Rahul's Checklist

Summary

Author: Rahul

Source: A checklist for smart contract security

Checklist

Reentrancy

A reentrancy attack in a solidity smart contract can completely drain your smart contract of funds. When an external call is made to another untrusted contract a reentrancy attack occurs. The untrusted contract makes a recursive call back to the original function in an attempt to drain funds which can cause the called contract to use the intermediate state of the calling contract. This situation is not always known during development. A simple example is when a contract does internal accounting with a balance variable and exposes a withdrawal function. If the vulnerable contract transfers funds before it sets the balance to zero an attacker can recursively call the withdraw function repeatedly and drain the whole contract.

Using the blockhash function

Blockhash function is similar to the timestamp dependency and it is not recommended to use it for the same reason as with the timestamp dependency because miners can manipulate these functions and change the withdrawal of funds in their analysis. This function is especially noticeable when block hash is used as a source of randomness.

Incorrect work with ERC20 token

Small discrepancies between the newly created token and actual ERC20 standard may lead to the non-functional method of the contract, and it will not be able to recognize the interface and thus leading to stuck funds and blocked contracts. There is a well-known OpenZeppelin implementation of the ERC-20 token, overused in modern protocols.

Frontrunning

Frontrunning can be defined as overtaking an unconfirmed transaction and it is one of the hardest issues to prevent. It is a result of the blockchain's transparency property. It is not easy to protect against frontrunning, and the issues detected in order to be fixed often requires some significant refactoring or redesign, All unconfirmed transactions are visible in the mempool before they are included in a block by a miner, and the interested parties can simply monitor transactions for their content and overtake them by paying higher transaction fees. This can be automated easily and has become quite common in decentralized finance applications.

Race conditions

This occurs when the nature of the system inhibits simultaneous operations when the operations should occur in a proper sequence. Smart contracts are executed only if certain conditions are met, there is a sequence to be followed. If external contracts are called in, it could inadvertently take over the control flow and make changes to the data. It was such a bug that caused the DAO's collapse. There are different types of race conditions. *Reentrancy* *race condition *occurs when functions are called repeatedly before the first invocation of the function was finished. *Cross function race conditions *can occur when two different functions share the same state in a contract.

Transaction Ordering Dependency (TOD)

Before any transaction is added in a block by a miner, it is kept in the mempool for a short while. This makes it possible to tell what actions occur before its added to the block. But when it comes to decentralized markets, it goes a bit off; as the market order implemented is seen before the other transaction gets included in the block.

Timestamp Dependence

Contracts could have timestamp conditions to perform certain actions. The problem is when the miners manipulate them. The Blockchain accounts for the local system's timestamp which causes a delay of seconds which is sufficient for hostile parties to launch an attack on the contract. This could affect the outcome of timestamp dependent contracts as the miner could then choose a different timestamp.

Reorder attack

When a miner competes with a smartcontract participant to place their own information stored on the contract. When a participant places his bet, corresponding data is saved on the blockchain and any miner can access it to see the bet number simply by calling a public mapping. Since the condition of the called function isn't updating until the end of mining, there is a risk of a reordering attack.

Overflow and underflow

This kind of vulnerability is quite hard to happen. Anyhow, there is a possibility for it to happen. In this type of vulnerability, if the balance token reaches the maximum or minimum unit value, it will circle back to zero or the maximum set limit. If your token's unit value is favorable for a user to reach the maximum limit of (2^256), then this vulnerability is a concern for you. But for most cases, it is not the case. Also, the ability of a user to update the unit value by calling a function makes the contract vulnerable to attack.

Short address attack

This vulnerability of ERC20 tokens was discovered by Golem team. A hacker can create an ethereum wallet address with trailing zeros and buy token from a contract by removing the trailing zeros. If the contract doesn't check the length of the address of the sender, Ethereum's virtual machine will just add zeroes to the transaction until the address is complete. This results in returning 256000 tokens for each purchase of 1000 tokens.

DoS with unexpected revert

An example would be an auction contract. When a refund is attempted to the previous high bidder and it fails, then it reverts. This opens up a vulnerability when malicious actors could place themselves as the highest bidder by ensuring that all refunds to their address fail and this would prevent anyone else from bidding. It could also be the case with crowdfunding contracts which requires it to iterate through an array to pay users. If one payment fails, then the entire payout would revert.

DoS with block gas limit

A related problem as the previous point is if you are paying out to a lot of addresses at the same time, then you risk running out the block gas limit which is the maximum amount of computation that can be processed by a block. If that happens, it results in a failed transaction. An attacker could take advantage of this by adding a number of addresses that are due to very small payments. Thing is, the gas cost of these payments end up being more than the limit itself and that would block the refunds altogether.

Sending ether to a contract

The logic in the contract is designed to disallow payments to the contract unless it is forcibly done. There are quite a few methods to send Ether without activating the fall-back function. Thus, it is possible to make the balance of the contract greater than zero and possibly affect its function if is programmed with a condition which checks the balance in the contract.

Simple logic bugs

Some of the above issues are more specific to smart contracts, others are common to all types of programming. By far the most common type of issue we detect consists of simple mistakes in the logic of the smart contract. These errors may be the result of a simple typo, a misunderstanding of the specification, or a larger programming mistake. These tend to have severe implications on the security and functionality of the smart contract.What they all have in common though, is the fact that they can only be detected if the auditor understands the code base completely and has an insight into the project's intended functionality and the contract's specification. It is these types of issues that are the reason smart contract audits take time, are not cheap, and require highly experienced auditors.