Solidity/Ethereum vulnerabilities

Ákos Hajdu

This is a supplementary material for the Blockchain Technologies and Applications (VIMIAV17) course at the Budapest University of Technology and Economics.

Introduction

There is a wide variety of vulnerabilities in blockchain-based infrastructures. The source of vulnerabilities is often the misalignment or the gap between the programmers intent and the actual execution semantics. Vulnerabilities can be categorized by the layer in which they appear. Here we discuss vulnerabilities in the context of Ethereum and Solidity, but most of them can appear in other blockchain infrastructures as well.

Programming language / contracts

- **Call to the unknown**: Some primitives in Solidity (to call functions and to transfer funds) invoke other functions (e.g., the fallback function) as a side effect. The invoked function might belong to a malicious contract. The malicious contract can perform some operations or even call back into the caller, which can be in an inconsistent state (see *reentrency* vulnerability). The programmer should be aware of such side effects.
- **Gasless send**: Sending funds might cause an exception due to insufficient execution fees if the called function performs computations as a side effect. For example, `transfer` and `send` executes the fallback function with a limited amount of gas by default, which is only enough to perform logging. A computation writing the blockchain state would run out of gas.
- **Mishandled exceptions**: There are several different ways of throwing exceptions in Solidity and handling them is not uniform. Therefore, the programmer might miss certain kinds of exceptions. For example, `transfer` propagates the exception up to the caller, but `send` and `call` indicates it in its return value. Analysis tools (or sometimes even the compiler) can warn when the return value of such functions is ignored, but the programmer should also pay attention.
- **Type casts**: While the compiler checks certain type errors, once deployed we cannot be sure about a type of a contract behind some address. If its type does not match but has a function with the same signature, it will still be called. For example, if we receive the address of a contract of type `C` as a parameter, there might actually be a malicious contract with type `M` behind the address, which has the same signature (i.e., same public API).
- **Reentrancy**: Programmers might have the intention of atomicity in transactions, but certain primitives in Solidity (e.g., `call`) pass the control over to the callee, allowing them to recursively call back into a contract. The caller might be in an inconsistent state when the recursive call is made. This caused the infamous DAO attack.
- **Keeping secrets**: To set a private state variable, a user has to send a transaction to call some setter function. The transaction and the function codes are public, so the value of the private variable can be inferred. Never store passwords or other sensitive information on the blockchain, not even in private variables.
- **Unchecked caller**: Anyone or any other contract can call public functions. Programmers tend to forget to check that some operations should only be called by the owner (e.g., killing the contract).

- **Input validation**: Anyone or any other contract can call public functions with possibly malicious input (e.g., invalid input causing an overflow or an address with a malicious contract behind).

Execution engine

- **Under/overflows**: The Ethereum Virtual Machine (EVM) can operate with 8, 16, 24, 32, ..., 256 bit signed or unsigned integers, which silently under/overflow without triggering an error. For example, `255 + 1 == 0` on 8 unsigned bits and `127 + 1 == -128` on 8 signed bits. Programmers should consider using the SafeMath library for arithmetic operations. An infamous overflow caused tokens to appear from nowhere.
- **Immutable bugs**: An inherent property of the blockchain is that the code of a contract cannot be modified or patched after publication, even if a bug is discovered: the contract will stay there to be exploited. Note, that there are some patterns to kill a contract or to forward calls to a mutable address, but that again brings up new vulnerabilities (an attacker might kill or hijack the contract). Such patterns should be implemented with great care.
- **Ether lost in transfer**: Sending Ether to an orphan address (e.g., the user lost the private key) is lost forever.
- **Stack size limit**: Until a hard fork, it was possible to set up a long chain of calls (one less than the limit) and then finally call a victim contract. If the victim called some other function, it would fail with an exception due to the stack size limit. The victim contract might not have been expecting an exception.

Blockchain and cross-peer protocols

- **Unpredictable state/transaction ordering dependency**: The state of a contract is determined by its data and its balance. Between issuing a transaction and its actual execution, other transactions might change the state of the contract. The order in which transactions are collected into block depends on the miners. For example, one might issue a transaction to buy some tokens for some price. A quick attacker could see this transaction and increase the price. Miners might include the price increasing transaction first.
- **Generating randomness**: The blockchain is inherently deterministic. Timestamps and block hashes might be used as seeds for pseudo-random generators. However, miners can influence these parameters and could bias the outcome.
- **Time constraints/timestamp dependency**: Some applications use the block timestamp to determine which actions are permitted (e.g., if the owner does not claim the funds, some other user might claim it after a given time). Miners can control the timestamps to a small extent.

References and further reading

The vulnerabilities were collected and categorized based on the following papers. You can refer to them for more details and examples.

- Atzei, Bartoletti, Cimoli - A survey of attacks on Ethereum smart contracts (2017)
- Luu, Chu, Olickel, Saxena, Hobor - Making Smart Contracts Smarter (2016)
- Nikolic, Kolluri, Sergey, Saxena, Hobor - Finding The Greedy, Prodigal, and Suicidal Contracts at Scale (2018)

It is also highly recommended to read about security considerations, common attacks and best practices. There is a handful of tools targeting the verification of contracts, including Truffle, Securify, Oyente, Maian, MythX, Slither, solc-verify and VeriSolid.