Software Security Analysis of Ethereum Smart Contracts

Nicolas Schapeler
TUM Chair of IT Security

Abstract

1 Motivation

In recent years blockchain technology has seen more and more widespread adoption, ranging far from its initial use case of a decentralized means to exchange electronic cash, as introduced by Bitcoin.

A key innovation in this space has been the introduction of so-called smart contracts, which were first implemented by the Ethereum protocol based on the ideas of Computer Scientist Nick Szabo. These are programs that execute automatically on the Ethereum network if certain conditions are met or if they are invoked by a blockchain participant. Through these, one can create more complex applications while still retaining the built-in trustlessness and security for which blockchain technology was initially designed. Today, smart contracts are used to automate various processes, ranging from supply chain management over energy distribution in smart grids to health care.

The use case that provides the main motivation for this paper, however, is Finance. Here smart contracts are used to automate and enforce financial agreements between different parties, such as loan provisions, asset exchanges, and asset management amongst others. At the time of this writing, four of the top smart contracts on the Ethereum network manage around 60 Billion USD, with hundreds of other contracts overseeing multiple millions in value. Considering this in addition to the relative novelty of the field to both developers and law enforcement along with the built-in anonymity of blockchain technology applications, one can deduce why this field presents such an attractive target to malicious actors - At its current stage, performing an attack on a smart contract application often yields extremely high rewards in comparison to the risk of facing consequences.

For these reasons, smart contracts have been the target of multiple high-profile attacks in the past years. The first large-scale attack was the DAO-Attack in 2016, where 3.6 Million Ether, worth around 50 Million USD at the time or 10 Billion USD today, was stolen and lead to the developers of Ethereum taking the controversial action of manually reverting the attacker's exploit transactions. Other prominent attacks include the King of the Ether Throne attack in 2016 and the Parity Multisig Wallet Attack in 2017 (more details regarding the exact exploits used in these attacks are given in the Background). The reason we mention these specific attacks is that all of them exploited vulnerabilities that in hindsight appear relatively simple. The key significance, however, is that the vulnerabilities utilized are all smart contract-specific, and may therefore not be intuitively detected by a developer with lesser experience developing such applications. As previously mentioned, due to the relative novelty of smart contracts, few developers are highly experienced and are therefore prone to fall victim to already-known vulnerabilities. For this reason, we believe the software security analysis methodologies and tools disclosed in this paper are of key importance in a space where each security flaw can lead to significant financial loss: They allow for the detection of vulnerabilities the developer may not be aware of or may have overlooked.

This Systemization of Knowledge begins by giving a brief overview of the background concepts regarding software security analysis and Ethereum smart contracts, followed by a systemization of different approaches to security fault detections, where we present and experimentally test claims regarding accuracy and performance of various existing tools.

2 Background

In this section, we provide the theoretical background information related to software security analysis and Ethereum necessary for understanding the rest of this paper.

2.1 Ethereum

Ethereum is an open software protocol that allows for the decentralized execution of smart contracts and the exchange of value in its native cryptocurrency, Ether [1]. These smart contracts run on the Ethereum Virtual Machine (EVM).

2.1.1 EVM

The Ethereum Virtual Machine is a distributed state machine with its own set of instructions and own state at any given time. This state records all relevant information related to the Ethereum network at any given time. As one might expect, the EVM's role is to define a state transition function using its instruction set which clearly defines what next state is reached given an input state and a certain instruction to be executed.

2.1.2 Accounts and Smart Contracts

In this context, a significant data structure for the Ethereum protocol is the account. This data structure has the ability to send and receive transactions, hold a balance of Ether, and interact with other smart contracts. We differentiate between two types of accounts, externally-owned accounts and smart contracts. Externally-owned accounts are generated using a public-private key-pair and can interact with the rest of the Ethereum Network using the private key's signature as proof an instruction stems from a specific account. These signed EVM instructions that accounts use to interact with one another are called transactions. Smart Contracts, on the other hand, are deployed as pieces of software and do not come with a private-public key pair. They define a set of callable functions that other network participants can invoke. As such, a key difference between smart contracts and externally-owned accounts is that smart contracts can only perform transactions if they were first invoked by a different account. As smart contracts can be called by any account (both externally owned and other smart contracts) on the Ethereum network, it is crucial that all network participants agree what a smart contract can and cannot do. To enforce this, the code of smart contracts is strictly immutable [2].

2.1.3 Gas Fees

As previously mentioned, the physical hardware the EVM runs on and transactions are executed on is provided by a subset of all network participants, called nodes. Since these do not want to give storage and computing power away for free, Ethereum implements a gas system, whereby each operation offered by the EVM is assigned a certain gas cost. This gas is just a fee in Ether paid for the computing effort and/or storage provided by the nodes. It is important to note that the maximum an account is willing to pay for these gas fees is set by the transaction initiator, not the processing nodes. The nodes, however, have the choice of processing transactions in whatever order they please and will usually choose transactions with the highest fees and lowest effort to themselves. This means a transaction can take a significant amount of time to be processed or be rejected by the nodes altogether if the nodes believe the fees are not high enough for their expended effort. It is also to be noted, that blocks in the Ethereum Network have an upper gas limit, a limit on how much computation can be done and how much gas can be spent in one block, which can also lead to a transaction being reverted if the work performed in it costs more gas than the upper gas limit.

[1] Technically the unit smart contracts and accounts exchange value in is Wei, however we use Ether in the context of this paper as this makes the numbers far more manageable. (1 Ether = 10^{18} Wei)

[2] To be clear, only the code of smart contracts is immutable. Smart Contracts can still contain various data structures such as variables, mappings, arrays etc. which are mutable.

2.2 Solidity

Although the smart contract representation one finds on the physical blockchain is in EVM bytecode, very few smart contracts are manually written in this form and instead employ higher-level languages. There are multiple programming languages specifically adapted for developing smart contracts, however, in the context of this paper, we set our focus on Solidity. The reason we choose this language is not only for it being the most popular [citation] but also due to it having received criticism for being designed in a way that facilitates errors that lead to security flaws in smart contract development. We introduce some of these patterns here in order to better help the reader's understanding of the types of issues the software security analysis tools detect in the Contribution section of this paper.

- Inconsistent Exception handling

  Exceptions can be caused by various factors, such as a transaction going over its allocated gas fee limit, the EVM call stack overflowing, or the `throw` command being called. However, when an exception is raised, Solidity is not consistent in how it handles these and may therefore facilitate creating vulnerabilities caused by incorrectly handling exceptions. For example, if one wants to transfer Ether from an account to another, this can be done in 3 ways:

  1. `<receivingAddress>.transfer(amount)`
  2. `<receivingAddress>.send(amount)`
  3. `<receivingAddress>.call().gas().value(amount)`

  These have some key differences: The `transfer` method throws an exception if the transfer fails for any reason and reverts the entire transaction it was a part of. The other two, however, simply return `False` on failure and do not revert the transaction. Additionally, `transfer` and `send` have a fixed gas fee when used, while `call` can have its gas fee set by the invoker.

- Fallback function

Smart contracts have the option of defining a so-called fallback function which is executed when receiving Ether without the sender specifying a function call[3]. Aside from being executed as a side effect to receiving Ether, this function is also executed in other situations that may seem unexpected. Given a contract *C* with a function with the signature `function f(uint) returns (byte)`, if one were to call the function `function f(int) returns (byte)` (notice the incorrect signature) on *C*, the result would not be a revoked transaction, exception, or a returned `False` boolean, but rather an execution of the fallback function of *C*. The same is true if one mistypes the address of *C* and attempts to invoke the nonexistent function `f` on a contract C', the result is an execution of the fallback function of C'.

This behavior worked in conjunction with the exception handling inconsistencies to cause the King of the Ether Throne attack mentioned in the introduction. There, the essence of the attack was invoked through the usage of the previously introduced `send` function in the vulnerable contract. The intention of `send` in the victim contract was to simply send an amount of Ether to an address, however the programmer had forgotten about the side-effect of the invoked fallback function if the recipient is a contract. Next, the `send` function's fixed gas fee was not enough to pay for the attacker's contract's fallback function fee and an exception was produced, which `send` returns through the use of a boolean. The victim contract, however, had no boolean check for `send`, which resulted in the victim contract finding itself in a deadlocked state.

- Reentrancy

  This vulnerability is not as much an influence of Solidity's design like the previous two examples, however we view it as one of the key attack vectors in smart contract development and therefore provide a short explanation. Assume we have two Smart Contracts:

```
contract Bank{
  mapping (address => uint) balances;

  function withdraw(uint _val) public {
    // msg.sender is the account that
    // made the function call
    address a = msg.sender;
    // Check sufficient balance
    if(balances[a] >= _val) {
      // Send the money
      // and check the send succeeded
      require(a.call.value(_val).gas());
      // Remove sent money from balance
        balances[msg.sender] -= _val;
        }
    }
}

contract Attacker{
  // Fallback function,
  // executed when receiving money
  function() payable {
    // Call Bank.withdraw again
  }
}
```

  As one can deduce from the provided code, the vulnerability here is that the bank first sends money to the withdrawing account before removing the sent money from the account's balance. Additionally, the bank uses the `call` function to send money and does not specify a maximum gas fee. An attacker can therefore implement a more complex fallback function without the risk of their money transfer failing due to the gas fee of their fallback function being too high. The attacker leverages this by calling the withdraw function again, thereby being able to withdraw as much money as he wants from the bank contract until the bank is out of money. This sort of vulnerability is what was leveraged in the DAO Attack mentioned in the introduction.

- Denial of Service using Gas Limit The final attack vector we seek to present is a DoS attack. Assume we have given the following contract:

```
contract DoS{
  address[] ads;

  function addAdr(address toAdd) {
    ads.push(toAdd);
  }

  function f(){
    for(uint i = 0; i < ads.length; i++) {
        // Some operation on each address
    }
  }
}
```

  An attacker can provoke a DoS by adding suficiently large number of addresses to the `ads` array of the contract. Then, if another account seeks to execute `f`, the gas cost will be so high that it surpasses the block gas limit and the execution will fail.

[3]If a contract does not define such a function, it cannot receive Ether.

2.3 General Security Analysis Methodologies

Finally, we introduce some general security analysis methodologies which adaptions to Ethereum will be presented in Section 3 of this paper.

2.3.1 Symbolic Execution

Symbolic Execution functions by taking a program's source code as an input and assigning symbolic values to unknown inputs. It then executes the given program using said symbolic values and is thereby able to compute expressions for the unknown inputs along with fitting constraints belonging to each possible execution path of a program. A key limitations of symbolic execution is path explosion, where the number of possible paths to symbolically execute becomes very large very quickly for large programs, especially if these contain loops that cannot be trivially rolled out.

2.3.2 Mutation Testing

The goal of mutation testing is to assess the efficiency of the test suite belonging to a certain piece of software. It works by generating so-called "mutants" of the provided source code, which are clones of the original code with slight alterations (for example, a change from a $>$ to a $\geq$). Then, the given test suite is ran on these generated mutants and the amount of mutants which pass the tests is measured. Generally, a low amount of mutants passing all tests is viewed as evidence of the test suite's high quality.