LNMIIT
The LNM Institute of
Information Technology

Smart Contract Security

Instructor - Dr. Mohit Gupta

Team Members:-

Vasu Gupta
(21ucs230)

Varad Bardapurkar
(21ucs228)

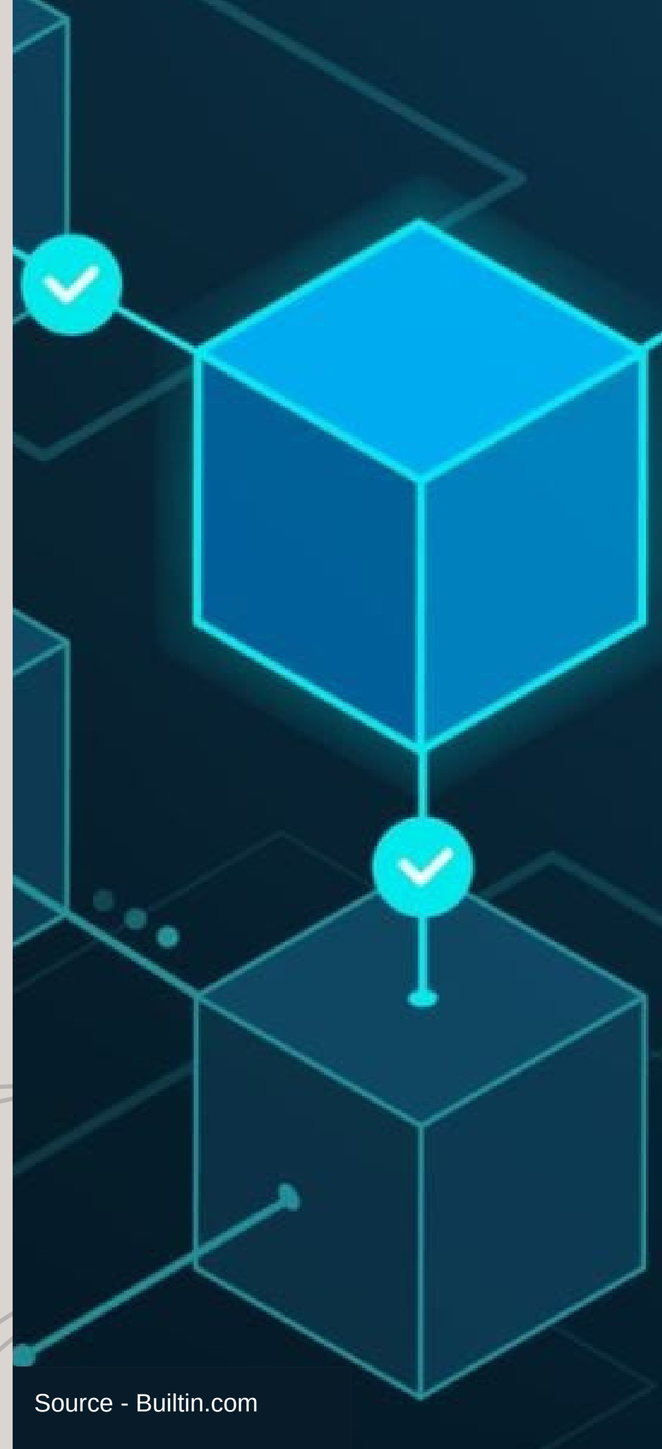
Mihir Mangal
(21ucs128)

Shivamkumar Rewatkar
(21ucs191)

Table of Contents:

Introduction to Blockchain

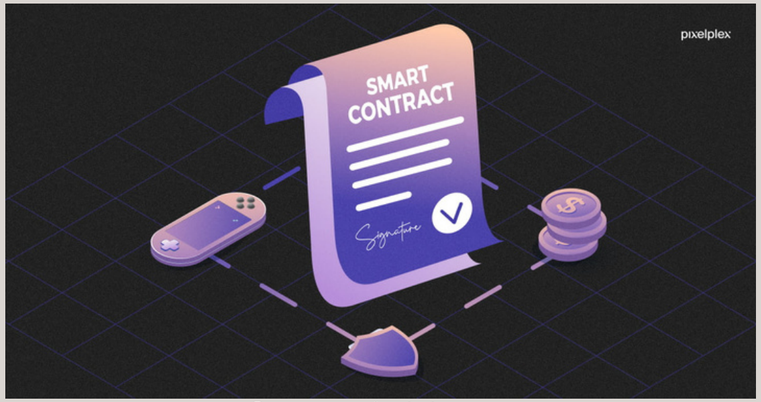
Blockchain is a decentralized digital ledger technology that ensures transparent, trustless and secured transaction between the parties involved.



Source - Builtin.com

Smart Contract Security Overview

Smart contracts are digital contracts stored on a blockchain that are automatically executed when predetermined terms and conditions are met.



Source - pixelplex.com

Damn Vulnerable DeFi

- Damn Vulnerable DeFi [1] is a compilation of challenges aimed to master blockchain security auditing.
- It consists of a set of vulnerable smart contracts to be audited and attacked using Hardhat framework.
- It will be the foundation for our security auditing of live projects and will help us in both manual and automated testing.

Unstoppable Vault

Problem statement

There’s a tokenized vault with a million DVT tokens deposited. It’s offering flash loans for free, until the grace period ends.
To pass the challenge, make the vault stop offering flash loans.
You start with 10 DVT tokens in balance.

Unstoppable Vault

- The goal of the first challenge is to perform a DOS (Denial of Service) [2] attack to the contract.

- Here is a suspicious line in the flashLoan function:

```
uint256 balanceBefore = totalAssets();
if (convertToShares(totalSupply) != balanceBefore) revert InvalidBalance();
```

- If we can manage to alter the poolBalance or the balanceBefore, we will achieve the goal.

- We can easily modify the balanceBefore by sending some token to the pool.

Naive Receiver

Problem statement

There’s a pool with 1000 ETH in balance, offering flash loans. It has a fixed fee of 1 ETH.
A user has deployed a contract with 10 ETH in balance. It’s capable of interacting with the pool and receiving flash loans of ETH.
Take all ETH out of the user’s contract. If possible, in a single transaction.

Naive Receiver

- In this challenge we have to drain all the funds from a contract made to call flash loans.
- The contract expects to be called from the pool, which is fine, but the vulnerability lies on the fact that anyone can call the flash loan function of the pool.
- In order to empty the contract in one transaction, we can create an attacker contract that calls the flash loan multiple times.

```
function onFlashLoan( address, address token, uint256 amount, uint256 fee,
bytes calldata) external returns (bytes32) {
```

Truster

Problem statement

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.
The pool holds 1 million DVT tokens. You have nothing.
To pass this challenge, take all tokens out of the pool. If possible, in a single transaction.

Truster

Here we have to get all the tokens from the pool, and our starter balance is 0. The flashLoan from the pool lets us call any function in any contract. So, what we can do is:

- Call the flashLoan with a function to approve the pool's tokens to be used by the attacker and then call the transferFrom function of the token, to transfer them to the attacker address.

```
token.transfer(borrower, amount);
target.functionCall(data);
```

- If we want to make it in one transaction, we can create a contract that calls the flashLoan with the approve, but instead of the attacker address, we set the created contract address.

- Then we transfer the tokens to the attacker in the same tx.

Side Entrance

Problem statement

A surprisingly simple pool allows anyone to deposit ETH, and withdraw it at any point in time.
It has 1000 ETH in balance already, and is offering free flash loans using the deposited ETH to promote their system.
Starting with 1 ETH in balance, pass the challenge by taking all ETH from the pool.

Side Entrance

- For this challenge we have to take all the ETH from the pool contract.
- It has no function to receive ETH, other than the deposit, which is also the attack vector.
- We can create an attacker contract that asks for a flash loan, and then deposit the borrowed ETH.
- The pool will believe that our balance is 1000 ETH, and that the flash loan was properly paid. Then we can withdraw it.

Side Entrance

```
 function deposit() external payable {
    unchecked {
        balances[msg.sender] += msg.value;
    }
    emit Deposit(msg.sender, msg.value);
}

function flashLoan(uint256 amount) external {
    uint256 balanceBefore = address(this).balance;

    IFlashLoanEtherReceiver(msg.sender).execute{value: amount}();

    if (address(this).balance < balanceBefore)
        revert RepayFailed();
}
```

The Rewarder

Problem statement

There’s a pool offering rewards in tokens every 5 days for those who deposit their DVT tokens into it.
Alice, Bob, Charlie and David have already deposited some DVT tokens, and have won their rewards!
You don’t have any DVT tokens. But in the upcoming round, you must claim most rewards for yourself.
By the way, rumours say a new pool has just launched. Isn’t it offering flash loans of DVT tokens?

The Rewarder

Here we have to claim rewards from a pool we shouldn't be able to.

Rewards are distributed when someone calls distributeRewards(), and depending on the amount of tokens deposited.

So, we can do all of this in one transaction:

- Wait five days (minimum period between rewards).
- Get a flash loan with a huge amount of tokens.
- Deposit the tokens in the pool.
- Distribute the rewards.
- Withdraw the tokens from the pool.
- Pay back the flash loan.

Selfie

Problem statement

A new cool lending pool has launched! It’s now offering flash loans of DVT tokens. It even includes a fancy governance mechanism to control it.
What could go wrong, right ?
You start with no DVT tokens in balance, and the pool has 1.5 million. Your goal is to take them all.

Selfie

The goal of this challenge is to drain all the tokens from the pool.

The pool has a drainAllFunds(address) function that can only be executed by a governance address, and this is what we will be exploiting:

- Request a flash loan and get all the tokens from the pool.
- Take a snapshot of the tokens -> Here lies one vulnerability Anyone can take a snapshot at any time.
- Propose to execute an action to transfer all tokens to the attacker (the proposal will be admited since we have a lot of tokens in the snapshot).
- Return the flash loan tokens to the pool.
- Wait two days (the grace period for the proposal).
- Execute the action to drain all funds.

Compromised

Problem statement

While poking around a web service of one of the most popular DeFi projects in the space, you get a somewhat strange response from their server.

A related on-chain exchange is selling (absurdly overpriced) collectibles called “DVNFT”, now at 999 ETH each.
This price is fetched from an on-chain oracle, based on 3 trusted reporters: 0xA732...A105,0xe924...9D15 and 0x81A5...850c.
Starting with just 0.1 ETH in balance, pass the challenge by obtaining all ETH available in the exchange.

Compromised

- The goal here is to drain all the ETH from the exchange.
- The exchange only has two methods, one to buy a token, and the other to sell it. The price is given by an oracle.
- The oracle [3] is properly initialized and only some trusted sources can update price with the postPrice method.
- We can check the addresses they correspond to the addresses we got from the given hint, and they are in fact the addresses from two of the trusted sources.
- With these keys, we can manipulate the price of the token to buy low and sell high, extracting all the ETH from the contract.

Compromised

```
function _computeMedianPrice(string memory symbol) private view returns (uint256) {
    uint256[] memory prices = getAllPricesForSymbol(symbol);
    LibSort.insertionSort(prices);
    if (prices.length % 2 == 0) {
        uint256 leftPrice = prices[(prices.length / 2) - 1];
        uint256 rightPrice = prices[prices.length / 2];
        return (leftPrice + rightPrice) / 2;
    } else {
        return prices[prices.length / 2];
    }
}
```

Puppet

Problem statement

There’s a lending pool where users can borrow Damn Valuable Tokens (DVTs). To do so, they first need to deposit twice the borrow amount in ETH as collateral. The pool currently has 100000 DVTs in liquidity.

There’s a DVT market opened in an old Uniswap v1 exchange, currently with 10 ETH and 10 DVT in liquidity.

Pass the challenge by taking all tokens from the lending pool. You start with 25 ETH and 1000 DVTs in balance.

Puppet

- The goal of this challenge is to drain all of the tokens from a pool.
- There's a borrow function in the pool that lets people borrow tokens for twice their price in ETH.
- The vulnerability lies on the fact that it is taking the price from a Uniswap pool with very low liquidity.
- So, we can lower the token price of the Uniswap pool by swapping some ETH to tokens.
- Then, when the price is low enough, we can borrow all the tokens from the pool for a very low price.

Puppet

```
function calculateDepositRequired(uint256 amount) public view returns (uint256) {
    return amount * _computeOraclePrice() * DEPOSIT_FACTOR / 10 ** 18;
}

function _computeOraclePrice() private view returns (uint256) {
    // calculates the price of the token in wei according to Uniswap pair
    return uniswapPair.balance * (10 ** 18) / token.balanceOf(uniswapPair);
}
```

Puppet V2

- This challenge has the same issues as the previous one. The price from the pool relies on a single oracle that can be attacked to change the price.

- The attack is the same as in the previous challenge, but instead of interacting with Uniswap v1, in this case it’s v2.

```
function _getOracleQuote(uint256 amount) private view returns (uint256) {
    (uint256 reservesWETH, uint256 reservesToken) =
        UniswapV2Library.getReserves(_uniswapFactory, address(_weth), address(_token));
    return UniswapV2Library.quote(amount.mul(10 ** 18), reservesToken, reservesWETH);
}
```

Free Rider

Problem statement

A new marketplace of Damn Valuable NFTs has been released! There's been an initial mint of 6 NFTs, which are available for sale in the marketplace. Each one at 15 ETH.
The developers behind it have been notified the marketplace is vulnerable. All tokens can be taken. Yet they have absolutely no idea how to do it. So they're offering a bounty of 45 ETH for whoever is willing to take the NFTs out and send them their way.
You've agreed to help. Although, you only have 0.1 ETH in balance. The devs just won't reply to your messages asking for more.
If only you could get free ETH, at least for an instant.

Free Rider

- The goal of this challenge is to send some NFTs from a vulnerable marketplace to a buyer.
- The marketplace contract has a bug in transferring the token from seller to buyer.
- The attack would be get a flash loan from Uniswap to buy NFTs, and return it in a single transaction.

```
if (msg.value < priceToPay)
    revert InsufficientPayment();

DamnValuableNFT _token = token;
_token.safeTransferFrom(_token.ownerOf(tokenId), msg.sender, tokenId);
payable(_token.ownerOf(tokenId)).sendValue(priceToPay);
```

Backdoor

- Gnosis wallet's setup function uses unchecked arguments (to & data) for setupModules.
- setupModules allows a malicious delegatecall modifying the wallet's state.
- We deployed a malicious contract and used setupModules data to manipulate the modules mapping within the wallet.

```
function setup( address[] calldata _owners, uint256 _threshold, address to,
    bytes calldata data, address fallbackHandler, address paymentToken,
    uint256 payment, address payable paymentReceiver
) external {
```

Climber

The challenge is to drain the funds in the vault administered by a Timelock contract. And the vault itself is UUPS upgradable.

There are two main vulnerabilities in the Timelock contract.

- The execute function allows anyone to call, which when properly safeguarded is fine. However, it allows a random low-level call to be executed prior to verify if the operation has been approved.
- The Timelock contract itself is self-administered as well, which is not a red flag but when combined with the first vulnerability allows an attacker to execute administrative actions through 'call' before the operation is verified..

Puppet V3

Lending pool relies on Uniswap v3 TWAP oracle for DVT price.
TWAP uses geometric mean (more resistant to manipulation than Uniswap v2's arithmetic mean).

Here too the main problem is low liquidity of the lending pool.
Split the USDC into multiple smaller swaps over time.
Each swap buys DVT with a portion of the USDC, tipping the TWAP price in our favor.

ABI Smuggling

The challenge asks us to drain all tokens in the vault contract. The vault token also inherits an authorization contract which only allows registered account to execute specific functions.

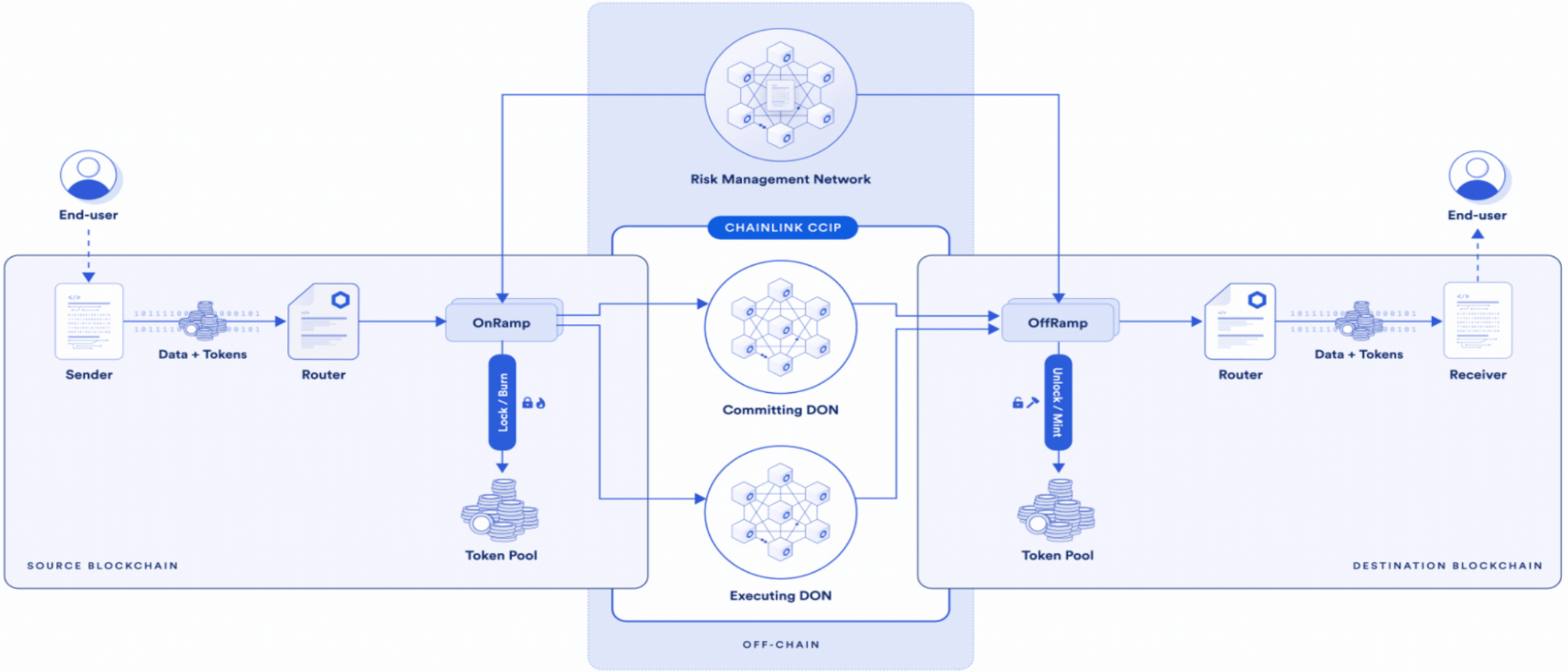
The vulnerability lies in the way the function selector is verified in execute. The function selector is pulled from msg.data at a fixed calldata byte offset uint256 calldataOffset = 4 + 32 * 3, which means that as long as we have the correct function selector in the calldata at this offset, we have a chance to pass.

Chainlink Project : CCIP

What is Chainlink :

- Chainlink [4] CCIP provides a single simple interface through which dApps and web3 entrepreneurs can securely meet all their cross-chain needs.
- CCIP is used to transfer data, tokens, or both data and tokens across chains.
- Streamlines development for dApps requiring cross-chain functionality.

ARCHITECTURE



Source - chain.link/cross-chain

Common Use Cases:

- Cross-chain lending: Chainlink CCIP enables users to lend and borrow a wide range of crypto assets across multiple DeFi platforms running on independent chains.
- Low-cost transaction computation: Chainlink CCIP can help offload the computation of transaction data on cost-optimized chains.
- Optimizing cross-chain yield: Users can leverage Chainlink CCIP to move collateral to new DeFi protocols to maximize yield across chains.
- Creating new kinds of dApps: Chainlink CCIP enables users to take advantage of network effects on certain chains while harnessing compute and storage capabilities of other chains.

Future Work

- After performing manual testing on the live repository we now move towards automated testing.

- We are going to use tools like
  - Static Analysis
  - Fuzzing
  - Formal Verification

- We are going to apply these tools to our Damn Vulnerable CTFs and then use them to audit our live project.

Acknowledgements

[1] https://www.damnvulnerabledefi.xyz/

[2] https://solidity-by-example.org/

[3] https://chain.link/education/blockchain-oracles

[4]https://chain.link/cross-chain/docs

Thank You