Smart Contract Vulnerabilities. Making profit and possible solutions

Group Number: 10
Team Members:

Alejandro Ly Liu	T06902105
Paul Vong	A06922104
Christopher Foead	T06705106
Vitor Esteves	A06922110

1. Introduction

With cryptocurrencies getting widespread and popular, we decided to dig into associated problems because, the more people make use of those, the more people are going to start trying to exploit it in order to make profit. This way, this report will address the Ethereum blockchain flaws and a specific honeypot strategy that can be done by the use of smart contracts.

Honeypots are originally a security measure used in computer science in order to record hacking attempts by creating an attractive "bait" for hackers to attack. In this case, these honeypots are instead used to lure innocent users into investing their money on malicious smart contracts. Normal smart contracts behave like investments, allowing users to gain some profit from the money they put in whereas malicious smart contracts scam the user by allowing the owner to keep all of the money invested without paying out to the investor. People who are new to cryptocurrencies will most likely be unable to identify what is a legitimate smart contract and what is a honeypot. Even those familiar with cryptocurrencies may find it difficult considering that potential users will need to understand Ethereum's particular coding language to determine which contracts are legitimate and which ones are not.

This report will highlight two methods in which people are able to create malicious honeypots and the key way to identify such issues. Code analysis will be done in order to explain how these exploits work. Additional information will be given on how Solidity, the programming language that handles Ethereum's smart contracts, handles some interactions to provide more context in order to show how these exploits can occur.

This is done in hope that readers can identify and gain a better understanding on how smart contracts work. Not only they will be able to protect themselves from malicious smart contracts, but they will also be able to understand the mechanisms of Ethereum's contracts and identify similar malicious smart contracts.
Furthermore, there are a few issues with Solidity itself which will be further explored, as they are what allow these malicious smart contracts to appear in the first place.

2. Smart Contracts. Tricking the user

Here we explain in more detail different malicious smart contracts taking both advantages of different vulnerabilities to take out all the ethers inside an smart contract.
At the same time, we also consider important to explain the attack to Parity Wallet.

MerdeToken [1]

MerdeToken (MDT) is our new coin offering. You can tell we're legit because we've selected a trusted third party to enforce withdrawal limits to protect your invested funds. Please inspect our contract carefully, verify that the contract instance has the correct address of the trusted third party, and then invest!

- By sending ETH to the deposit method, you will receive an equivalent amount of MDT.
- The contract implements a minimal ERC-20 sub-set so you can transfer your tokens.
- The contract uses onlyOwner and onlyTrustedThirdParty modifiers. Only the trusted third party can change the withdrawal limit, and only the owner can withdraw ETH (up to the withdrawal limit).
- We have some bonus code functionality that will be completed in a later contract. You can ignore that for now.

The bug and how it works:

Note: As far as we know, this submission and its exploit only rely on expected Solidity behaviour, and don't take advantage of any compiler or EVM bugs.

The bug is in the following line in popBonusCode():

require(bonusCodes.length >= 0);

This condition is always true since array lengths are unsigned. The code was trying to check that the array was not empty so it s**hould have used > instead of >=.** This is a common off-by-one bug that could plausibly be dismissed as coder error. Furthermore, the bug is in a part of the contract completely unrelated to funding.
Buggy or not, it doesn't appear as though any of the bonus code methods could influence the funding mechanisms at all. I believe that is the significance of this submission: mistakes in dynamic array handling can result in total contract compromise in non-obvious ways.

Exploitation

The contract owner can attempt to underflow an array size by executing the following code when the length of the bonusCodes array is 0:

```
bonusCodes.length--; // No pop() method?
```

The length attribute seems to be treated as a normal variable since this operation causes the array length to wrap up to the maximum uint value, 2^256 - 1. At this point, bounds checking on the array has been effectively bypassed since all indices will be valid (except 2^256 - 1). Since the owner can call modifyBonusCode to alter any element in the array, which now encompasses nearly all of the storage address space thanks to wrapping, arbitrary writes to any location in storage are possible, like changing the value of the withdrawLimit. This is somewhat analogous to C-style pointer manipulation bugs.

Roundtable [2]

This Smart Contract is a DAO-like structure, which has an ICO where "seats" around a table are auctioned. The contract creator (owns seat 0) and has already pitched in 100 ether to initialize the roundtable and apparently gets a little honorarium for each bid (100 wei).
The problem with this smart contract is that an exploit have been set at line 121, which the creator can use to set his honorarium to basically infinite.
The details for voting and things like that have been left out, let's assume all such logic is implemented in the war_chest, which grants each holder of the 15 seats an equal share of the assets.

Technical explanation

The exploit concerns the length of ABI-encoded arrays. The entire model of the auction is modeled around this quirk of ABI-encoding and how solidity handle arrays.

- First, a large value is loaded into memory and make *claimHonorarium* to take a val parameter.
- To hide the strange validation of the array sizes, the validation of msg.length is used for the protection against short-address attacks, although in this case is not needed.
- The call needs a dynamic parameter, but the exploit requires a static size. Then to mask that an interface with the call signature is defined.

The exploit

The exploit is undetected by many solidity-coders, but it cannot be exploited using normal solidity-dev tools, since it requires a custom ABI-encoded payload. Is important to notice that since the creator knows this backdoor exists, he can always outbid the other bidders, fully aware that he can get all money back. And if he is successful doing so, he won't have to actually use the exploit.

Analysis of the attack to Parity Wallet [3]

In this attack, a hacker took control of a library contract inside parity wallet for later killing it, stopping the functionality for 500 multi-signature wallets and freezing $150M.

A hard fork was required to restore the contract and/or return funds. The attack was made by making use of the function initWallet which claimed the contract, this was possible because Parity did not set the only_uninitialized variable. At this point, the owner of the library contract,can call any privileged function, amongst which kill(). The kill function calls suicide(),which is now replaced by self-destruct.

The suicide function sends the remaining funds to the owner, destroying the contract and clearing its storage and code. This multi-signature wallets were now locked and the majority of the functionalities depending on the library returned zero for every function call.

Best practices [3]

Smart Contracts security is an open research field. The development of bug-free source code is still an utopia for traditional software development, after decades of analysis and development of engineering approaches. Error freedom is even more daunting for blockchain software development, which started less than a decade ago. It is worth remarking here that vulnerabilities like the one leading to the Parity attack had been highlighted in literature, a fact that strengthens even more our call for the adoption of standard and best practices in Blockchain Software Engineering.

A. Anti-patterns

An anti-pattern is a common response to a recurring problem that is usually ineffective and risks being highly counterproductive.

For example, faults could be so corrected in SC code and the corrected version of the same SC can be successively redeployed and accepted by miners. Once provided with the address of the debugged contract, the very same contracts that were calling the faulty version can call the debugged contract.

B. Testing

Testing smart contract is challenging and critical, because once deployed on the blockchain they become immutable, not allowing for further testing or upgrading. At present, to the best of our knowledge, there is not a testing framework for Solidity. The nature of smart contracts introduces at least two complications to testing: an application may be critical and it is very difficult to update an application once deployed. As a result, it is desirable to use robust testing techniques. Manual test generation is likely to form an important component but inevitably is limited; there is a need for effective automated test generation (and execution) techniques.

Currently available options to test contracts are:

- Deploy the contract to live Ethereum main network and execute it. This costs about 5 minutes and real money to deploy and execute.
- Deploy the contract to the test-net Ethereum network and execute it. This costs about 2 minutes to deploy and free ether.
- Deploy the contract to an Ethereum network simulator and execute it. This costs about 3 seconds and is free, but limited interaction and no realistic test of network related issues.

Possible solutions are symbolic execution, code coverage, model-based testing.

3. Possible Solutions

Possible solutions for this problems would be:

- Standardize the use of data-structures, so low-level array length manipulation is unnecessary.
- The implementation of an iterable map supporting delete operations.
- The use of a standardize developer tools, like solidity-dev tools.
- Bias every smart contract by ourselves.
- Improve EVM by adding tests.

4. Conclusions

In summary, we just want to denote the importance of a Blockchain software cycle and the use of testing frameworks, so we can mitigate code flaws or errors in the deployment of smart contracts.
Many of these testing activities are currently being done by developers doing their own testing, but as blockchain gains popularity, testers will need to start getting involved as well. There is a need to check and recheck code, and then audit it again. This will involve designing and implementing a code review process, while paying careful attention to the functionalities provided in library contracts.

This way, we hope that this report can raise some awareness about the security flaws of the blockchain, while giving in an explanation about past occurrences, because understanding past hacks can also help shine a light on some of the security vulnerabilities that exist already and allow to predict newer ones.

5. Division of work

All of the group members contributed equally to the final product without having to distinguish between people individually, since we used share document technologies such as google docs.

6. References

[1] USCC Submissions 2017 - Doughoyte
https://github.com/Arachnid/uscc/tree/master/submissions-2017/doughoyte
[2] USCC Submissions 2017 - martinswende
https://github.com/Arachnid/uscc/tree/master/submissions-2017/martinswende
[3] Smart Contracts Vulnerabilities: A Call for Blockchain Software Engineering?
https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8327567