbitrary numbers.

Incorrect Calculation of the Output Token Amount:

A large variety of actions in the contract argument are linked to token transactions to and from the contract. It opens a wide range of possibilities for errors related to calculating appropriate proportions, service charges, and revenues. That is why it is critical to work with a trustworthy token advancement business to guarantee the achievement of your endeavour.

The following are the most common errors: Incorrect digits processing, especially when interacting with a token like USDT; inaccurate command of procedure during service charge estimations, leading to substantial precision failure; and the precision incessant that was genuinely neglected in the maths processes.

B. Overview of Code Analysis

Code analysis can be done using a variety of tools and methods. They can be roughly divided into two types: static code analysis and dynamic code analysis.

Before releasing a code, static code analysis is performed to discover numerous common coding issues. It implies manually inspecting the code or using tools to automate the procedure. Analysis of static code software can examine a program without running it. They could detect syntax errors, ensure coding guidelines are adhered to, and seek out possible data breaches. It is frequently conducted before the code is deployed, at an early stage of development. It can aid in the identification of difficulties that may be hard to miss during diagnostics and save both money and time by detecting issues prior to become quite severe.

Dynamic code analysis comprises deploying the code and observing its performance. They can detect runtime issues, test the program's efficiency, and ensure that it operates correctly. It is usually done after the program has been published. It can detect issues that were not obvious throughout assessment, such as performance problems or security flaws that arise only under certain circumstances.

C. Smart Contract Code Analysis Tools

Oyente:

Oyente is a security analytical technique for Ethereum smart contracts that is publicly available. It employs optimization technique to investigate all possible smart contract execution paths and identify potential security flaws such as integer overflows, reentrancy issues, and other flaws. Development teams can use Oyente to check the security of their smart contracts prior to deploying them to the Ethereum blockchain, ensuring that the contracts are protected and feature as destined. Security researchers use Oyente to find flaws in smart contracts that are currently utilised to the Ethereum blockchain. Oyente can spot possible security flaws as well as provide advice on how to fix them by analysing the contract's bytecode. This is especially true for

smart contracts that manage large amounts of money, like the ones utilized in decentralised finance (DeFi) applications.

However, Oyente is no longer actively retained and has been primarily supplanted by other safety analysis software such as Mythril and Manticore, which offer more useful capabilities and greater services for the newest version of the Solidity computer language.

Slither:

A smart contract analyzer called Slither is free and statically evaluates the program. Solidity Abstract Syntax Tree (AST) is obtained from the source code of Solidity Compiler and is accessible to Slither. The Slither tool is made up of several analyzers that can find weaknesses in the contract. Additionally, it offers ideas for bettering the code and provides rich visual data about the contract. Slither offers a communication API. Abstract Syntax tree is a data architecture that is commonly used with compilers to portray code structure. The abstract syntax tree (AST) is a tree structure which symbolizes the source code in a technical way. So, every link in the tree is a code construct.

Slither generates a human-readable overview of the contract. It also recognises and reflects the contracts acquired by a particular contract using a diagram. Slither is published in Python, so it requires Python 3 to install and run. Analysis tools can produce false positives as well as false negatives. A false positive is when a tool claims a problem that is not essentially a mistake. False negatives are mistakes that go completely unnoticed. According to a number of similar research, several flaws may be identified in concept by static analysis techniques but are not discovered in practice due to tool restrictions.

The structure is presently utilized for the specified objectives: A wide range of smart contract errors can be discovered with no requirement for human interference or extra configuration work. Slither finds code optimization techniques that the compiler does not. Slither sums up and showcases contract details to help you in your runtime environment research. Slither can be interacted with via its API. Slither can identify various security flows such as reentrancy, function visibility, uninitialized storage pointers, unused variables and integer overflow and underflow.

Slither is a Python package that needs Python 3+ to be downloaded on the system. It pairs well with the tool solc-select, which allows users to select from multiple variations of the Solidity compiler. We can run the following lines in the terminal to install slither and solc-select.

$ pip3 install slither-analyzer

$ pip3 install solc-select

Securify:

Securify is an Ethereum smart contract security analysis tool. It is employed to detect potential security flaws in smart contracts prior to their deployment to the Ethereum blockchain. To recognise possible safety risks such as reentrancy attacks, integer overflows, and other common security flaws, the tool uses a mixture of static and dynamic analysis methods. Programmers can use Securify to check the stability of their smart contracts during the development process, ensuring that the contracts are stable and feature as intended. Security experts may additionally employ the device to identify flaws in smart contracts that are currently utilized to the Ethereum blockchain.

Securify outperforms other security analytical techniques due to its incorporation with Remix, a famous web-based application framework for Ethereum smart contracts, interoperability with the newest version of the Solidity programming language, and advanced analytical skills.

Mythril:

Mythril is an open-source vulnerability assessment tool for analyzing smart contracts published in Solidity, the Ethereum blockchain's most widely used programming dialect for smart contracts. It is intended to identify possible security flaws in smart contracts and make proposals to enhance the code. Mythril operates through symbolic execution, a method for analyzing a program's behavior by discovering all available options it can take. Mythril examines the bytecode of the smart contract and creates a control flow diagram that depicts all different possibilities that the code can accept.

Mythril can identify various vulnerabilities like Reentrancy, Integer overflow and underflow, Out of gas which is a threat in which a contract consumes more gas than what is available, resulting in the transfer of funds failing and the last one is solidity compile bug which is a risk caused by a virus in the Solidity compiler that can result in unforeseen actions.

Setup of Mythril on various systems:

PyPI on Mac OS:

brew update

brew upgrade

brew tap ethereum/ethereum

brew install solidity

pip3 install mythril

PyPI on Ubuntu:

Update

sudo apt update

Install solc

sudo apt install software-properties-common

sudo add-repository ppa:ethereum/ethereum

sudo apt install solc

Install libssl-dev, python3-dev, and python3-pip

sudo apt install libssl-dev python3-dev python3-pip

Install mythril

pip3 install mythril

myth version

Manticore:

Manticore is a symbolic execution tool that analyzes smart contracts and binary files. It is primarily used mostly for Ethereum smart contract security assessment, allows programmers and security experts to recognise security flaws in the code and enhance the contract's security level. Manticore utilizes symbolic execution to investigate all plausible smart contract execution paths and recognise possible safety risks such as integer overflows, reentrancy vulnerabilities, and other popular smart contract security flaws. Manticore is suitable for performing tasks apart from security analysis, such as bug identification, test case generation, and contract validation. Manticore may be employed as a stand-alone tool or merged into other tools and procedures like the Truffle framework, Remix IDE, and other Ethereum design tools. This tends to make it a flexible and potent tool for optimizing the safety of smart contracts.

All in all, Manticore is a strong and adaptable tool for analyzing Ethereum smart contracts. It employs symbolic execution to investigate all possible execution paths of a contract and identify potential security vulnerabilities, making it an indispensable tool for developers and security researchers seeking to ensure the security and reliability of their smart contracts.

Manticore's procedure can be categorized as follows:

Manticore analyzes the bytecode or source code of a smart contract as input to identify the various functions and variables in the contract. Manticore employs symbolic execution to investigate all potential available options of the smart contract, beginning with the contract's initial state. Manticore creates a collection of constraints that reflect the circumstances needed for executing each route as it examines various operation pathways. Manticore then employs a constraint solver to ascertain whether every collection of limitations is acceptable, — in other words, how an input collection remains that fulfils the restrictions. If Manticore discovers a collection of requirements that is satisfactory, it indicates that the contract contains a possible vulnerability that an intruder could manipulate. Eventually, Manticore produces an analysis that enumerates all of the possible risks found in the smart contract, as well as suggestions regarding how to solve them.

III. Environmetnal Setup

This section entails a comprehensive description of the experimental conditions and tools used in the implementation. The environmental setup for this project involved the use of several software tools and platforms for smart contract development and analysis. Firstly, MetaMask was utilized as a digital wallet for storing and managing Ethereum cryptocurrency. MetaMask is a browser extension that lets users manage their Ethereum accounts and interact with Ethereum decentralized applications (dApps) right from their browser. It acts as a gateway between the user's browser and the Ethereum network, allowing them to manage their digital assets and interact with dApps without the need for a full node.

Secondly, Etherscan was used as a blockchain explorer to track and verify Ethereum transactions. Etherscan allows users to browse and search for information about transactions, smart contracts, and addresses on the Ethereum blockchain. It provides a user-friendly interface for users to view and analyze blockchain data, monitor their Ethereum account balances, and track the status of their transactions.

Thirdly, Brownie was used as a smart contract development and testing framework to build and deploy Solidity contracts on the Ethereum network. It provides a comprehensive set of tools and libraries for building decentralized applications (dApps) and interacting with the Ethereum network, including contract deployment, testing, debugging, and gas optimization. We can install brownie using pip, the Python package manager, using the command “pip install eth-brownie”. Once installed, a new project can be created using “brownie init”. New smart contract files can be created in the “/contracts” directory, which can be coded using Solidity, which is one of the popular programming languages for Ethereum smart contracts. After writing the code, it can be compiled using the command “brownie compile”. Once compiled, it can be deployed to the Ethereum network using the brownie CLI.

Fourthly, Mythril was employed as a security analysis tool to detect vulnerabilities in the smart contract code. In addition, Forge was used as a smart contract vulnerability scanner to automatically identify and fix security issues in the code.

Lastly, Slither was utilized as a static analysis tool to detect and prevent vulnerabilities in the Solidity smart contracts’ code before deployment. It is designed to help developers and auditors identify potential security vulnerabilities in Solidity smart contracts before they are deployed to the blockchain. Slither uses a combination of heuristics, taint analysis, and symbolic execution techniques to analyze the bytecode of a smart contract and identify

potential issues. It can detect common vulnerabilities such as reentrancy, integer overflow and underflow, uninitialized storage pointers, and more.

All these tools were integrated into a unified environment that allowed for seamless testing and analysis of smart contracts on the Ethereum blockchain. The study design and implementation were carried out in accordance with best practices in smart contract development and security analysis, with rigorous attention paid to the environmental setup and configuration.

IV. Analysis and Exploit

A. *Contract 1 : Flash Loan Denial of Service*

Smart contracts are hard to keep secure because they are immutable and censorship resistance ie a smart contract having a bug is deployed in a blockchain then it cannot be corrected and will remain there waiting for exploitation

About the exploit:

It is a denial-of-service hack on a flash loan in a smart contract. In this environment, a certain amount of loan can be availed by anyone but must be returned within the same transaction. This is coded in such a way that if the loan is not reverted in the same transaction, then payment will not go through in the first place. There's a tokenized vault with a million DVT tokens deposited. It's offering flash loans for free, until the grace period ends.

Objective:

To complete the attack successfully we need to make the vault stop offering flash loans.

About the code:

There are 2 smart contracts in this project :

1. ReceiverUnstoppable.sol
2. UnstoppableLender : This code defines a smart contract that provides flash loans.

Setup:

We will be using the Brownie framework which is based on python to carry out this deployment and attack.

Brownie can be installed by following these steps :

It is recommended that Brownie be installed through Pipex, it allows us to run end user applications written in python.

Therefore, first step is installation of pipex :

python3 -m pip install --user pipx

python3 -m pipx ensurepath

After restarting terminal, we install brownie :

$ pipx install eth-brownie

verification can be done with the following command, a similar output will be visible on the screen :

$ brownie

Brownie - Python development framework for Ethereum

Usage: brownie <command> [<args>...] [options <args>]

Exporting code from git

After setting up Brownie framework, we will be cloning the code from git using the *git clone* command and with the help of VS Code we start working.

1. *Deploying the contract*

Deployment of contract will be done with the help of Metamask which is a plugin for internet browser (Firefox in this project.) MetaMask is a software cryptocurrency wallet used to interact with the Ethereum blockchain. Users are able a chrome extension or mobile app to retrieve their Ethereum wallet, and then utilize it to connect with decentralized applications. Following is a screenshot of one of the accounts created for this project.

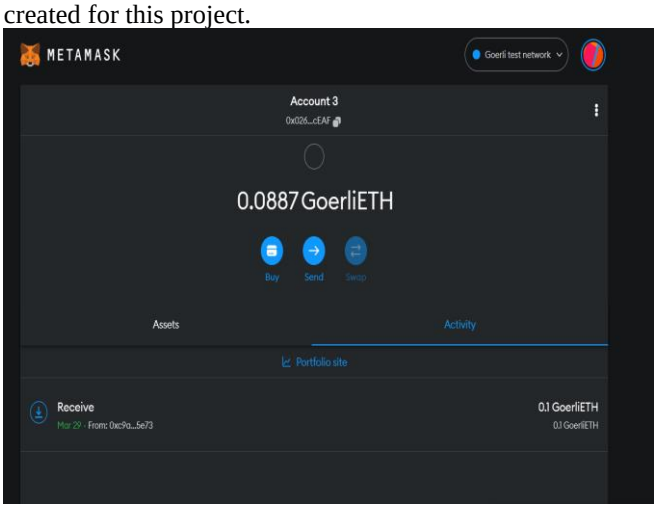


Figure 1Metamsk Interface

Goerli Test network is used to simulate the block chain and for testing of our contracts. To connect to Goerli network, Brownie tool is used which can be installed on Linux or Windows machine (in VSC code on windows for this case). Following commands are used to deploy the contract and connect to test network.
brownie console –network goerli (this command opens a brownie python shell)

Next, we import the private keys of accounts that were setup previously. The following command was used:

accounts.add(config["wallets"]["attacker"])
accounts.add(config["wallets"]["deployer"])
accounts.add(config["wallets"]["someUser"])

Further, we assign variables to these accounts, with the following commands:

Deployer = accounts[0]
Attacker = accounts[1]
someUser = accounts[2]

```
>>> accounts.add(config["wallets"]["deployer"])
<LocalAccount '0x01422dDD01a8fEDf34D31Dbc8064fe460c00d430'>
>>> accounts.add(config["wallets"]["attacker"])
<LocalAccount '0x491dE9924Da3a2816F29Ce9096dAFa5b4BA6aE44'>
>>> accounts.add(config["wallets"]["someUser"])
<LocalAccount '0x026262fc42256Aec8dE0ff8bfa6C884b6A30cEAF'>
>>> token = DamnValuableToken.deploy({'from': deployer})
```

Figure 2 Setting up Wallets.

Lastly, we deploy the damnValuableToken smart contract which will also initialize the DamnValuableTokens (DVT)with the following command to all accounts:

token = DamnValuableToken.deploy({"from":deployer}) // creation of DVT

Deployed contract on Etherscan:

Figure 3 Deployed Contract

Now 'damnValuableToken' contract will be deployed. This essentially mints the 'damnValuableToken' to whoever deploys the contract.

Following steps has been used:

First, we describe the acceptable behavior of the contract.

Deployment of UnstoppableLender contract; it has 2 functions, depositing tokens and executing flash loans. Deployment is done with the following command:

Pool = unstoppableLender.deploy(token.address,{"from": deployer})

Post deployment we transfer tokens to the pool with the following command. Here we are transferring 1 million DVT:

Pool.depositToken(1000000000000000000000000, {"from":deployer})

Now we deploy the flash loan through the following command:

receiveContract = ReceiverUnstoppable deploy (pool address, {"from": deployer})

To test the normal behavior, we execute the following command:

receiveContract.executeFlashLoan(1000000000000000000 0000000, {"from": deployer})

Below is the snippet of the above commands being run:

```
>>> token = DamnValuableToken.deploy({'from': deployer})
Transaction sent: 0x2641b00dfa84709073e6ed8ec215a78b1fc1aacf5e5ca3b69241dc8ff914ffc6
  Gas price: 41.944164801 gwei   Gas limit: 800085   Nonce: 4
  DamnValuableToken.constructor confirmed   Block: 8740804   Gas used: 727350 (90.91%)
  DamnValuableToken deployed at: 0xCF5f15c3ce0D5d37C4c884D4C1036E5D023E482b

>>> pool = UnstoppableLender.deploy(token.address, {'from': deployer})
Transaction sent: 0x7cb3e4e1ae91e64006309af9ebe02fb9eeada7cb2bebadf8f0e2f56076088c94
  Gas price: 43.945172791 gwei   Gas limit: 467086   Nonce: 5
  UnstoppableLender.constructor confirmed   Block: 8740806   Gas used: 424624 (90.91%)
  UnstoppableLender deployed at: 0xd40F1EDa45Ffe68386d34f13C772CdfA891C5aD7

>>> token.approve(pool.address, 1000000000000000000000000, {'from': deployer})
Transaction sent: 0xb7dcfd8d721b364ad1c6fd401e68cbaa072ea1644c9b5518e3a27eb5aad96151
  Gas price: 38.41807152 gwei   Gas limit: 50801   Nonce: 6
  DamnValuableToken.approve confirmed   Block: 8740809   Gas used: 46183 (90.91%)

<Transaction '0xb7dcfd8d721b364ad1c6fd401e68cbaa072ea1644c9b5518e3a27eb5aad96151'>
>>> pool.depositTokens(1000000000000000000000000, {'from': deployer})
Transaction sent: 0x45fd088eca1a21c007f20c597fd2ce2c7da49cb5d26741547eac57462510c15c
  Gas price: 38.180420532 gwei   Gas limit: 101091   Nonce: 7
  UnstoppableLender.depositTokens confirmed   Block: 8740810   Gas used: 82111 (81.22%)

<Transaction '0x45fd088eca1a21c007f20c597fd2ce2c7da49cb5d26741547eac57462510c15c'>
>>> receiveContract = ReceiverUnstoppable.deploy(pool.address, {'from': deployer})
Transaction sent: 0xca65ca9a912e73b95b639b2426b3843a6a4a1d703e807b6556150d03908b7f6a
  Gas price: 36.183760181 gwei   Gas limit: 272104   Nonce: 8
  ReceiverUnstoppable.constructor confirmed   Block: 8740811   Gas used: 247368 (90.91%)
  ReceiverUnstoppable deployed at: 0x10eE10C7e6cC71743C9eFbA74Dab49a7882206Fc
```

Figure 4 Snippet of the above commands after running

When this transaction is viewed in Etherscan we can see that a loan was received from an address and was credited back to the same address within the same transaction which is the normal acceptable behavior(Refer to the screenshot below).

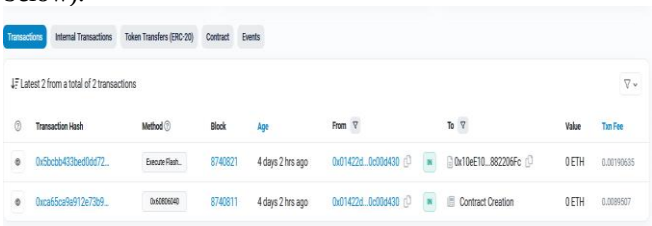


Figure 5 Transaction on Etherscan

2. *Analysis of Smart Contract:*

For analysis Mythril and Slither tools are used. Both tools are generally used for Ethereum smart contract analysis.

Mythril:

It is a security analytical tool for Ethereum smart contracts. It debuted at the 2018 HITBSecConf. Many security flaws, such as integer underflows and owner-overwrite-to-Ether-withdrawals, are caught by Mythril. It should be noted that Mythril is designed to locate common vulnerabilities and is unable to identify problems with an application's business logic. Furthermore, because they frequently are unable to examine all potential states of a program, Mythril and symbolic executors are generally flawed.

Command: *myth analyze*

ReceiverUnstoppable.sol was executed as shown below to pinpoint the location of vulnerability.

```
==== External Call To User-Supplied Address ====
SWC ID: 107
Severity: Low
Contract: ReceiverUnstoppable
Function name: executeFlashLoan(uint256)
PC address: 727
Estimated Gas Usage: 3003 - 38934
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: ReceiverUnstoppable.sol:25

pool.flashLoan(amount)
```

Figure 6 Execution of Mythril Tool

As can be seen from the above screenshot. Mythril shows the location of vulnerability in the contract, which is located in *"executeFlashloan()"* function. According to the information provided by the mytril tool "Reentering the contract in an intermediate state may lead to unexpected behavior" we can provide vulnerable arguments to the function to control the execution flow that might lead to the vulnerability. However, the mythril tool does not provide detailed information regarding what arguments are to be supplied to the function.

Slither:

Slither is a Python-based system for contract security that was developed by Josselin Feist, Gustavo Grieco, and Alex Groce in a 2019 paper. To aid in programmer understanding, the Slither platform offers automated risk and optimization identification as well as helpful codebase descriptions.

We run the command *"slither ReceiverUnstoppable.sol"*, the result can be seen in the below screenshot.

```
darsh@ubuntu:~/Desktop/damn-vulnerable-defi-brownie/contracts/unstoppable$ slither ReceiverUnstoppable.sol
INFO:Detectors:
UnstoppableLender.depositTokens(uint256) (UnstoppableLender.sol#22-27) ignores return value by damnValuableTo
r,address(this),amount) (UnstoppableLender.sol#25)
UnstoppableLender.flashLoan(uint256) (UnstoppableLender.sol#29-44) ignores return value by damnValuableToken.
Amount) (UnstoppableLender.sol#38)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
INFO:Detectors:
UnstoppableLender.flashLoan(uint256) (UnstoppableLender.sol#29-44) uses a dangerous strict equality:
        - assert(bool)(poolBalance == balanceBefore) (UnstoppableLender.sol#36)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
```

Figure 7 Analysis using Slither.

As can be seen, Slither tool provides detailed information about the vulnerability. According to the tool the contract is using strict equality condition *"PoolBalance == balanceBefore"* which might lead to the vulnerability if the condition is not satisfied. In addition, the Slither tool also provides reference link that explains this vulnerability. The screenshot is provided below.

Use of strict equalities that can be easily manipulated by an attacker.

Exploit Scenario:

```
contract Crowdsale{
    function fund_reached() public returns(bool){
        return this.balance == 100 ether;
    }
```

`Crowdsale` relies on `fund_reached` to know when to stop the sale of tokens. `Crowdsale` reaches 100 Ether. Bob sends 0.1 Ether. As a result, `fund_reached` is always false and the `crowdsale` never ends.

Figure 8 Reference Link Provided by Slither

3. *Exploit Flow:*

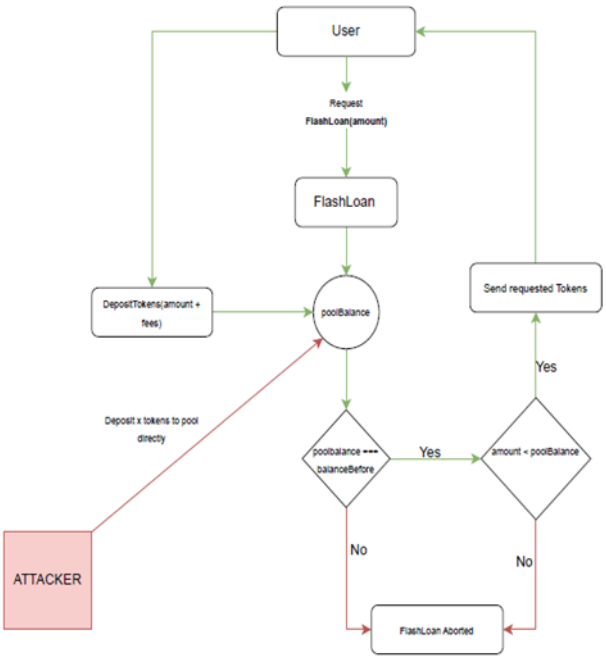


Figure 9 Flowchart of Exploitation

As seen in the Flowchart if the attacker is able to manipulate the condition *"poolbalance == Balancebefore"*, it will lead to the crash of the contract resulting in DOS.
For this logic, token.transfer is used to transfer a single token to the pool. After the transfer, value of *PoolBalance becomes 100000000000000000000* and *BalanceBefore 100000000000000000001*. Thus resulting in *Poolbalance != Balancebefore* and crashing of the contract.

The above logic has been tested using the Forge tool, the below screenshot can be referred for the same.

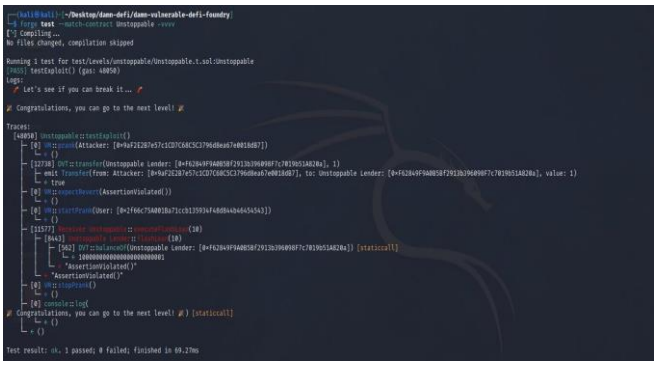


Figure 10 Use of Forge Tool.

Exploit:

Now, the following commands are used in the brownie console to introduce exploit in the contract.

Token.transfer(pool,1, {"from": attacker})

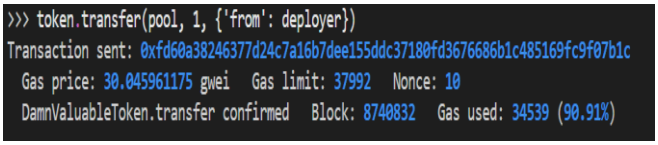


Figure 11 Transfer Single Token in the Pool.

Next, we execute the ExecuteFlashloan() function :

receiveContract.executeFlashLoan(1000000000000000000000000000000, {"from": someUser})

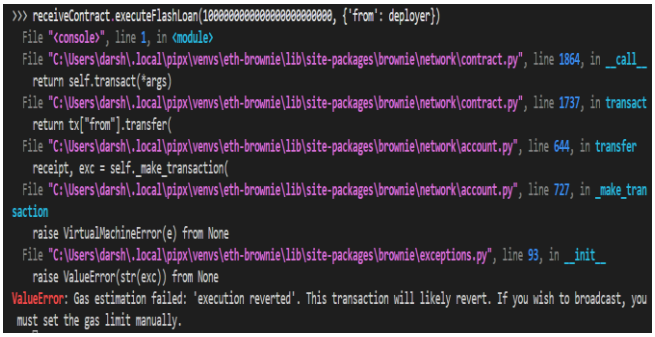


Figure 12 Executing FlashLoan Function.

As seen in the above screenshot after inserting malicious arguments the contract crashes, resulting in DOS.

B. Contract 2 : Flash Loan - Contract Drain

1. Analysis of Smart Contract:

Mythril is a security analysis tool for Ethereum smart contracts. It was introduced at HITBSecConf 2018. Mythril detects a range of security issues, including integer underflows, owner-overwrite-to-Ether-withdrawal, and others. Note that Mythril is targeted at finding common vulnerabilities and is not able to discover issues in the business logic of an application. Furthermore, Mythril and symbolic executors are generally unsound, as they are often unable to explore all possible states of a program.

Command: *myth analyze FlashLoanReceiver.sol* was executed as shown below to pinpoint the location of vulnerability.

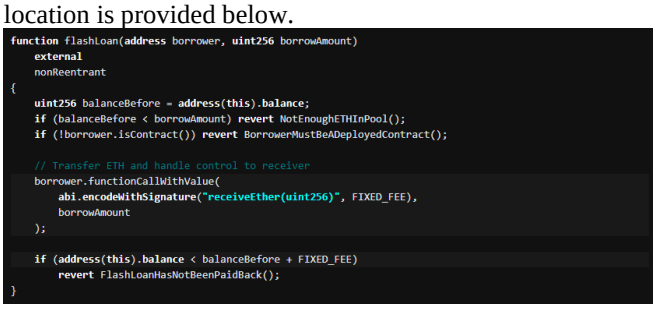
Figure 13 Vulnerability detection using mithril

Here we can see that in analysis it gives an overview of the vulnerability that says that any function can reenter the contract and call the functions without any security check.

For detailed analysis we have used slither tool.
We run command *"slither NaiveReceiverLenderPool.sol" and the output is below:*

Figure 14 Error Detection using Slither.

The above screenshot is taken from analysis done by slither, in which it explains how the FlashLoan function can send ETH to arbitrary users who calls the FlashLoan function. there is no user authentication which leads to this exploitation. Moreover, this tool also gives us the reference link to check the vulnerability. The screenshot of the linked location is provided below.



```
function flashLoan(address borrower, uint256 borrowAmount)
    external
    nonReentrant
{
    uint256 balanceBefore = address(this).balance;
    if (balanceBefore < borrowAmount) revert NotEnoughETHInPool();
    if (!borrower.isContract()) revert BorrowerMustBeADeployedContract();

    // Transfer ETH and handle control to receiver
    borrower.functionCallWithValue(
        abi.encodeWithSignature("receiveEther(uint256)", FIXED_FEE),
        borrowAmount
    );

    if (address(this).balance < balanceBefore + FIXED_FEE)
        revert FlashLoanHasNotBeenPaidBack();
}
```

Figure 15 Reference Link by Slither.

2. Exploit Flow:

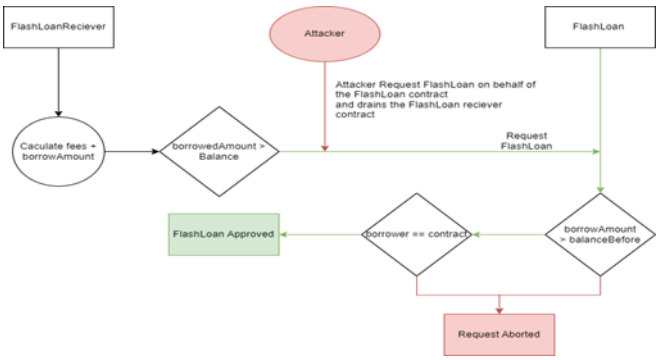


Figure 16 Flowchart of Exploitation.

Lending Pool has 1000 ETH in balance, and there is receiver contract with 10 ETH balance. Lending Pool ask for 1 ETH fees every time someone request the Loan. Here in FlashLoan contract first check if there is enough balance to give out loan, after that it checks if the borrower is a contract, if any of the condition is false, that request aborts else FlashLoan is approved.

Here in Receiver contract, first it calculates total amount to pay back and then checks if the amount is received from pool or not. FlashLoan receiver contract doesn't check who is calling the FlashLoan function on its behalf. This is where attacker can take advantage of exploit and drain the contract by looping and keep requesting the FlashLoan.

The above logic has been tested using the Forge tool, the below screenshot can be referred for the same.

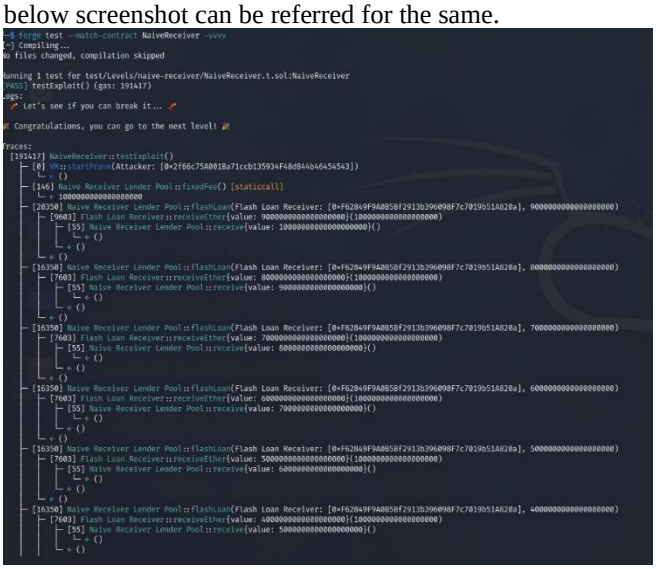


Figure 17 Use of Forge Tool for Testing.

Here, we used forge testing framework to write a exploit function which called the FlashLoan contract on behalf of receiver contract in loop until it drained the receiver contract. as we can see in above figure that FlashLoan is called over and over by our exploit function which drains 1 ETH for every transaction.

V. Conclusion

In conclusion, our analysis of the Unstoppable and Naive Receiver contracts revealed several vulnerabilities that could be exploited by attackers. We were able to successfully test one of these exploits using the FoundryUp toolkit, demonstrating the potential harm that could be caused if these vulnerabilities were left unaddressed. We also deployed the contracts on the Goerli testnet and used analysis tools such as Slither and Mythril to identify potential security issues.

Our findings underscore the critical importance of security assessments in the development and deployment of DeFi smart contracts. As the popularity and value of DeFi applications continue to grow, so does the potential for malicious actors to exploit vulnerabilities for financial gain. It is therefore crucial that developers take a proactive approach to security and conduct rigorous testing and analysis before deploying any code on the mainnet.

Going forward, we recommend that the developers of the smart contracts and other DeFi projects implement robust security measures, such as input validation and fail-safe mechanisms, to prevent potential attacks. Additionally, it is important to regularly monitor and update contracts to address any new vulnerabilities that may emerge. By taking these steps, we can help ensure the safety and sustainability of the DeFi ecosystem.

Acknowledgment

The most important details are the contributions and support provided by the group members and professor in the completion of the final paper. The group members' hard work, dedication, and willingness to go above and beyond have been instrumental in the quality and depth of the paper. The professor's wisdom, experience, and expertise have been invaluable in shaping the direction and focus of the research.

References

[1] Installation and Setup — Mythril v0.23.9 Documentation. https://mythril-classic.readthedocs.io/en/master/installation.html J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[2] "Smart Contract." Corporate Finance Institute, https://corporatefinanceinstitute.com/resources/valuation/smart-contract/. K. Elissa, "Title of paper if known," unpublished.

[3] Academy, Kerala Blockchain. "Code Analysis Tools for Solidity Smart Contracts: An Overview." Medium, 20 Jan. 2023, https://kbaiiitmk.medium.com/code-analysis-tools-for-solidity-smart-contracts-an-overview-b3642939c195.

[4] "Slither: The Leading Static Analyzer for Smart Contracts." Trail of Bits Blog, 27 May 2019, https://blog.trailofbits.com/2019/05/27/slither-the-leading-static-analyzer-for-smart-contracts/.

[5] ImmuneBytes. Top 10 Smart Contract Vulnerabilities: That Can Lock Your Crypto Assets. 29 July 2022, https://www.immunebytes.com/blog/smart-contract-vulnerabilities/

[6] "9 Common Smart Contract Vulnerabilities [List for 2023]," Blaize, Aug. 3, 2021. https://blaize.tech/article-type/9-most-common-smart-contract-vulnerabilities-found-by-blaize/

[7] "7 Smart Contract Vulnerabilities & How to Prevent Them [2023]," PixelPlex, Dec. 16, 2022. https://pixelplex.io/blog/smart-contract-vulnerabilities/

[8] Molina, Alberto. "Smart Contracts Common Vulnerabilities (Solidity)." Coinmonks, 18 Jan. 2023, https://medium.com/coinmonks/smart-contracts-common-vulnerabilities-solidity-e64c5506b7f4.

[9] Kathoh, David. "Automated Smart Contract Security Review with Slither." *Coinmonks*, 21 Nov. 2021, https://medium.com/coinmonks/automated-smart-contract-security-review-with-slither-1834e9613b01.

[10] "Detector Documentation." GitHub, https://github.com/crytic/slither/wiki/Detector-Documentation. Accessed 3 Apr. 2023.

[11] Slither - Static Analysis Tools - Alchemy. https://www.alchemy.com/dapps/slither.

[12] What Is Mythril? — Mythril v0.23.9 Documentation. https://mythril-classic.readthedocs.io/en/master/about.html.