Security in Smart Contracts

Léo Gabaix
Jean-Paul Morcos Doueihy
Nina Raganová
Marc Alba

Smart Contracts

Smart Contracts are programs that get **executed automatically** after fulfilling **certain conditions**.

The Smart Contracts are **stored and executed on a blockchain**, such as Bitcoin or Ethereum.

Smart Contracts are widely used in various fields, to name a few:

- Decentralized Finance (DeFi)
- Cryptocurrency Markets
- Electronic Voting Systems

Concerning SC and Security

To ensure and enhance the **robustness and resilience** of blockchain Smart Contracts, developers must address carefully any potential security threat.

Thereupon we will address the following most common security topics impacting developing Smart Contracts:

- Vulnerabilities **by coding**

We'll look into the DAO attack.

- Vulnerabilities **by design**

Overview at SC vulnerabilities

Vulnerabilities by code:

- Reentrancy bugs
- Integer Overflows/Underflows
- Unauthorized access

Vulnerabilities by design:

- Gas Limit and Loops
- Timestamp dependence
- Front Running

Reentrancy bugs

- Situation that allows to continuously withdraw funds from the victim.

- Attacker: smart contract without the ability to receive tokens.

- This mismatch will trigger the fallback function, which receives funds

```
contract TheBank {
    mapping(address => uint) theBalances;

    function deposit() public payable {
        require(msg.value >= 1 ether, "cannot deposit below 1 ether");
        theBalances[msg.sender] += msg.value;
    }

    function withdrawal() public {
        require(
            theBalances[msg.sender] >= 1 ether,
            "must have at least one ether"
        );
        uint bal = theBalances[msg.sender];
        (bool success, ) = msg.sender.call{value: bal}("");
        require(success, "transaction failed");
        theBalances[msg.sender] -= 0;
    }

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

Reentrancy bugs: Guard Solution

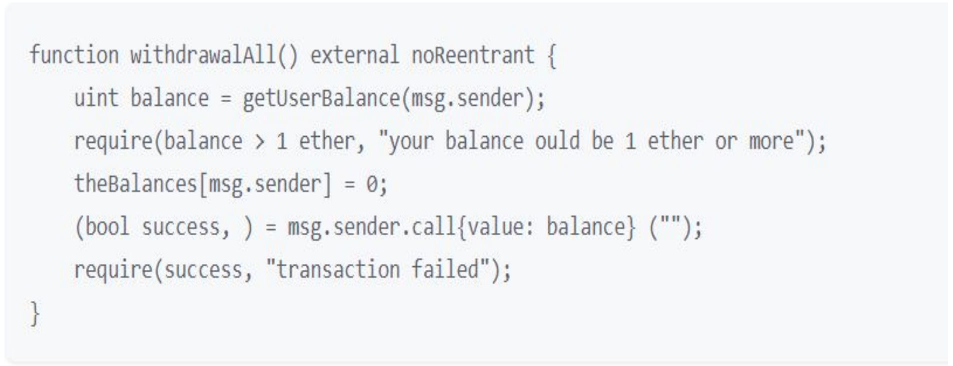
The reentrancy guard solves this as it stops the execution of multiple vital functions at the same time.

```
bool internal locked;

modifier noReentrant() {
    require(!locked, "cannot reenter");
    locked = true;
    _;
    locked = false;
}

function withdrawalAll() external noReentrant {
    uint balance = getUserBalance(msg.sender);
    require(balance > 1 ether, "your balance ould be 1 ether or more");
    (bool success, ) = msg.sender.call{value: balance} ("");
    require(success, "transaction failed");
    theBalances[msg.sender] = 0;
}
```

Reentrancy bugs: Check and Effects Convention

- Immediately update your state after you have set the requirement check.

- Be mindful of the execution flow of your contract.



```
function withdrawalAll() external noReentrant {
    uint balance = getUserBalance(msg.sender);
    require(balance > 1 ether, "your balance ould be 1 ether or more");
    theBalances[msg.sender] = 0;
    (bool success, ) = msg.sender.call{value: balance} ("");
    require(success, "transaction failed");
}
```

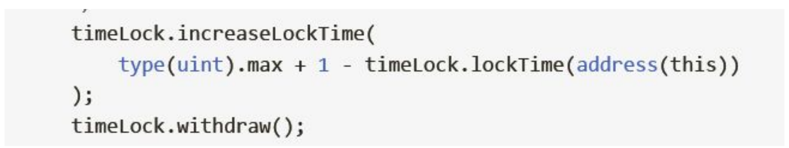
DAO Attack

The **D**ecentralized **A**utonomous **O**rganization Attack(2016): Using a massive reentrancy attack, over the course of a few weeks, the hacker would almost entirely drain the $150 million worth of ETH controlled by The DAO. This attack lead to a hard fork in the Ethereum network, resulting in Ethereum (ETH) and Ethereum Classic (ETC).

Integer Overflow

- Overflow occurs when a calculation produces a number that is larger than the maximum value that can be stored in the allocated space.
- Can be exploited by attackers to manipulate the execution of the contract.

An attacker could:



```
timeLock.increaseLockTime(
    type(uint).max + 1 - timeLock.lockTime(address(this))
);
timeLock.withdraw();
```

Integer Overflow

```
contract TimeLock {
    mapping(address => uint) public balances;
    mapping(address => uint) public lockTime;

    function deposit() external payable {
        balances[msg.sender] += msg.value;
        lockTime[msg.sender] = block.timestamp + 1 weeks;
    }

    function increaseLockTime(uint _secondsToIncrease) public {
        lockTime[msg.sender] += _secondsToIncrease;
    }
```

```
    function withdraw() public {
        require(balances[msg.sender] > 0, "Insufficient funds");
        require(block.timestamp > lockTime[msg.sender], "Lock time not expired");

        uint amount = balances[msg.sender];
        balances[msg.sender] = 0;

        (bool sent, ) = msg.sender.call{value: amount}("");
        require(sent, "Failed to send Ether");
    }
}
```

Integer Overflow mitigation

- SafeMath library: Developers should make sure to use safe math libraries or appropriate types that provide overflow detection. In case of an error, the library fails the transaction and updates the status of the transaction to be reverted.
- Compiler Version: The attack can be prevented by using solidity >= 0.8. Libraries like SafeMath is embedded in the compiled code.
- Use modifier ‘onlyOwner’, additionally, they can also incorporate checks in their code to ensure the values never exceed their expected range.

Timestamp Dependence

Description

Contracts that depend on block timestamps for critical operations are susceptible to manipulation, as miners can slightly adjust the timestamps.

Impact

This can lead to unfair advantages in games, easier puzzle solutions, and flawed randomness, all of which an attacker can exploit.

Timestamp Dependence

Example

```
uint256 constant private salt =  block.timestamp;

function random(uint Max) constant private returns (uint256 result){
    //get the best seed for randomness
    uint256 x = salt * 100/Max;
    uint256 y = salt * block.number/(salt % 5) ;
    uint256 seed = block.number/3 + (salt % 300) + Last_Payout + y;
    uint256 h = uint256(block.blockhash(seed));

    return uint256((h / x)) % Max + 1; //random number between 1 and Max
}
```

Access Control Vulnerabilities

Description

Access control vulnerabilities exist when a contract fails to properly restrict who can call certain functions. This can result in unauthorized function calls.

Impact

If a contract function isn’t protected adequately, unauthorized actors can manipulate the contract state, steal funds, or take other damaging actions.

Access Control Vulnerabilities

Example

```
// This contract is vulnerable to an access control vulnerability

contract AccessControlVulnerable {
    mapping(address => uint256) private balances;

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

    function withdraw(uint256 amount) public {
        // Missing require statement to check balance
        uint256 balance = balances[msg.sender];
        // No check for sufficient balance before withdrawal
        balances[msg.sender] -= amount;
        payable(msg.sender).transfer(amount);
    }

    function getBalance() public view returns (uint256) {
        return balances[msg.sender];
    }
}
```

Front-running Attacks

Description

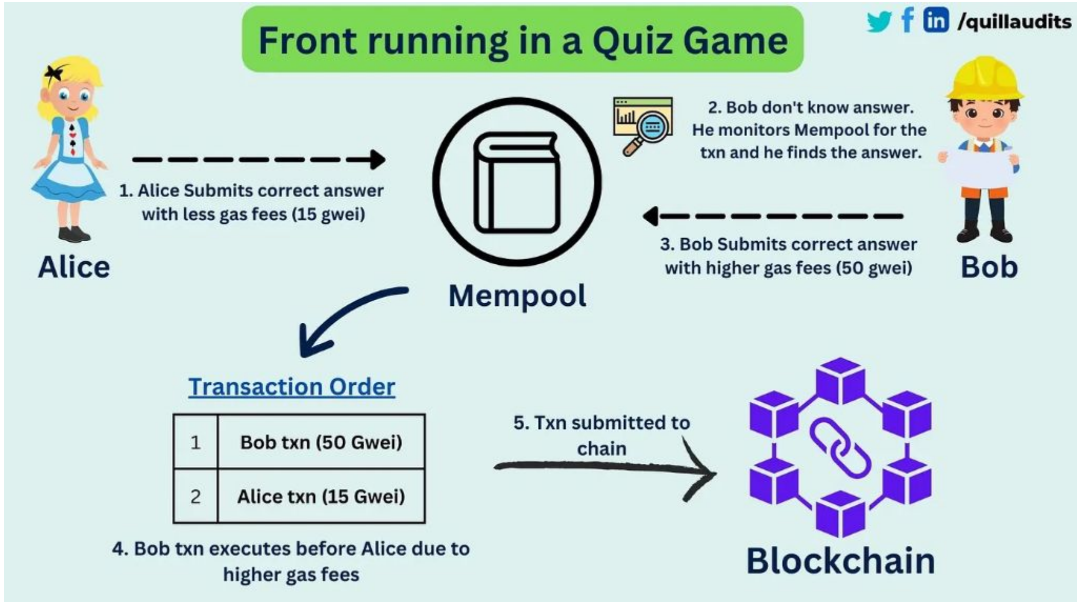
Front-running happens when someone sees a pending transaction and manages to get their own transaction processed first by offering a higher gas price. This is possible in public blockchain networks like Ethereum where transaction data is publicly accessible before being mined.

Impact

Front-running can result in financial loss, as an attacker can intercept and modify the outcome of a transaction, for instance, in a decentralized exchange trade.

Front-running Attacks

Example


Front running in a Quiz Game
/quillaudits
Alice
1. Alice Submits correct answer with less gas fees (15 gwei)
Mempool
2. Bob don't know answer. He monitors Mempool for the txn and he finds the answer.
3. Bob Submits correct answer with higher gas fees (50 gwei)
Bob
Transaction Order
1 Bob txn (50 Gwei)
2 Alice txn (15 Gwei)
4. Bob txn executes before Alice due to higher gas fees
5. Txn submitted to chain
Blockchain

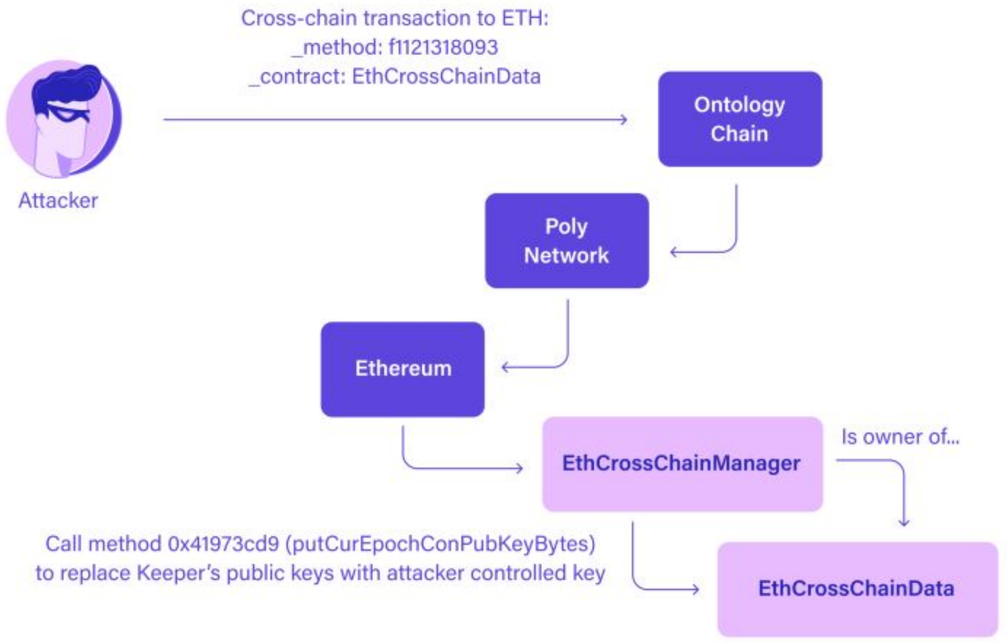
Memorable blockchain hacks: Poly Network

PolyNetwork

- August 2021
- $611 millions

EthCrossChainManager: privileged contract that has the right to trigger messages from another chain. All cross-chain transactions are executed through it.

It has a function named *verifyHeaderAndExecuteTx* that anyone can call to execute a cross-chain transaction.


Cross-chain transaction to ETH:
_method: f1121318093
_contract: EthCrossChainData
Attacker
Ontology Chain
Poly Network
Ethereum
EthCrossChainManager
Is owner of...
Call method 0x41973cd9 (putCurEpochConPubKeyBytes)
to replace Keeper's public keys with attacker controlled key
EthCrossChainData

Memorable blockchain hacks: BNB Bridge Hack

- October 2022
- $586 millions

BSC Token Hub: BNB bridge between the old Binance Beacon Chain and BSC, now BNB Chain

- BSC Token Hub was used to move assets between different blockchain networks
- There was a security flaw in the way the bridge verified whether a transaction was legitimate or no
- Hackers tricked the bridge into thinking legitimate transactions were going on. They obtained 2 x 1 million BNB tokens
- To avoid assets freeze the hackers moved as much coins as possible to other blockchains. They could escape with 127 millions
- Binance stopped the chain 8 hours to freeze the attack and investigate the incident

Formal verification

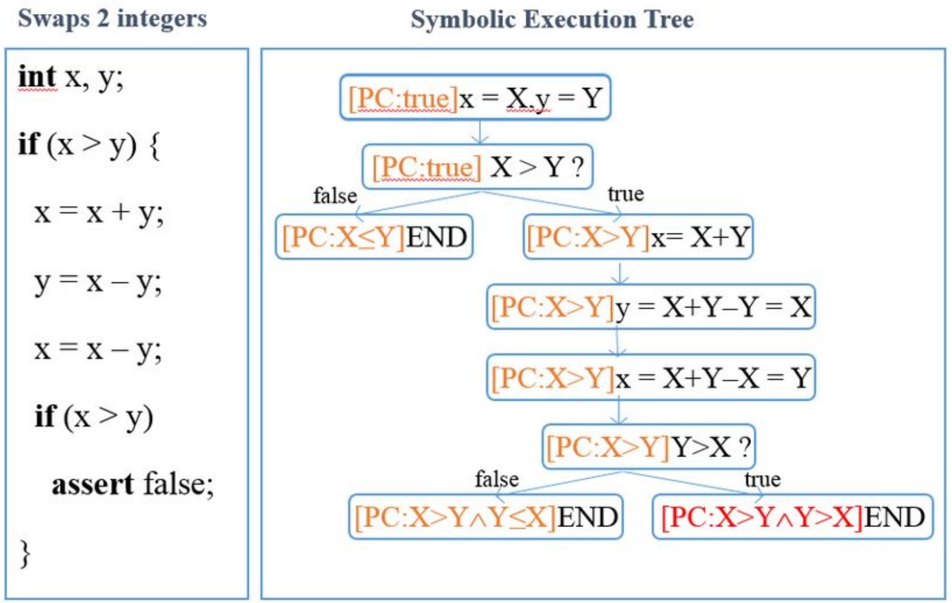
- mathematical proof of the properties (formal representation of business logic)
- to ensure correctness and prevent potential financial losses

1. translate code into formal representation
2. compute state space
3. specify desired behavior
4. detect and fix bugs

Tools for enhancing security

1. Static analysis tools
   - manually / automatic tools
   - syntax, standards, potential vulnerabilities
   - Slither (AST - formal representation), Oyente (symbolic execution directly with EVM bytecode)

2. Dynamic analysis tools
   - runtime errors, performance evaluations, unexpected behavior
   - Mythril (monitoring execution on different inputs)


Swaps 2 integers

```
int x, y;
if (x > y) {
  x = x + y;
  y = x – y;
  x = x – y;
  if (x > y)
    assert false;
}
```

Thank you for your attention!

Any questions?