

# Exploiting smart contracts

By

Zhuohao Liang  
(170243845)

August 2018

Supervisor: Dr Changyu Dong

MSc. Computing Science  
Newcastle University

[Word count: 14512]

# **Declaration**

This dissertation is submitted in part of fulfillment of the requirements for the degree of MSc Computing Science in the Newcastle University.

I declare that this dissertation represents my own work except where otherwise stated.

Zhuohao Liang

August

2018

## **Abstract**

With the fast advancement of technology, companies as well as people are investing and utilising smart contracts in blockchain and Ethereum platforms. However, the threat of online issues are expanding fast and becoming vast, which could be costly. This project focused on understanding these threats and vulnerabilities based on the mentioned platforms that affect smart contracts, and suggest ways to prevent them. Some of the vulnerabilities identified included transaction ordering dependence, attacks on the contract, timestamp dependence, botched exceptions, reentrancy vulnerability, greedy contracts, suicidal contracts, posthumous contracts, and others. These vulnerabilities level can also be classified as low, medium, and high, depending on how they affect a secure system. Some of the mitigations include making smart security checks to a contract before executing it. It is essential for Solidity to integrated overflow-protected arithmetic types that can assist systems to detect and curb with integer related vulnerabilities.

## **Acknowledgments**

I would like to thank my project supervisor Dr. Changyu Dong for her valuable guidance, help and support during the course of this project.

# Table of contents

|  |    |
|--|----|
| 1 Introduction.....                    | 2  |
| 1.1 Problem statement.....             | 4  |
| 1.2 The reason for this choice.....    | 4  |
| 1.3 Objectives.....                    | 4  |
| 1.4 Research questions.....            | 5  |
| 1.5 Outline.....                       | 5  |
| 2 Background.....                      | 6  |
| 2.1 Blockchain.....                    | 6  |
| 2.2 Ethereum.....                      | 7  |
| 2.3 smart contract.....                | 8  |
| 2.4 Solidity.....                      | 9  |
| 2.5 Remix-ide.....                     | 10 |
| 2.6 EVM.....                           | 10 |
| 2.7 cryptocurrency wallet.....         | 10 |
| 2.8 NPM.....                           | 10 |
| 2.9 Nodejs.....                        | 11 |
| 3 Integer overflow.....                | 13 |
| 3.1.1 Code demo.....                   | 14 |
| 3.1.2 Official Fix.....                | 16 |
| 3.1.3 Vulnerability demo.....          | 16 |
| 3.2 Proxy Overflow.....                | 18 |
| 3.2.1 Introduction of ERC20.....       | 18 |
| 3.2.2 Real World Case.....             | 18 |
| 3.2.3 Normal procedure.....            | 19 |
| 3.2.4 Demo.....                        | 20 |
| 3.2.4Fix.....                          | 21 |
| 3.2.5 Malicious Use in Real World..... | 21 |
| 3.3 BurnOverflow .....                 | 22 |
| 3.3.1 Real World Case.....             | 22 |

|   |    |
|---|----|
| 3.3.2 Demo.....                                       | 23 |
| 3.3.3 Fix.....  | 25 |
| 3.4 Summary.....                                      | 26 |
| 4 Access Permission and Custom Modifier.....          | 28 |
| 4.1 ceoAnyone.....                                    | 30 |
| 4.1.1 Introduction.....                               | 30 |
| 4.1.2 Code demo.....                                  | 30 |
| 4.1.3 Real World Case.....                            | 32 |
| 4.1.4 Attack demo.....                                | 33 |
| 4.1.5 Fix.....  | 36 |
| 4.2 OwnerAnyone.....                                  | 36 |
| 4.2.1 Real World Case.....                            | 36 |
| 4.2.2 Attack demo.....                                | 37 |
| 4.2.3 Fix.....  | 39 |
| 4.2.4 Summary.....                                    | 40 |
| 5 Trade Trap.....                                     | 41 |
| 5.1 Transaction ordering dependence.....              | 41 |
| 5.1.1 Attacks on the contract.....                    | 41 |
| 5.2 Arbitrary Increase in Token Balance.....          | 44 |
| 5.2.1 Real world case.....                            | 44 |
| 5.3 Manipulatable Prices and Unfair Arbitrage.....    | 45 |
| 5.3.1 Real World Case.....                            | 45 |
| 5.3.2 Demo.....                                       | 45 |
| 5.3.3 Try to hide the Vulnerability.....              | 47 |
| 5.4 Other contracts with trace vulnerabilities.....   | 48 |
| 5.4.1 Suicidal contract.....                          | 48 |
| 5.4.2 Greedy contracts.....                           | 48 |
| 5.4.3 Posthumous contracts.....                       | 49 |
| 5.5 Some vulnerabilities not mentioned in detail..... | 49 |
| 5.5.1 Timestamp dependence.....                       | 49 |
| 5.5.2 Race condition.....                             | 49 |

|  |    |
|--|----|
| 6 How was done.....                            | 51 |
| 6.1 Literature search.....                     | 51 |
| 6.2 Taxonomies.....                            | 51 |
| 6.3 Audited smart contracts.....               | 51 |
| 6.4 Justification of the selected methods..... | 51 |
| 7 Result and Evaluation.....                   | 53 |
| 8 Conclusion.....                              | 55 |
| 8.1 Limitations of the project.....            | 55 |
| 8.2 Future Work.....                           | 55 |
| 9 References.....                              | 56 |

## Table of figures

|  |    |
|--|----|
| Figure 1: Code demo of flow.....   | 15 |
| Figure 2: Result of flow.....  | 15 |
| Figure 3: Safemath library.....  | 16 |
| Figure 4: Modification on code.....  | 16 |
| Figure 5: Simple flow demo.....  | 17 |
| Figure 6: Endless loop by flow.....  | 17 |
| Figure 7: Overflow from transferProxy().....                                 | 18 |
| Figure 8: TransferAllowed modifier.....                                      | 19 |
| Figure 9: Test without overflow.....   | 20 |
| Figure 10: Test with overflow.....   | 20 |
| Figure 11: Balance of _from and msg.sender.....                              | 20 |
| Figure 12: Modification on transferProxy function.....                       | 21 |
| Figure 13: Implementation of standard transfer function.....                 | 22 |
| Figure 14: Detail of _transfer function.....                                 | 22 |
| Figure 15: Test with normal input data and success result.....               | 23 |
| Figure 16: Balance of “to” address after transaction.....                    | 23 |
| Figure 17: Test with normal revert situation.....                            | 24 |
| Figure 18: Test with overflow.....   | 24 |
| Figure 19: Modification on _transfer function.....                           | 25 |
| Figure 20: Test with safemath.....   | 25 |
| Figure 21: Basic model of onlyOwner.....                                     | 30 |
| Figure 22: Test with owner address.....                                      | 30 |
| Figure 23: Using non-owner address to test.....                              | 31 |
| Figure 24: A ceoAnyone-affected Crypto Idle Game Smart Contract.....         | 32 |
| Figure 25: Code of EtherCartel contract.....                                 | 33 |
| Figure 26: Function to set up contract.....                                  | 33 |
| Figure 27: A success transaction.....  | 33 |
| Figure 28: Balance at address “A”.....                                       | 34 |
| Figure 29: Modify ceoAddress with address “C”.....                           | 34 |
| Figure 30: Balance of address A and C.....                                   | 34 |
| Figure 31: Modification on code.....   | 35 |
| Figure 32: Part of code of AURA.....   | 36 |
| Figure 33: Part of code of Owned.....  | 36 |
| Figure 34: First time allocate balance by true owner.....                    | 37 |
| Figure 35: Attacker tamper the owner address.....                            | 37 |
| Figure 36: Attacker allocate balance.....                                    | 37 |
| Figure 37: Modification on setOwner.....                                     | 38 |
| Figure 38: Creation from owner.....  | 38 |
| Figure 39: Attacker try to tamper owner address.....                         | 38 |
| Figure 40: A contract that rewards user that solve a computation puzzle..... | 41 |

|  |    |
|--|----|
| Figure 41: MinToken function.....  | 43 |
| Figure 42: The manipulatable interfaces of affected smart contracts..... | 44 |
| Figure 43: Set up the contract.....                                      | 45 |
| Figure 44: Use B address to buy tokens then sell them out.....           | 45 |
| Figure 45: Hide setPrice code in setName function.....                   | 46 |
| Figure 46: Tools function to use.....                                    | 46 |



# 1 Introduction

The current rate of technology development has resulted into more applications on the internet through computers. The regular innovations being introduced into the world have had significant impacts in pushing the limits of software and systems today. One of these innovations is blockchain, which has proved to have a wide application, starting from financial systems, Internet of Things, public services, and internet interaction systems [1].

One application of block chain is in cryptocurrencies, which has resulted into a lot of attention from the academic, government, as well as financial fields.

However, with the development of such technologies and applications to make business and learning easier, there have been numerous setbacks as well, one of which is the security issues that arise. On the regular internet usage, the effects of security issues include lack of privacy, stolen identities, hacking and theft, data mining without client authorisation, and so much more that is enough to cause issues. However, this has further evolved with the increase in innovation. Platforms that host blockchains and its applications are now vulnerable to much more threats. Smart contracts form part of the blockchain innovation.

In order to understand the security issues facing smart contracts, it is critical to examine three key issues that are constant in this project, which are blockchains, Ethereum, and smart contracts.

A blockchain according to [2], is public ledger that is fully distributed through peer-to-peer platforms, which uses cryptography to make a secure connection to a platform, store data, transfer messages, as well as transfer digital currency. This system was invented in 2008 and introduced in 2009 under the name of Bitcoin. This became the pioneer of many innovations in cryptography.

In 2017, a second platform of cryptocurrency was introduced known as Ethereum. There has not been a definite definition of Ethereum, but from the inventor, Ethereum can be termed as a form of general use blockchain, which can be used in any programming language. This indicates that it could have wider applications such as the ones mentioned for use in blockchain and many more. This also means that a developer could build and install several applications all in one place, instead of building separate blockchains.

The third and most important part of the introduction is understanding smart contracts. These can be referred to as applications in blockchain and Ethereum platforms that are decentralised and support the competencies of these platforms[3], also explains smart contracts as user

defined programs that are running alongside blockchain and Ethereum, which are used to simulate trust between a user and the other party[4].

Smart contracts, can be defined as self-executing contracts (contracts without a third party), with the respective terms of the contract existing only between the buyer and the seller, and these are written directly into the lines of code[5]. The code, as well as the agreements that they contain, would exist across a distributed, as well as decentralized blockchain network[6]. What this implies is that smart contracts facilitate and permit trusted transactions, as well as agreements to be carried out among unrelated, anonymous parties, and these parties do not need a central authority, any external enforcement authority or a legal system. What the smart contracts do, is that they render these transactions transparent, traceable, and irreversible.

Essentially, smart contracts are computerized transaction protocols that can be used to execute the terms of contract. Fontein innovative technological advancements are introduced regularly with the aim of coming up with a novel, as well as better approach of implementing systems and software products in a more effective manner, and one of the most important such technologies is blockchain[7]. Blockchain, according to, has the capacity or is implementable in an array of use cases, for example, the internet interaction systems, the internet of things, public services and financial systems.

The popularity of smart contracts, especially in the past few years is one of the reasons why researchers have taken much interest in this phenomenon[8]. However, the challenges facing them are believed to be the more reason more and more studies have tried to critically examine how the contracts work, and what the vulnerabilities to these contracts are. Billions of US dollars are exchanged through the smart contract technology, and this is just an indication of how important this technology might be not just to the United States or Europe, but also to many other countries across the world.

Blockchain technology and most recently, cryptocurrency, have enjoyed a lot of attention from researchers as well as industry as earlier mentioned. Most notably in 2015, there was a release of what is referred to as the Frontier network of Ethereum, and ever since, there have been more and more cases where the execution of these smart contracts that manage Ether coins are characterized with conflicts or problems in general. These are just few of the challenges that characterize smart contracts, which necessitates the need to come up with strategies and methods that can help in addressing these challenges.

Different researchers have suggested different approaches or methodologies that could be used in detecting vulnerabilities in smart contracts. Fontein suggests the use of Mythril, which

provides an alternative of three input options; the first option is what is referred to as solidity code file. This option only works in situations where the solc command line compiler has been installed. The second option is what is referred to as solidity bytecode, and thus could be used in situations where the user does not have solc. The user, in this case, can compile the code using Remix before passing the runtime binary code to Mythril through what is referred to as a -c argument. The third and final option is the contract address, where the user can use the -a ADDRESS option to scan a particular contract's instance on the blockchain.

## **1.1 Problem statement**

With the fast advancement of technology, many companies as well as people are investing and utilising smart contracts in blockchain and Ethereum platforms. However, the threat of online issues are expanding fast and becoming vast, which could be costly. This project focuses on understanding these threats and vulnerabilities based on the mentioned platforms that affect smart contracts, and suggest ways to prevent them. This project will contribute new information to the field of technology, specifically in cryptography, which heavily relies on smart contracts, which as mentioned can be exploited.

## **1.2 The reason for this choice**

Blockchain is emerging Technologies and can cover many area in life. Not only trade or mine bitcoin or Eth online, but also can apply into real world to replace currency trading, without cumbersome procedure, just send token through account. But there are criminals try to find bug or vulnerability to get profit. So nowadays, lots of websites give some report to analyze smart contract vulnerability. After reading many of them, most can not give a detailed context to explain with real case but only a lot of space description. The most useful place to discuss vulnerability is probably forum or blog. So here try to collect data and do summary, then choose some classic type to analyze.

## **1.3 Objectives**

The following are the project objectives:

- To analyse security vulnerabilities in smart contract programming based on blockchain and Ethereum platforms

- To suggest solutions on how to prevent vulnerabilities in smart contracts that promote secure programming

## 1.4 Research questions

The following are the research questions:

- How can the security vulnerabilities in smart contracts be categorised?
- How can these vulnerabilities be mitigated for secure smart contracts?

## 1.5 Outline

The project will go through some stages that will be discussed in the next chapters as outlined below:

- Chapter 2 will be looking at existing related works to introduce concept about Ethereum
- Chapter 3 will present the most classic vulnerability type, integer flow, in smart contract, basic mechanism why flow will happen, recent case of flow and fix.
- Chapter 4 will present the next vulnerability, access permission and custom modifier, also the simulation of attack demo of case and fix.
- Chapter 5 will present a type of special vulnerability, trade trap, how do some malicious contract cheat user and some option to hide the trap.

## 2 Background

### 2.1 Blockchain

A blockchain is a ledger which is shared and it is used to keep track all the transactions that have taken place and all the data that has been reciprocated from the time it was created [9]. It basically keeps a continuous emerging list of catalogues known as blocks. Each particular block has distinctive transactions and it is linked to the block that was generated before, up to the block that was created first. Mining involves the process adding a block and certifying the rationality of the transactions (avoid double expenditure) via a proof of work or other relevant conceptions such as Proof of Stake . Basing on this model, blockchains are viewed to be solidly impermeable on changing the data that was preserved in the blockchain before. Moreover, blockchain is duplicated in multiple nodes within networks which decentralizes the technology of blockchains and enable users to send transactions in a way that is secure without requiring a third party. Elimination of third party interface results to lower costs of processing, disposal of single point of failure risk and increase in the certainty of transactions. Zheng further outlined the characteristics of blockchain and they include: Decentralization: This property eliminates third party control. Another feature is Persistency which states that Blockchains are associated with the inability of deleting and modifying transactions once they are modified[10]. The third feature is anonymity which is rooted to cryptographic hashing of blockchain data and asymmetric cryptography such as digital cryptographic keys and transactions. The fourth property is audibility which states that blockchains are related to global truth and there data is available to the public which makes it easy to certify and trace previous transactions. Blockchains have a close association to the actuality of cryptocurrencies basing on incentive initiation which launched the first reliable generation of cryptocurrencies. However, blockchains can also be designed and work appropriately without necessarily using cryptocurrencies . Therefore, there are distinct blockchain application domains and a diversity of blockchain taxonomies [11]. Zheng et al. study identified five categories of blockchain domains and they include: finance security, reputation systems, IOT and public service. Nevertheless, some of these application domains are faced with extreme challenges such as inadequacy of available consensus protocols, propensity for centralization, privacy leakage, vulnerability and scalability. The most recommended blockchain taxonomy is based on how easily a blockchain network can be accessed and it includes public and private networks. Public networks consist of Ethereum and Bitcoin; anyone can operate them

and perform transactions here while private networks are normally operated by companies that desire to work in a supervised, distributed and managed surrounding but still remain private. Considering private networks, blockchain employs distinctive filters such as who is able to mine and transact on the network. In conclusion, a variety of use cases of blockchain technology can be applied in different domain applications. However, Ethereum introduces a second generation of blockchain which can be used to create complex applications .

## 2.2 Ethereum

Ethereum serves as both a payment system and a computing platform. Ethereum is termed as the global computer and as the future internet which is capacitated by the blockchain technology [12]. These suggestions arose from the Ethereum's novel concept of distributed computational processing of applications with total clarity and with no third party control. Buterin et al. study presents major components and features of Ethereum, beginning with its general architectural design, mining process, state, method of transaction, financial derivative, decentralized autonomous organizations and collections of concerns and challenges[13]. In Ethereum, the fundamental object state is an account. Each account is related to a 20-byte address and an ether balance. It has two distinctive accounts: externally owned own which belong to system users (operated by the private of clients) and contract accounts which autonomous accounts are managed by contract nodes. Contract accounts are related to a code and storage which can be accessed and altered by the code. Transactions and messages have an impact on state changes. Transactions are normally created by external accounts; a messaged created by the contract and once a transaction or a message has been sent to the contract a count, it causes the related code to be executed. Finally, the sender will sign the using personal private key and attaches a signature on the transaction. The signature derives the public key and the sender's address. Once the contract has received transactions, the code will be executed eventually. V.P ether is deducted from the user's account before any execution takes place and if the balance is insufficient, the computation has v 'gas' available. Then the computation begins; each instruction in the Ethereum virtual machine (EVM) have cost in terms of gas [14]. The instructions are executed till no gas is available. Smart contract contains distinct storage, and in Ethereum they act as external owned accounts. They represent self-certifying and self-autonomous agents which are kept in the blockchain and also constitute fields and functions [15]. At the time they are positioned in the blockchain they occupy distinctive addresses where users interact and it is termed as contract account to

distinguish it from external accounts which are operated by humans. A code that is reserved in the blockchain succeeding the positioning is a low-level stack-based bytecode which represents high programming languages such as solidity and JavaScript. The behaviour of smart contracts is fully anticipatable and their codes can easily be inspected by every node within the network since their bytecode is publicly available in the blockchain.

## **2.3 smart contract**

The current project critically examines smart contracts, and the vulnerabilities that characterize them, and as has been discussed and analysed in this project, smart contracts are just code, and just like any other code, may face or have vulnerabilities. These vulnerabilities and bugs can quite easily be exploited by people of malicious intent. Smart contracts, according to facilitate, and permit trusted transactions, as well as agreements to be carried out among unrelated, anonymous parties. An important aspect about smart contracts as has been pointed out in this study is that they do not need a central authority, any external enforcement authority or a legal system. In other words, these smart contracts are supposed to render these transactions transparent, traceable, and irreversible.

Smart contracts could be defined as a set of coded operations that can be executed automatically when a person sends an input to the contract. The smart contracts are behind the popularity of Ethereum network, as well as its cryptocurrency, Ether adds that the concept of smart contracts is what powers a majority of ICOs today, however, it is important that ICOs also run on other Ethereum-based tools and services.

What this implies is that smart contracts are computerized transaction protocols that can be used to execute the terms of contract, and since they are just code, it is very possible for a hacker to exploit a vulnerability or a bug and steal a lot of money or generally temper with the system. Innovative technological advancements are introduced from time to time aiming at coming up with a novel, as well as better approaches of implementing systems and software products in better ways to ensure that all the loopholes are sealed. There is, therefore need to use technology to achieve this, and one of the such technologies is blockchain.

Vulnerabilities in smart contracts are a huge problem, since they lead to losses, and this is why different researchers have suggested different approaches that could be used to detect vulnerabilities in smart contracts. As hinted earlier in this paper, one of them could be the use of Mythril. Mythril constitutes alternative three input options; the first option is solidity code

file, and this is discussed in the next sub section. This option, as researchers have established, only works in situations where the solc command line compiler is installed. Other options apart from this one include solidity bytecode, and thus could be used in situations where the user does not have solc. Having examined how vulnerable a smart contract can get and how hackers can, sometimes, easily exploit a bug; it becomes clear that writing a safe smart contract is not an easy job as would be demonstrated in this project. Writing a safe smart contract requires different security considerations from our traditional software development, especially solidity, which allows a developer to customize modifier is part of the key steps that the programmers must take to ensure they code contracts that have minimal or no vulnerabilities. Different structures or detail of modifier may concern user to know about code because the same name on modifier like “onlyOwner” will have totally different implementation, and this is highly dependent on who is developing.

## **2.4 Solidity**

Solidity is a static type programming language for developing intelligent contracts running on EVM. Solidity is compiled to be byte code that can be executed on EVM. With Solidity, developers can write applications that implement the self-executing business logic contained in smart contracts, leaving undeniable and authoritative transaction records. Writing smart contracts in a smart contract specific language (such as Solidity) is called simple (seemingly simple) for people who already have programming skills.

As Wood pointed out, it is designed around the ECMAScript syntax to familiarize existing Web developers; unlike ECMAScript, it has static types and variable parameter return types. Solidity contains many important differences from other EVM target languages of the time, such as Serpent and Mutton. Supports complex membership variables including any hierarchical mapping and structural contract. Contracts supports inheritance, including multiple inheritance of C3 linearization. An application binary interface (ABI) was introduced that facilitates multiple type-safe functions (later supported by Serpent) in a single contract. A user-centric description of the branch used to specify method calls is also included in the proposal, called a "natural language specification".



## 2.5 Remix-ide

Remix, as would be established, is one of the most powerful tools that help in the writing of solidity contracts straight from the browser. This would be demonstrated throughout the paper.

Remix is a powerful, open source tool that helps you write Solidity contracts straight from the browser. Written in Javascript, Remix supports both usage in the browser or locally.

Remix also supports testing, debugging and deploying of smart contracts and much more.

## 2.6 EVM

Ethernet virtual machine (EVM) is the runtime environment of intelligent contracts in Ethernet. It is a 256 bit register stack designed to run the same code exactly as expected. This is the basic consensus mechanism of Ethernet. The formal definition of EVM is detailed in the Yellow Book of the Tai Fang. It is a sandbox and is completely isolated from other processes such as network, file system or host computer system. Each Ethernet node in the network runs EVM to implement and execute the same instructions. On February 1, 2018, there were 27,500 nodes in the main Ethernet network. The Eherehere virtual machine has been implemented in C++, Go, Haskell, Java, JavaScript, Python, Ruby, Rust, and Web Assembly (currently under development).

## 2.7 cryptocurrency wallet

Encrypted currency wallet stores public and private keys that can be used to receive or use encrypted currencies. A wallet can contain multiple public keys and private key pairs. As of January 2018, there were more than 1300 encrypted currencies; the first and most famous one was bitcoin. The encryption currency itself is not in the wallet. In the case of derivatives from bitcoin and encrypted currency, encrypted currency is stored and maintained in a publicly available ledger. Each encrypted currency has a private key. Using the private key, it can be written in the public ledger and effectively use the relevant encryption currency.

## 2.8 NPM

NPM is the package manager of the JavaScript programming language. It is the default package manager for JavaScript runtime environment Node.js. It consists of a command line client (also known as npm) and an online database of publicly paid private packets (called the NPM registry). Through the client access to the registry, you can browse and search the

available packages through the NPM website. The package manager and registry are managed by npm, and Inc. NPM is included as a recommended feature in the Node. JS installer. NPM contains a command line client that interacts with remote registries. It allows users to use and distribute JavaScript modules available in the registry. The package on the registry is in CommonJS format and contains metadata files in JSON format. There are more than 477000 packages on the main NPM registry. The registry does not submit a review process, which means that the packets found there may be low-quality, unsafe, or malicious. Instead, NPM relies on user reports that delete packages if they violate policies because of poor quality, insecurity, or malice. NPM exposes statistics, including the number of downloads and the number of dependent packages, to help developers determine the quality of packages.

In npm version 6, the audit feature was introduced to help developers identify and fix vulnerability and security issues in installed packages. The source of security issues were taken from reports found on the Node Security Platform (NSP), and has been integrated with npm since npm's acquisition of NSP.

NPM manages packages that are local dependencies for a particular project, as well as JavaScript tools that are installed globally. When used as a dependency manager for a local project, NPM can install all the dependencies of the project in a single command through the package. JSON file. In the package. JSON file, each dependency can specify a set of valid versions using a semantic version control scheme, allowing developers to automatically update their packages while avoiding unnecessary major changes. NPM also provides version enhancement tools for developers to mark their packages with specific versions. NPM also provides a package-lock.json file that, after evaluating semantic version control in package.json, has the exact version entries used by the project.

## **2.9 Nodejs**

Node.js is an open source, cross-platform JavaScript runtime environment that executes JavaScript code outside the browser. Historically, JavaScript has been used primarily for client-side scripting, where scripts written in JavaScript are embedded in the HTML of Web pages and run on the client side through the JavaScript engine in the user's Web browser. Node.js allows developers to use JavaScript to write command-line tools and server-side scripts to run script server-side to generate dynamic Web content before the page is sent to the user's Web browser. Therefore, Node. JS represents an example of "JavaScript is ubiquitous" and unifies Web application development around a single programming language, rather than different languages for server-side and client-side scripting. Although. JS is a regular file

extension of JavaScript code, the name "Node. js" does not refer to a particular file in this context, just the name of the product. Node.js has an event driven architecture capable of asynchronous I / O. These design choices are designed to optimize throughput and scalability in Web applications with many input / output operations, as well as for real-time Web applications (such as real-time communicators and browser games).

There are thousands of open-source libraries for Node.js, most of them hosted on the npm website. The Node.js developer community has two main mailing lists and the IRC channel #node.js on freenode. There are multiple developer conferences and events that support the Node.js community including NodeConf, Node Interactive and Node Summit as well as a number of regional events.

The open source community has developed the Web framework to speed up the development of applications. Such frameworks include Connect, Express. js, Socket. IO, Koa. js, Hapi. js, Sails. js, Meteor, Derby, and so on. Various software packages have been created to connect to other languages or runtime environments, such as Microsoft. Net.

Modern desktop IDE specifically provides editing and debugging functions for Node.js applications. Such IDEs include Atom, Brackets, JetBrains WebStorm, Microsoft Visual Studio (with Node. JS Tools for Visual Studio, or TypeScript with node definitions), NetBeans, Nodeclipse Enide Studio [64] (based on Eclipse) and Visual Studio Code. Certain Web-based online IDEs also support Node. js, such as Codea Studio Visual flow editor in nywhere, Codenvy, Cloud9 IDE, Koding and Node-RED.

### 3 Integer overflow

Blockchain technology has an edge over traditional software if the security for solidity is strong enough to avoid lost or hacked ether. Smart contract security is critical, and techniques that can analyze the solidity programs are being devised by researchers[16]. Even though blockchains use symmetric and asymmetric cryptography to reinforce security around smart contracts, a few red flags have been realized that compromise this security. The warnings arise from eclipse attacks and transactions malleability such as transaction ordering dependence, mishandled exceptions and integer overflow and underflow attacks which allow hackers to access change and manipulate transactions before the cryptocurrency networks can detect and confirm it. Formal verification, *a process of verifying a computer program to ensure it satisfies certain formal statements*, is used for solidity to enhance smart contract security. While writing contracts, it is essential to consider software bugs that may surface as a result of poorly written codes granting hackers the ability to manipulate transactions.

Vulnerabilities on integers surface when applications fail to notice irregularities and inconsistencies in the way processors handle the length values, before and after performing the operations. Integer overflow is a condition that involves an integer value that exceeds the given range that can be represented by the maximum value. This error occurs when the integer variable in Solidity is used without any safety checks. The minimum value of a unit is 0, and the maximum representable value is  $2^b-1$ , where  $b$  is the value in bits, so where a program uses uint 8 the maximum value is 255, represented as  $2^8-1$  [17]. When uint 8, which is limited to 256 numbers, is assigned a number that is out of range of 0 to  $2^8$  either a value larger than the maximum or lower than the minimum, then an integer overflow occurs. When a user has functions that can manipulate the balance to reaches the maximum unit value, the system becomes insecure and vulnerable to integer overflows. When a system is forced to make calculations based on tokens out of thin air, violating accuracy and common sense of systems that perform within a limited range, it raises alarms in the cryptocurrency networks. Such flaws are common in weak management systems and lack of range checking in solidity language and low-level system programming that makes a system susceptible to software exploits such as de reuse attacks. Medium Corporation gives an example of the Y2K problem which was accredited to the turn of the millennium, the year 2000, which programs stored it as 00 since systems stored calendar data in a two digit format hence reverting it to 1900[18]. There are several types of integer overflows. The first one is the overflow, where the result of the operation exceeds the maximum value of the integer variable that is used to store it and

the balance circles to zero. The second integer overflow is the underflow, where the integer unit is less than the minimum value, which is 0 and causes the unit to be circled to its maximum value. The third caveat is noticed when signedness errors occur, where an application confuses signed and unsigned integers by assigning a function that has an unsigned value to a signed integer. The integer overflow leads to exploitation vulnerabilities such as Denial of Service, logic errors and bypass arbitrary codes and sanitization checks that make a system insecure. However, it is difficult to trace integer overflows directly, as more often than not, they are embedded deep in a program making it hard to develop mitigation measures. When the integer overflow occurs, the application that it runs on executes calculations with the assumption that the operation is correct [19].

As the longest and most classic type of vulnerability on smart contract, Integer overflows and underflows are always detected from many contract in a new form. The Solidity language does not support floating-point data types like float double in C supports signed or unsigned integers with int/uint lengthening. The variable supports from uint8 to uint256, and int8 to int256. It is important to consider that uint and int default to uint256 and int256. The value range of uint8, is the same as uchar in C, that is, the value range is 0 to  $2^8-1$ , and the range of values supported by uint256 is 0 to  $2^{256}-1$ , and the rest of the data types and so on. Basically, the data type that is used most in solidity is uint256. As mentioned before, it ranges 0 to  $2^{256}-1$ .

An integer overflow occurs in the event that an arithmetic operation has attempted to create a numeric value that is beyond the range or outside the range that can be represented using a given number of bits. What this means that the number could either be lower or smaller than the minimum or larger than the maximum representable value.

### **3.1.1 Code demo**

In computer programming, an integer overflow occurs when an arithmetic operation attempts to create a numeric value that is outside of the range that can be represented with a given number of bits – either larger than the maximum or lower than the minimum representable value, has been defined, and in this case, it is important to examine this phenomenon in the context of smart contracts. Usually, in solidity, this phenomenon would most likely occur in such situation:

Overflow in conversion between unsigned and signed; overflow in addition or multiplication of two (un)signed numbers, underflow in subtraction of two (un)signed numbers; overflow in ++ on a (un)signed number underflow in -- on a (un)signed number;

overflow in += ;overflow in -= ;overflow in \*= ;overflow in /=

The code of figure 1 and result from figure 2 will show what is flow:

```
pragma solidity ^0.4.20;
contract C {
  // (2**256 - 1) + 1 = 0
  // 0 - 1 = 2**256 - 1
  function flow() returns (uint256 _overflow,uint256 _underflow) {
    uint256 max = 2**256-1;
    _overflow=max+1;

    uint256 min = 0;
    _underflow=min-1;
  }
}
```

Figure 1: Code demo of flow

|                  |   |
|------------------|---|
| status           | 0x1 Transaction mined and execution succeed   |
| transaction hash | 0x47fda208b6402f5b61dcd5a2341f942c7123a7d7e141250cbe0e5199b4c01e6f  |
| from             | 0x0a35b7d915458ef540ade6068dfa2f44e8fa733c  |
| to               | C.flow() 0x038f160ad632409bfb18582241d9f88c1a072ba  |
| gas              | 3000000 gas   |
| transaction cost | 21489 gas   |
| execution cost   | 217 gas   |
| hash             | 0x47fda208b6402f5b61dcd5a2341f942c7123a7d7e141250cbe0e5199b4c01e6f  |
| input            | 0x343...aad82   |
| decoded input    | {}  |
| decoded output   | {           "0": "uint256: _overflow 0"           "1": "uint256: _underflow 115792089237316195423570985008687907853269984665640564039457584007913129639935"         } |
| logs             | []  |
| value            | 0 wei   |

Figure 2: Result of flow

As can be seen, after overflowing, overflow directly wraps around with a return value of 0. When uint256 takes 0 underflow, it directly wraps around, and the return value is  $2^{256}-1$ . This is the normal case of integer overflow scenarios in solidity.

### 3.1.2 Official Fix

To fix overflow and underflow is easy, there is already official way to deal with it, using SafeMath library in contract. After import the library from figure 3 and reedit the code like figure 4 and figure 5

```
/**
 * @title SafeMath
 * @dev Math operations with safety checks that throw on error
 */
library SafeMath {
    function mul(uint256 a, uint256 b) internal constant returns (uint256) {
        uint256 c = a * b;
        assert(a == 0 || c / a == b);
        return c;
    }

    function div(uint256 a, uint256 b) internal constant returns (uint256) {
        // assert(b > 0); // Solidity automatically throws when dividing by 0
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold
        return c;
    }

    function sub(uint256 a, uint256 b) internal constant returns (uint256) {
        assert(b <= a);
        return a - b;
    }

    function add(uint256 a, uint256 b) internal constant returns (uint256) {
        uint256 c = a + b;
        assert(c >= a);
        return c;
    }
}
```

Figure 3: Safemath library

```
contract C {
    // (2**256 - 1) + 1 = 0
    // 0 - 1 = 2**256 - 1

    using SafeMath for uint256;
    function flow() returns (uint256 _overflow, uint256 _underflow) {
        uint256 max = 2**256-1;
        _overflow=max.add(1);

        uint256 min = 0;
        _underflow=min.sub(1);
    }
}
```

Figure 4: Modification on code

If there is still flow happen, the results of this operations will be checked and an error will be thrown stopping the execution of your smart contract.

### 3.1.3 Vulnerability demo

In this part, there will be two simple code to show how the vulnerability may happen

```
function votes(uint postId, uint upvote, uint downvotes) {
  if (upvote - downvote < 0) {
    deletePost(postId)
  }
}
```

Figure 5: Simple flow demo

As show in figure 5, the function is a depiction of a operation result which is an unsigned integer on two unsigned integers , and what this means is that they will never get a true negative number but only wrap around the range of uint256. So this function never pass the check in “if” case.

```
for (var i = 0; i < 1000; i ++) {
}
```

Figure 6: Endless loop by flow

The second example explain the soon-to-be-deprecated var keyword. Because while compiling the variable with keyword var is always regarded as the smallest type enough to contain the assigned value “0”, it will be compiled as an uint8,which can only value max to 255 to hold the initial value 0. In this loop,if cycle more than 255 times, it never reach the max range number, keep wrap around from 0 to 255, stop until the execution runs out of gas.



## 3.2 Proxy Overflow

### 3.2.1 Introduction of ERC20

The ERC20 token standard describes the functions and events that an Ethereum token contract has to implement. Basically use like an interface, has the following method-related functions, `balanceOf()`, `transfer()`, `approve()`, some events like `Transfer` to record balance transaction and more.

### 3.2.2 Real World Case

There are quite a few ERC20 tokens affected by this overflow, here choose one of them “SMT” to demonstrate. The address of SMT contract:

<https://etherscan.io/address/0x55f93985431fc9304077687a35a1ba103dc1e081#code>

ProxyTransfer is a new EIP in a process wherein the transaction fees are paid by the users in the form of tokens and not Ether, as opposed to traditional ERC20 contracts. In a traditional ERC20 contract, even if the user has 100 tokens, he cannot simply operate unless he holds some Ether too. However, this does not really make sense as ‘Tokens themselves hold value and to say that tokens can only act in conjunction with Ether makes them unusable to a certain extent’. To fix this, a group of developers introduced a new proposal (EIP-xyz) where users can spend tokens as Transaction Fees. EIP gained a lot of traction that several DApps tried to incorporate, while the proposal itself is reviewed. However, some of those contracts came with an exploit and hidden cost[20].

```
205     function transferProxy(address _from, address _to, uint256 _value, uint256 _feeSmt
206     ) public transferAllowed(_from) returns (bool){
207
208         if(balances[_from] < _feeSmt + _value) revert();
209
210         uint256 nonce = nonces[_from];
211
212         uint256 temp=_value+_feeSmt;
213         if(balances[_to] + _value < balances[_to]
214            || balances[msg.sender] + _feeSmt < balances[msg.sender]) revert();
215         balances[_to] += _value;
216         Transfer(_from, _to, _value);
217
218         balances[msg.sender] += _feeSmt;
219         Transfer(_from, msg.sender, _feeSmt);
220
221         balances[_from] -= _value + _feeSmt;
222         nonces[_from] = nonce + 1;
223         return true;
224     }
```

Figure 7: Overflow from transferProxy()

To make to convenient for test, here remove the verification code of digital signature from token sender. As the code shown in figure 8, before running this function, system will check modifier of transferAllowed()

```
modifier transferAllowed(address _addr) {
    if (!exclude[_addr]) {
        assert(transferEnabled);
        if(lockFlag){
            assert(!locked[_addr]);
        }
    }
    _;
}
```

Figure 8: TransferAllowed modifier

Basically after the contract is created on the chain, the owner will set transferEnabled to true. If the \_from address is in exclude mapping, it can pass this check. If not, the address can pass if it has not been locked by the owner in the locked mapping. To summarize, the transferProxy function is allowed to run if the from address hasn't been locked and the contract is allowed to transfer. Then the overflow occurs in the first "if" case in figure 6. Both fee and value are input parameters which could be controlled by the attacker. If \_fee + \_value happens to be 0, the check in line 208 could be passed. It means the attacker could transfer huge amount of tokens to an address (line 215) with zero balance. Also, a huge amount fee would be transferred to the msg.sender in line 219. But in line 221, because of overflow, the address would not reduce any amount of balance. So this vulnerability does not lead to generating countless tokens which should not be exist.

### 3.2.3 Normal procedure

Here analyze the normal procedure to use transferProxy function: After create contract and initially allocate token for some accounts by contract owner, user A who have token on this contract but not have ETH may want to deal with other, C here. Because they need to trade with ETH because C have no tokens, so the third-party C who have digital sign for A will take this mission. And the procedure can be described like this: "A->C(token), C initial the transaction because he has permission from A, which allow to transfer token from A to C. Following their agreement, C will transfer equivalence ETH to B.

### 3.2.4 Demo

Using the uint data from this transaction to do test attack,



As data from figure 11, it is clear that both two address increase great amount of balance but without decrease any amount in `_from`. Actually the balances' increase depend on case: "balances[address]+=value;", so from line 215,218,221 in figure 6, only increase on these address but no decrease on `_from` address because overflow, it reduce 0.

According this demo, obviously it creates huge amount of non-existing Ethereum to attacker's account, leading collapse of Ethereum market. Still kind of contracts like GG Token, Mesh and so on, are exposed to proxyoverflow vulnerability.

### 3.2.4 Fix

Almost every contract that has Integer flow vulnerability can be fixed by the same way using safemath library, including this proxyoverflow.

```

252     function transferProxy(address _from, address _to, uint256 _value, uint256 _feeSmt
253     ) public transferAllowed(_from) returns (bool){
254
255         if(balances[_from] < feeSmt.add(_value)) revert();
256
257         uint256 nonce = nonces[_from];
258
259         uint256 temp= value+ feeSmt;
260         if(balances[_to].add(_value) < balances[_to]
261         || balances[msg.sender].add(_feeSmt) < balances[msg.sender]) revert();
262         balances[_to] += _value;
263         Transfer(_from, _to, _value);
264
265         balances[msg.sender] += _feeSmt;
266         Transfer(_from, msg.sender, _feeSmt);
267
268         balances[_from] -= _value + _feeSmt;
269         nonces[_from] = nonce + 1;
270         return true;
271     }

```

Figure 12:Modification on transferProxy function

Following the official method, after importing the safemath library, the necessary step to edit the code is to use add method from safemath to do the addition operation. If the result will overflow, it will revert the transaction back to initial state.

### 3.2.5 Malicious Use in Real World

Until now, most attackers use overflow to escape the amount check while transfer balance, some famous cases like batchoverflow, multioverflow, and two case will be introduced later. The strange or most interesting thing is that it is not difficult to implement an attack with flow; the attackers just need to find out a contract with token or transfer function which forgets to make sure all the checks have been done in the function use safemath, then attackers can try

until the input data flows so that can pass the check. This always behaves in a way that facilitates the transfer of much of balance to some address with no or very little consumption from msg.sender or from address in the Input parameters. As the result, it creates a lot of nonexistent Ethereum or token, which may lead the market break down.

## 3.3 BurnOverflow

### 3.3.1 Real World Case

The vulnerability of overflow always lead to generate tokens out of nowhere including this burnoverflow. This case seems pretty simply because contract programmer just implement the logic of the standard ERC-20 transfer() function by calling his customized function with conventional check, still with some negligence[21]. Here is the contract address:

<https://etherscan.io/address/0xB5335e24d0aB29C190AB8C2B459238Da1153cEBA#code>

Now let's analyze the code structure:

```
23 ▾  /* PUBLIC */
24 ▾  /* Send tokens */
25 ▾  function transfer(address _to, uint256 _value) public returns (bool success) {
26      _transfer(msg.sender, _to, _value);
27
28      return true;
29  }
```

Figure 13: Implementation of standard transfer function

```
76 ▾  /* INTERNAL */
77 ▾  function _transfer(address _from, address _to, uint _value) internal {
78 ▾      /* Prevent transfer to 0x0 address. Use burn() instead */
79 ▾      require (_to != 0x0);
80 ▾      /* Check if the sender has enough */
81 ▾      require (balanceOf[_from] >= _value + burnPerTransaction);
82 ▾      /* Check for overflows */
83 ▾      require (balanceOf[_to] + _value > balanceOf[_to]);
84 ▾      /* Subtract from the sender */
85 ▾      balanceOf[_from] -= _value + burnPerTransaction;
86 ▾      /* Add the same to the recipient */
87 ▾      balanceOf[_to] += _value;
88 ▾      /* Apply transaction fee */
89 ▾      balanceOf[0x0] += burnPerTransaction;
90 ▾      /* Update current supply */
91 ▾      currentSupply -= burnPerTransaction;
92 ▾      /* Notify network */
93 ▾      Burn(_from, burnPerTransaction);
94 ▾      /* Notify network */
95 ▾      Transfer(_from, _to, _value);
96 ▾  }
```

Figure 14: Detail of \_transfer function





As the transaction initiator is contract creator who has balance on the contract, it can finish the transaction to send balance to other address. So after the transaction check the public mapping variable `balancedOf` with the address just send balance, can successfully find out 1 value balance.

[illegible]

Figure 17: Test with normal revert situation

Because the test address has no balance on contract, as the transaction initiator, it cannot pass the check in line 81 from figure 14 as

(balanceOf [ from ]) = 0 < 3 = ( value + burnPerTransaction ), resulting in the transaction revert.

Then following how to attack.

|                  |   |
|------------------|---|
| status           | 0x1 Transaction mined and execution succeed   |
| from             | 0x14723a09actf6d2a60dcd7a44af1308fddc160c   |
| to               | Hexagon: transfer(address,uint256) 0x08970fed061e7747cd9a38d680a601510cb659fb   |
| gas              | 8000000 gas   |
| transaction cost | 65933 gas   |
| execution cost   | 41077 gas   |
| input            | 0x9059ebb00000000000000000000000000000583031d1113ad414f02576bdeafabfb302140225ff  |
| decoded input    | <pre>{   "address": "to": "0x583031D1113aD414F02576BDeaFABfB302140225",   "uint256": "value": "115792089237316195423570985008687907853269984665640564039457584007913129639934" }</pre>  |
| decoded output   | <pre>{   "0": "bool: success true" }</pre>  |
| logs             | <pre>[   {     "from": "0x08970fed061e7747cd9a38d680a601510cb659fb",     "topic": "0x0c16f5db4873280815c1e09dbd06736ff0c184412cf7a71a0fdb75d397c45",     "event": "Burn",     "args": [       {         "0": "0x14723A09ACff6D2A60Dcd7A44AFf308FDDC160C",         "1": "0x14723A09ACff6D2A60Dcd7A44AFf308FDDC160C",         "from": "0x14723A09ACff6D2A60Dcd7A44AFf308FDDC160C",         "value": "2",         "length": 2       }     ]   },   {     "from": "0x08970fed061e7747cd9a38d680a601510cb659fb",     "topic": "0xddf252ad1be2c89b69c2b068fc378da1952ba7f163c4a11628f55a4df523b3ef",     "event": "Transfer",     "args": [       {         "0": "0x14723A09ACff6D2A60Dcd7A44AFf308FDDC160C",         "1": "0x583031D1113aD414F02576BDeaFABfB302140225",         "2": "115792089237316195423570985008687907853269984665640564039457584007913129639934",         "from": "0x14723A09ACff6D2A60Dcd7A44AFf308FDDC160C",         "to": "0x583031D1113aD414F02576BDeaFABfB302140225",         "value": "115792089237316195423570985008687907853269984665640564039457584007913129639934",         "length": 2       }     ]   } ]</pre> |

Figure 18: Test with overflow

As shown in figure 19, the sender has no balance on contract but make the transaction success.

### 3.3.3 Fix

```
123 ▾ /* INTERNAL */
124 ▾ function _transfer(address _from, address _to, uint _value) internal {
125 ▾     /* Prevent transfer to 0x0 address. Use burn() instead */
126 ▾     require (_to != 0x0);
127 ▾     /* Check if the sender has enough */
128 ▾     require (balanceOf[_from] >= _value.add(burnPerTransaction));
129 ▾     /* Check for overflows */
130 ▾     require (balanceOf[_to].add(_value) > balanceOf[_to]);
131 ▾     /* Subtract from the sender */
132 ▾     balanceOf[_from] -= _value + burnPerTransaction;
133 ▾     /* Add the same to the recipient */
134 ▾     balanceOf[_to] += _value;
135 ▾     /* Apply transaction fee */
136 ▾     balanceOf[0x0] += burnPerTransaction;
137 ▾     /* Update current supply */
138 ▾     currentSupply -= burnPerTransaction;
139 ▾     /* Notify network */
140 ▾     Burn(_from, burnPerTransaction);
141 ▾     /* Notify network */
142 ▾     Transfer(_from, _to, _value);
143 ▾ }
```

[illegible]

25



ERC20 contract inherit from Token. Finally, ensure that you finish your own contract inherited from ERC 20 with customize code to call standard as you like.

### 3.4 Summary

Seem the current compiler cannot perfectly fix the integer overflow and underflow, since such flow vulnerability already caused great damage on block chain. But if safemath is imported into compiler as default detector for all uint or int type, that would cause great gas to run it. Execution of Ethereum smart contracts is limited by the gas limit on each block which is currently around 6m. This only allows for between one thousand and one million instructions in each block (every 15 seconds). Checking for overflow a million times each second is not expensive on processors produced in this millennium. It is, however, also important to note that it would be bad for overall EVM performance and by extension bad for tx rate, node performance. To prevent flow vulnerability mostly depend on contract developer themselves. However, quite many contract developers can not realize the severity yet, forget to check the vulnerability then allow malicious attacks, leading many time of revert and waste of space on chain. There is no way to prevent this Vulnerability, the only remaining option is to make code normalization.

As for flow, these steps may be useful:

- After declaring the version of pragma solidity, import the safemath library.
- After defining contract, use safemath for uint256 at first.
- Use uint256 but not other bits of uint as far as possible.
- Check every arithmetic operation handle with outside data like the input parameter, then use method add,sub,mul,div from safemath instead of +,-,\*,/.
- Test your contract, especially the transfer or withdraw function.

## 4 Access Permission and Custom Modifier

Blockchain-based platforms such as Ethereum have allowed the implementation of smart contracts which can ensure that contractual terms are expertly executed. Before users access the blockchain system, they have to be authenticated and identified to be granted access permission. Businesses are rapidly attempting to invest in the blockchain technology as a trend that can reform systems for improved performance [22]. The popularity of blockchain technology arises from its decentralization ability which allows the system to be always active; information is secure and acts as a database to permanently store data. The data that is stored in the blockchain system is visible and accessible by all users. Due to this public accessibility, programmers initiate codes that govern the modification and access of information.

Access permission allows users to get customized access to the blockchain state determining which users are able to view the transaction data. Certain individuals may be granted access to see relevant transactions such as auditors even though the level of access may be obstructed for security purposes. In the business scenario, permission to access information is given to the members in different ways [23]. The first option is through the proof of stake, where members are to consistently demonstrate their level of aggregate value they hold in the stake to ease the confirmation process. The method reduces the decentralized confirmation process while consecutively giving security assurances as a result of the costly execution. The second way of verifying transaction is through a multi-signature approach. This technique ensures that the dominant members approve transactions before they are legitimized. This ensures that attackers are unable to bypass the protocols that are executed by restricting their access. The third option that is offered by blockchain is the Practical Byzantine Fault Tolerance (PBFT). This algorithm ensures that similar nodes are yielded when networks members in an arrangement are different from one another to achieve consensus.

With reference to access permission, blockchains can either be permissioned or ‘permissionless’/public according to the type of control granted to the users[24]. The permissioned blockchain applications are further divided into the consortium and private blockchains. Consortium blockchains control the consensus process, which is a process that ensures members come to a unified decision despite the hurdles, through a set of nodes that are predefined. The access of the users is limited to either writing or reading information hence it has limited decentralization. In the private blockchain, however, the power to access information lies in a central organization who give permission to read or write data. The organization can, however, provide access to the public to an arbitrary extent. According to

Hofmann, Strewe, and Bosia, the permissioned blockchains augment the validation process, ensure cheaper transactions, security is high since fewer nodes are involved and are entirely controlled with no crypto economics such as proof of work that is energy intensive[25].

In 'permissionless' blockchains, the network is uncontrolled, and access is given to anyone with an internet connection. In the case of Bitcoin and Ethereum, users can contribute to the consensus process, read, and write data since the code is open source hence anyone can build, analyze, and participate in transaction validation. The consensus mechanism involves crypto economics such as the proof of work system which Ethereum developers plan to move to, from a proof of work system, where the validators will be used as a resource-friendly strategy. The degree to which user can participate in the consensus is dependent on the percentage of the stake that they can bear in the blockchain. Despite of the permissioned blockchains having attractive benefits to institutions, public blockchains are able to gain network benefits from the unrestricted access by institutions allowing direct transactions in the absence of third parties and ensures that no particular user has special privileges over the other in the decision-making process. The system is secure from malicious attacks because of the decentralization feature making it mathematically expensive to hack into the validation nodes that are not known [26].

Smart contracts can restrict the access that users have to transaction information by using harder encryption such as modifiers. Modifiers ensure that certain conditions are adhered to before executing any coding, giving additional functionality to solidity [27]. With function modifiers, the function behavior is easily modified and checks the behavior of conditions before they are executed. For example, the modifier 'onlyOwner' is used to verify if whether the user is the owner of the contract to determine whether to execute the code in the rest of the body. The custom modifier allows the solidity to restrict access to the kill function giving an additional modification to the function. The modifier ensures that the validation passes some form of rules before the functions are executed in a readable manner.

The largest feature of blockchain is the fact that it is fully public. It is always open source for its transaction, and this includes some simple details like what function call is expected during the transaction on chain, in the meantime, all users can check pending transactions, code for each contract and token. In some cases, someone some back door or vulnerability in their contract that will make it exposed to criminals who try to attack and steal Ethereum. A good example could be when some careless programmer forgets to set limitation for their function so that external personnel can directly call the owner function. This part is about access

permission in solidity. About custom modifier, this is also pretty useful technology in solidity, and as such, there is need to introduce with code because this vulnerability usually affects the key work “onlyOwner”.

## **4.1 ceoAnyone**

### **4.1.1 Introduction of crypto-games**

Blockchain-based crypto-games have become popular especially with the success of CryptoKitties and EtherCartel because of its low investment since the end of 2017. Crypto idle game, one of Crypto game, is a funny game that allows players to make money by idling for hours. Later players can trade with goods generated from the game and get profit-making transactions. Players spend little time but earn quite a lot of currency on Bitcoin or Ethereum. Many of the crypto idle game owners make profit from the transaction fee. However, some owners leave some vulnerability that an attacker can use to tamper the owner of the contract and steal the profit. Such vulnerability is fully based on the developer's negligence.

### **4.1.2 Code demo**

Smart contracts are totally open source on blockchain, and part of the reason why is to distinguish and make it easy to manage. Solidity allows programs to customize modifiers, which also accept input parameters. Many developers in the recent past have used a common modifier “onlyOwner”

If the function is defined with onlyOwner; this means before running the code in the function, it will check the message sender's address. If it is the owner (maybe some contract will allow multiple owners), it will go ahead and continue into the function, otherwise it would be reverted.

```

contract owned {
    address public owner;

    function owned() {
        owner = msg.sender;
    }

    modifier onlyOwner {
        if (msg.sender != owner) revert();
        _;
    }

    function transferOwnership(address newOwner) onlyOwner {
        owner = newOwner;
    }
}

contract ownerTest is owned{
    function test() public onlyOwner returns(bool){
        return true;
    }
}

```

Figure 21: Basic model of onlyOwner

From figure 21, what has been shown is what can be referred to as a general purpose code structure. It is always applied as a father class, once there is other contract to inherit, programmer can use onlyOwner to modify the function. Every transaction call test method needs to pass onlyOwner check;

|                  |   |
|------------------|---|
| status           | 0x1 Transaction mined and execution succeed                 |
| from             | 0xc3a35b7d915458ef540ade6068dfe2f44e8fa733c                 |
| to               | ownerTest.test() 0x5e72914535f202659083db3a02c984188fa26e9f |
| gas              | 3000000 gas   |
| transaction cost | 21806 gas   |
| execution cost   | 534 gas   |
| input            | 0xf8a8fd6d  |
| decoded input    | {}  |
| decoded output   | {<br>"0": "bool: true"<br>}                                 |
| logs             | []  |
| value            | 0 wei   |

Figure 22: Test with owner address

|                  |   |
|------------------|---|
| status           | 0x0 Transaction mined but execution failed                  |
| from             | 0x14723a09acff6d2a60dcd77aa4aff308fddc160c                  |
| to               | ownerTest.test() 0x5e72914535f202659083db3a02c984188fa26e9f |
| gas              | 3000000 gas   |
| transaction cost | 21724 gas   |
| execution cost   | 452 gas   |
| input            | 0xf8a8fd6d  |
| decoded input    | {}  |
| decoded output   | {<br>"0": "bool: false"<br>}                                |
| logs             | []  |
| value            | 0 wei   |

transact to ownerTest.test errored: VM error: revert.  
 revert The transaction has been reverted to the initial state.  
 Note: The constructor should be payable if you send value. Debug the transaction to get more information.

Figure 23: Using non-owner address to test

As shown in figure 22 and 23, first switch the address to the contract creator, also the owner at the same time in this demo, then call the test function. Obviously, pass the modifier and return true, meaning transaction success. While the second test, using non-owner address to call, only get the false and revert.

Besides this model, there are other types of custom modifiers, which normalize and set rules for smart contract. While the code is fully open source, but the developer need to set some function to manage or maintain their contract. It is also important to ensure that unauthorized persons or strangers are allowed to abuse this part, and when this is done, such type of modifier becomes a better choice. In general, the modifier makes the function in a way that it can only be accessed by owner but not unrelated persons. The real question here, therefore, is where and how does the vulnerability happen?

### 4.1.3 Real World Case

This is one of contract affected by ceoAnyone and contract address:

<https://etherscan.io/address/0x5088b94cf8a1143eb228b6d3f008350ca742ddc2#code>

```

39  function DrugDealer() public{
40      ceoAddress=msg.sender;
41  }
42
43  function sellDrugs() public{
44      require(initialized);
45      uint256 hasDrugs=getMyDrugs();
46      uint256 drugValue=calculateDrugSell(hasDrugs);
47      uint256 fee=devFee(drugValue);
48      claimedDrugs[msg.sender]=0;
49      lastCollect[msg.sender]=now;
50      marketDrugs=SafeMath.add(marketDrugs,hasDrugs);
51      ceoAddress.transfer(fee);
52      // balances[ceoAddress]+=fee;
53      msg.sender.transfer(SafeMath.sub(drugValue,fee));
54  }
55  function buyDrugs() public payable{
56      require(initialized);
57      uint256 drugsBought=calculateDrugBuy(msg.value,SafeMath.sub(this.balance,msg.value));
58      drugsBought=SafeMath.sub(drugsBought,devFee(drugsBought));
59      ceoAddress.transfer(devFee(msg.value));
60      //balances[ceoAddress]+=devFee(msg.value);
61      claimedDrugs[msg.sender]=SafeMath.add(claimedDrugs[msg.sender],drugsBought);
62  }

```

Figure 24 :A ceoAnyone-affected Crypto Idle Game Smart Contract

Figure 24 shows code of such a smart contract of a crypto idle game named Ether Cartel. Similar to Ether Shrimp Farm that requires end users to hatch and sell shrimp in a bid to maximize production before ultimately exchanging eggs for ether, Ether Cartel takes the same concept but applies it to drug running. The game “features a high tech automated market that lets you instantly buy or sell drugs with a single transaction. The more kilos you have, the more drugs they produce (each kilo produces at a rate of 1 per day). Collect more kilos with your drugs to multiply your production.”[28].The problem occurs in lines 39-41: there is a public function DrugDealer() that allows all users to modify the beneficiary address — ceoAddress. Whenever buyDrugs() or sellDrugs() is called by other users do transaction in the contract,the ceoAddress collects the fee . To attack this contract is really easy, just call the DrugDealer function to modify the ceoAddress make it transfer value to the address you want.

#### 4.1.4 Attack demo

To make it convenient in observing the attack, add a public mapping variable balances to record the send of value using the data following the first few records in the contract given:

```

mapping [address => uint256] public balances;
function sellDrugs() public{
    require(initialized);
    uint256 hasDrugs=getMyDrugs();
    uint256 drugValue=calculateDrugSell(hasDrugs);
    uint256 fee=devFee(drugValue);
    claimedDrugs[msg.sender]=0;
    lastCollect[msg.sender]=now;
    marketDrugs=SafeMath.add(marketDrugs,hasDrugs);
    ceoAddress.transfer(fee);
    balances[ceoAddress]+=fee;
    msg.sender.transfer(SafeMath.sub(drugValue,fee));
}
function buyDrugs() public payable{
    require(initialized);
    uint256 drugsBought=calculateDrugBuy(msg.value,SafeMath.sub(this.balance,msg.value));
    drugsBought=SafeMath.sub(drugsBought,devFee(drugsBought));
    ceoAddress.transfer(devFee(msg.value));
    balances[ceoAddress]+=devFee(msg.value);
    claimedDrugs[msg.sender]=SafeMath.add(claimedDrugs[msg.sender],drugsBought);
}

```

Figure 25: Code of EtherCartel contract

DrugDealer

getFreeKilo

seedMarket

drugs:

0x0012c

transact

sellDrugs

Figure 26: Function to set up contract

Following these steps to set up contract and allow transaction:

1. Create contract with address “A”
  - 2.call DrugDealer() with address “A”
  - 3.call seedmarket() with data in figure 5
  - 4.call getFreeKilo()
  5. call sellDrug()
- Then we can start normal deal:

|                  |   |
|------------------|---|
| status           | 0x1 Transaction mined and execution succeed                       |
| from             | 0x14723a09acff6d2a60dcdf7aad4aff308fddc160c                       |
| to               | EtherCartel.buyDrugs() 0x75a3a98f5696299071da253c1433a2661898103e |
| gas              | 3000000 gas   |
| transaction cost | 74366 gas   |
| execution cost   | 53094 gas   |
| input            | 0x9fa0f763  |
| decoded input    | {}  |
| decoded output   | {}  |
| logs             | [ ]   |
| value            | 150 wei   |

Figure 27: A success transaction





It is very clear in this case, to see that the second transaction transfers the fee to address C but not A because the function to change ceoAddress is public, everyone can easily call it to change address, causing great economic damage to the true owner.

### 4.1.5 Fix

It is never difficult to fix such vulnerability, just import father contract owned and modify the function DrugDealer to onlyOwner, but for this case, the structure of code seems not very compliant with a standard. Here, we can try to improve it.

```
address public ceoAddress;
mapping (address => uint256) public kilos;
mapping (address => uint256) public claimedDrugs;
mapping (address => uint256) public lastCollect;
mapping (address => address) public referrals;

function EtherCartel() public {
    ceoAddress=msg.sender;
    owner=msg.sender;
}

function DrugDealer() onlyOwner{
    ceoAddress=msg.sender;
}
```

Figure 31: Modification on code

In the case, the developer defines and initializes ceoAddress as default address which is not suitable. It is better to assign in the constructor. About ceoAddress and owner, maybe here there are two options: I replace ceoAddress with owner, then if owner transfers its ownership of contract, the new owner can directly obtain and get handling fee from transaction.

It is clear so far that transactions on blockchain are run by miner. In this case, to earn more processing fee, miners always seek transaction with high gas. It is also important to note that there are transactions calling same function but send by different address, miners preferentially run the one with high gas. Therefore, in the event that some transaction initiated by contractor to manage contract, but be preempted by attacker and lock the contract, the question is what can be done next in averting this situation. In this case we will introduce the vulnerability that allows anyone to modify owner and call manager function, conflicting with the true owner's transaction.

## 4.2 OwnerAnyone

### 4.2.1 Real World Case

This is one of contract affected by ownerAnyone:

<https://etherscan.io/address/0xcdcf0f66c522fd086a1b725ea3c0eeb9f9e8814#code>

```
136 ~ function unlockToken() onlyOwner {  
137     locked = false;  
138 }  
139  
140 bool public balancesUploaded = false;  
141 ~ function uploadBalances(address[] recipients, uint256[] balances) onlyOwner {  
142     require(!balancesUploaded);  
143     uint256 sum = 0;  
144     for (uint256 i = 0; i < recipients.length; i++) {  
145         balanceOf[recipients[i]] = safeAdd(balanceOf[recipients[i]], balances[i]);  
146         sum = safeAdd(sum, balances[i]);  
147     }  
148     balanceOf[owner] = safeSub(balanceOf[owner], sum);  
149 }  
150 ~ function lockBalances() onlyOwner {  
151     balancesUploaded = true;  
152 }
```

Figure 32: Part of code of AURA

```
function setOwner(address _owner) returns (bool success) {  
    owner = _owner;  
    return true;  
}  
modifier onlyOwner {
```

Figure 33: Part of code of Owned

As shown in figure 32 and 33, in AURA contract, they set limitation with onlyOwner modifier to some function, unlockToken and lockBalance. About unlockToken, it handles the change of variable locked, which will be checked in the transfer and approve the function, only run while locked is in false value or called by owner, and lockBalance also controls variable balancesUploaded, the function uploadBalances will only run if balancesUploaded is false, but the unlockToken function in Owned contract, developer just has to set as default modifier, no modifier, which will make this function possible to be accessed from outside like public.

Basically, the process to publish this contract on chain is as follows: after create contract, call uploadBalances function to allocate balance in contract, then lock variable balancesUploaded with lockBalance function. Later release the permission to transfer by calling unlockToken function. Now we try to attack it[29].

#### 4.2.2 Attack demo

After creating the contract, the owner is still calling uploadBalances to allocate token, maybe someone else could find the transaction and contract address from chain. Before the owner locks the balance, attacker can call setOwner function to change owner then call uploadBalances to allocate balances for his address on contract. Lastly, before owner realizes

[illegible]

Downloaded from <http://ajph.org/> on November 10, 2015

[illegible]

The same as fix ceoAnyone, and even much more easier comparatively ,in this case, we just need add onlyOwner modifier in function setOwner.

Figure 37: Modification on setOwner

Figure 38: Creation from owner

Figure 39: Attacker try to tamper owner address

#### 4.2.4 Summary

Writing a safe smart contract is not an easy job as has variously been demonstrated in this section. Writing a safe smart contract requires different security considerations from our traditional software development, especially solidity, which allows a developer to customize modifier. Different structures or detail of modifier may concern user to know about code because the same name on modifier like “onlyOwner” will have totally different implementation, which is always dependent on developers. In this part, two cases show that if a developer forgets to check access permission before publishing the contract, it may allow people to tamper the owner of contract so that attacker can call function as freely as they want. Sometime the time inconsis and gas of transaction may lead to problems. Besides, the transaction that is allowed will be truncated because it suddenly has no permission to access function, resulting to a situation where it remains pending on chain. We cannot over-emphasize the importance of smart contract auditing but here from this vulnerability we can conclude some rules to that need to be obeyed:

- If developers have some function that they need it to be called by owner, they should use onlyOwner correctly by using right template.
- Before publishing contract on the chain, check each function’s modifiers and visibilities.
- if possible, try to test contract with different addresses by remix-ide.

## 5 Trade Trap

Publicly ERC-20 tokens which can be traded have a recommendable high market value. Different exchanges either decentralised (such as ForkDelta, EtherDelta and IDEX) or centralised (such as Binance, Okex and Huobi.pro), provides a marketplace by alphabetizing them mainly those with high liquidity for public trading. It is evident that the security and transparency of their relative smart contracts is cardinal. In practice, there are de-facto requirements that this contract has to be verified publicly on etherscan.io. Furthermore, it reflects the basic 'code-is-law' trust and spirit of blockchain technology and once these contracts are deployed they should no more be subjects of centralized manipulation and control . A security issue known as Trade trap is reported which violates the requirement above. Trade trap affects a lot of ERC-20 tokens and it has been confirmed that over dozens of these tokens are traded publicly in exchange markets, and it has also been recognised that this plagued tokens are of high profit arbitrage to bad people. This security issue can be exploited as either resulting to unsuitable arbitrage or direct control of buying or selling of affected tokens .(PeckShield:T) These could eventually lead to financial loses for trading customers and this shows lack of adequate security of the exchanges affected when highlighting the tokens for trading. Security threats associated with contracts. This study explains different security bugs that enable malicious users to exploit contracts and earn profits.

### 5.1 Transaction ordering dependence

A block consist a number of transactions therefore a blockchain is updated several times in each span [30]. When two scenarios  $X_o$  and  $X_p$  are considered when a blockchain is at state  $Q$  by invoking the same contract, users have limited knowledge at the state of the contract where personal invocations are executed. Therefore, there is inconsistency between contracts' state where users want to cite at and where the real state of implementation will take place. Users who mines the block are the only that can decide the order of transactions and updates.

#### 5.1.1 Attacks on the contract

It might not be clear to users that having dependence on ordering of transaction is problem to smart contracts. First, even amiable invocations to the contract might yield unanticipated



outcomes to users if there are simultaneous invocations. Second, malicious miners can exploit transaction ordering-dependence contracts to earn more profits or even steal from other users. These scenarios are explained using the figure below.

```

contract Puzzle{
    address public owner;
    bool public locked;
    uint public reward;
    bytes32 public diff;
    bytes public solution;

    function Puzzle(){
        owner =msg.sender;
        reward=msg.value;
        locked=false;
        diff =bytes32(11111);
    }

    function() {
        if (msg.sender==owner){
            if (locked)revert();
            owner.send(reward);
            reward=msg.value;
        }
        else {
            if(msg.data.length>0){
                if(locked) revert();
                if(sha256(msg.data)<diff){
                    msg.sender.send(reward);
                    solution= msg.data;
                    locked=true;
                }
            }
        }
    }
}

```

Figure 40: A contract that rewards user that solve a computation puzzle

Amiable scenario: Two transactions Xy and Xz are considered and they are sent to the puzzle at an approximately same time where Xy comes from the owner of the contract while Xz is from the user who consents a suitable solution to demand for a reward. Since Xy and Xz are transmitted to the puzzle at an approximately same time there is a higher probability of including both transactions in the next block. The order of transaction is used to determine the amount of reward a user is likely to receive from the submitted solution. Users expect to receive rewards basing on the solution submitted but they might receive rewards if Xy is implemented first. The problem is notable when the contract operates as decentralised in the marketplace of exchanges [31]. Sellers update prices frequently in these contracts and users send orders to purchase some products. According to transaction ordering the customers' purchase requests might or might not be successful or even users need to pay more than the marked price while they were making requests.



Malicious scenario: Benign situation might just be accidental since the puzzle owner might not be aware on when to submit the solution. In contrary, evil owners can exploit transaction-ordering dependence to be benefitted financially. In Ethereum the time to determine a new block is approximately twelve seconds. Therefore, malicious owners of the contracts keep confirming the network to find if there is any transaction submitting a solution to the contract and if there is, these owners send transaction Xy to modify the reward and make it smaller to almost zero. Within definite probability, both Xy and Xz are comprised in the new block and Xy is implemented before Xz thus they benefit free solutions to the puzzle. The malicious owners can prejudice the chances of their transactions to be implemented first by involving in the mining directly.

The previous two type of vulnerabilities always come out from developers' negligence so that attackers can seek such code and benefit from it. Whatever the case, to prevent such mistakes, developers can only pay more attention to code. But unavoidably, some people with bad will always try to treat take advantage of any form of vulnerability and exploit it. In smart contract, there are some cases that code as trap.

As research from ERC20, publicly tradable ERC-20 tokens have considerable high market value. Various exchanges, either centralized (e.g., Binance, Huobi.pro, and OKex) or decentralized (e.g., IDEX, EtherDelta, ForkDelta), provide the marketplace by listing them, especially with high-liquidity ones, for public trading. Evidently, the transparency and security of their corresponding smart contracts is paramount. In practice, there is a de-facto requirement for these contract to be publicly verifiable on etherscan.io. Moreover, reflecting the fundamental 'code-is-law' spirit and trust of blockchain technology, these contracts once deployed should not be further subject to centralized control or manipulation." Once smart contracts of publicly tradable ERC-20 tokens are deployed, they should not be further subject to centralized control or manipulation. Unfortunately, tradeTrap plagues 700+ ERC20 tokens and we have so far confirmed at least dozens of them are publicly tradable on current exchanges

The existence of highly manipulatable interfaces (or knobs), however, could be exploited to either make inappropriate arbitrage or even directly control buy / sell prices of affected tokens. All these will eventually result in financial loss for trading customers and essentially reflect lack of enough security of affected exchanges when listing thesae tokens for trading.

In the following, we would like to disclose two types of manipulatable interfaces which could be exploited to achieve unfair arbitrage.

## 5.2 Arbitrary Increase in Token Balance

### 5.2.1 Real world case

Firstly, we would like to expose an interface which can be used to arbitrarily increase the token balance of arbitrary address, completely at the control of contract owner! As shown in Figure 1, a function named *mintToken()* is implemented in these ERC20 smart contracts[32].Contract address:

<https://etherscan.io/address/0x5121e348e897daef1eef23959ab290e5557cf274#code>

```
131 function mintToken(address target, uint256 mintedAmount) onlyOwner {  
132     balanceOf[target] += mintedAmount;  
133     Transfer(0, owner, mintedAmount);  
134     Transfer(owner, target, mintedAmount);  
135 }
```

Figure 41: MinToken function

In general, this function is accessible only by the contract owner, which is used to increase the token supply. However, in spite of a reasonable functionality during the *presale* period, the invocation to this function should be constrained in some way to avoid any abuse. Otherwise, a malicious owner is capable of sending tokens to a specified addresses arbitrarily. These tokens issued with zero-cost could completely destroy the market at the cryptocurrency exchanges. It seems that contracts in part *transferProxy* and *ownerAnyone* also provide function to increase balance in their contract, but the differences between them are that: function to add balance always need check just like `check allocateEndTime < now` in which `allocateEndTime` is already been assignment as `now+1day` in SMT contract while in AURA contract they check `balancesUploaded`. Now including top ones like Binance and OKEx, lot of tradable tokens are affected by this problem. Part of the tokens are tradable with high transaction volume and may affect tens of thousands of trading customers once being exploited.

## 5.3 Manipulatable Prices and Unfair Arbitrage

### 5.3.1 Real World Case

Then, we would like to expose a series of functions that can be directly used to achieve arbitrage. In essence, there exists three manipulatable interfaces, defined in corresponding functions i.e., *setPrices()*, *buy()* and *sell()* (Figure 2). The *setPrices()* function, protected by the *onlyOwner* modifier, merely allows the owner to set the buy / sell prices of the tokens. The *buy()* and *sell()* functions are public accessible and can be used by any trading customers to buy some tokens or sell the tokens they owned. Contract address:

<https://etherscan.io/address/0x2bba3cf6de6058cc1b4457ce00deb359e2703d7f#code>

```
203    /// @notice Allow users to buy tokens for `newBuyPrice` eth and sell tokens for `newSellPrice` eth
204    /// @param newSellPrice Price the users can sell to the contract
205    /// @param newBuyPrice Price users can buy from the contract
206    function setPrices(uint256 newSellPrice, uint256 newBuyPrice) onlyOwner public {
207        sellPrice = newSellPrice;
208        buyPrice = newBuyPrice;
209    }
```

Figure 42: The manipulatable interfaces of affected smart contracts

A keen reader may notice that the aforementioned buy / sell prices seem a little bit weird, as the prices of a tradable token should be determined by the market (i.e., exchanges). Here comes the tricky but interesting part: anyone can be an arbitrageur of these tokens if she wants. Here are simple steps she may want to take:

- One can make a profit by buying tokens with *buyPrice* and then sell them with the market selling price if the latter is greater than the former;
- One can make a profit by buying tokens with the market buying price and then sell them with the *sellPrice* if the latter is greater than the former;

As it is impossible to guarantee that the buy / sell prices are identical to their counterparts (i.e., the market prices) all the time, one can always find an opportunity to make (inappropriate) arbitrage. As a result, the circulation of the tokens would not be maintained anymore. Moreover, as both *buyPrice* and *sellPrice* are modifiable by the owner, the owner becomes the one to directly control the market price[33].

### 5.3.2 Demo

Deploy

1000,"HashCoin ","HSC "

▼

transfer

^

\_to:

"0x692a70d2e424a56d2c6c27aa97d1a86395877b3a"

\_value:

500000

transact

setPrices

newSellPrice:

10

newBuyPrice:

5

Figure 43 :Set up the contract

Following the original contract's transaction, after create the contract, there is balance for this contract, so the owner must call transfer function to allocate balance.

Environment

JavaScript VM

VM (-) ▼

i

Account

0x147...c160c (99.99999999999994882 ▼

📄 🗑

Gas limit

8000000

Value

50

wei ▼

buy

freezeAccount

address target, bool freeze

▼

sell

5

▼

balanceOf

"0x14723a09acff6d2a60dcdf7aa4aff308fddc160c"

▼

0: uint256: 5

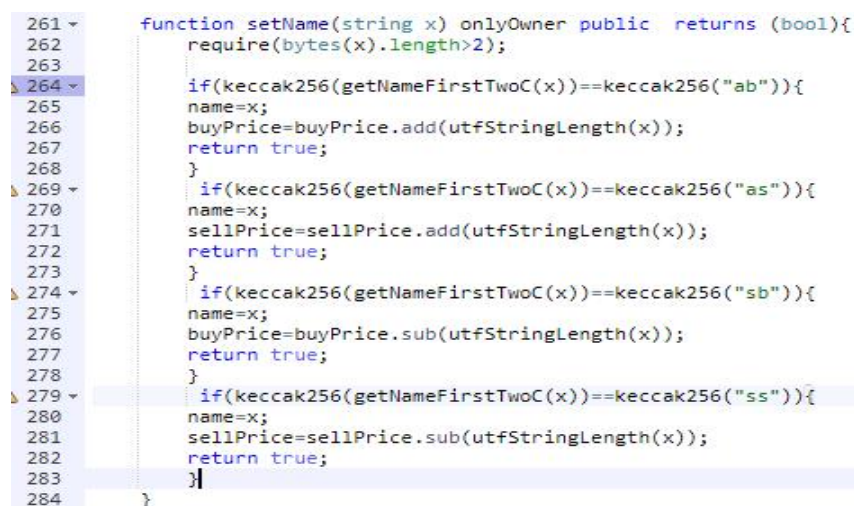
Figure 44 : Use B address to buy tokens then sell them out

Use address B to buy token with 50 wei, get back 10 amount. Then sell 5amount token, it can receive 50 wei back, but still have 5 amount balance on the contract. So this contract spend nothing but just gas gain extra 5 token on this contract.

### 5.3.3 Try to hide the Vulnerability

Because this is some kind of malicious code, we do not need to attack. The owner use setPrices function to cause unbalanced of tokens, which is already attack. From the perspective of contract developer, what should he do is to hide the problem code, avoiding to be discovered. Here try to give some way:

- Function overload: if people just follow others to call the function, it could be possible to set another function with different parameter to hide. But if people check the code or test it with remix-ide, the problem is easy to find out.
- Limit with permission. Use some variable as flag to check before calling some function to modify price. This one is quite not feasible as immediately be found after check transaction.
- Hide code in other function. Here we try to give example:



```
261 function setName(string x) onlyOwner public returns (bool){
262     require(bytes(x).length>2);
263
264     if(keccak256(getNameFirstTwoC(x))==keccak256("ab")){
265         name=x;
266         buyPrice=buyPrice.add(utfStringLength(x));
267         return true;
268     }
269     if(keccak256(getNameFirstTwoC(x))==keccak256("as")){
270         name=x;
271         sellPrice=sellPrice.add(utfStringLength(x));
272         return true;
273     }
274     if(keccak256(getNameFirstTwoC(x))==keccak256("sb")){
275         name=x;
276         buyPrice=buyPrice.sub(utfStringLength(x));
277         return true;
278     }
279     if(keccak256(getNameFirstTwoC(x))==keccak256("ss")){
280         name=x;
281         sellPrice=sellPrice.sub(utfStringLength(x));
282         return true;
283     }
284 }
```

Figure 45: Hide setPrice code in setName function

```

function utfStringLength(string str) constant public
returns (uint256 length)
{
    uint i=0;
    bytes memory string_rep = bytes(str);

    while (i<string_rep.length)
    {
        if (string_rep[i]>>7==0)
            i+=1;
        else if (string_rep[i]>>5==0x6)
            i+=2;
        else if (string_rep[i]>>4==0xE)
            i+=3;
        else if (string_rep[i]>>3==0x1E)
            i+=4;
        else
            //For safety
            i+=1;

        length++;
    }
}

bytes private temp=new bytes(2);
function getNameFirstTwoC(string x) internal returns (string){
    temp[0]=bytes(x)[0];
    temp[1]=bytes(x)[1];
    return string(temp);
}

```

Figure 46: Tools function to use

Nearly no one will check setName function, maybe contract developer can hide in it. Even when someone checks the transaction, they can only see input and output---always name as input and return true or false.

## 5.4 Other contracts with trace vulnerabilities

### 5.4.1 Suicidal contract

This contracts always results to decrease in security by trusted owner due to emergency situations such as being voided by their Ether because of attacks or inappropriate functioning. However, when a contract is eliminated by any arbitrary account which will make it implement suicidal instructions is considered as vulnerable. To eliminate a contract the miner sets two different functions, one to create ownership and the other is to eliminate the actual contract.

### 5.4.2 Greedy contracts

These contracts remain alive and indefinitely lock Ether enabling Ether to be released at zero conditions. These contracts might occur as a result of direct errors. The most habitual such errors normally happen in contracts that accept Ether but completely with no instructions that eliminates Ether or such instructions cannot be reached.

### **5.4.3 Posthumous contracts**

When a contract is eliminated, its code and global variables are emptied from the blockchain and it therefore prevents further execution of the code. In contrary, all eliminated contracts continue receiving transactions even though these contracts can no more invoke the code of the contract, if Ether is send along them and is added the balance of the contract.

## **5.5 Some vulnerabilities not mentioned in detail**

### **5.5.1 Timestamp dependence**

Another security threat experienced by the contract might use the block timestamp as an activation setting to implement some technical performances such as to send money to users. Users have to position the block timestamp basing on the time of their local systems; however, miners can alter the time by approximately 15 minutes while having other users accepting the block . The disparity of block stamps might be less than 15 minutes now basing on the fact that Ethereum obliges nodes within the network to be approximately same to those of local timestamps. Miners normally confirm whether the block timestamp greatly differs from the previous timestamp of the block and it lies between 15 minutes from their local system's timestamp. Thus, the miners are likely to select several timestamps so that they can alter the results of timestamp-dependent contracts. Miners can adjust the timestamp of a block to a distinctive value which manipulates the utility of the timestamp-dependent contract and favour the miners. This attack is illustrated in the theRun where the block number, precursory block and last pay-outs are clearly known. Thus users can alter and select timestamps so that the function random generate results that favour them. As a result, the adversarial might prejudice the result of the random seed to completely any value and hence granting any player they want [34]. But to most developer,it is easy to not use such time stamp in contract unlike remember to do the integer check,there is few past and no recent cases,so do not have detailed analysis here.

### **5.5.2 Race condition**

The following are a different type of race condition inherent to Blockchains: the fact that the order of transactions themselves (within a block) is easily subject to manipulation.Race

condition is also known as time-of-check vs time-of-use (TOCTOU), race condition, transaction ordering dependence (TOD).

Since a transaction is in the mempool for a short while, one can know what actions will occur, before it is included in a block. This can be troublesome for things like decentralized markets, where a transaction to buy some tokens can be seen, and a market order implemented before the other transaction gets included. Protecting against this is difficult, as it would come down to the specific contract itself. For example, in markets, it would be better to implement batch auctions (this also protects against high frequency trading concerns)[35].

Since miners always get rewarded via gas fees for running code on behalf of externally owned addresses (EOA), users can specify higher fees to have their transactions mined more quickly. Every miner want to get higher fee with shorter time or mining less transaction,so always seek and complete transaction with high fee.Because the Ethereum blockchain is always public, allow anyone to check the detail of others' pending transactions including TX information and contract code on many website just like etherscan.io. This means if a given user is revealing the solution to a puzzle or other valuable secret, other user with malicious can copy the solution and initial other transaction with higher fees to race with the original solution,which is most classic case of TOD. If developers of smart contracts forget to check its permission, maybe someone will use it,leading the front-running attacks.

As said,if try to simulate attack,need to build a great node net to reproduce the situation that some miners miner the transaction with higher gas but maybe ignore other transaction with same context but publish earlier. But only use remix-ide and cryptocurrency wallet is impossible to construct such big net. It will get result accidentally with only few nodes and miners. Besides, this is problem cause the mechanism of ethereum but not code. It has no way to analyze with code.



## **6 How was done**

### **6.1 Literature search**

Part of what was done in the method was first of all a thorough search of literature, which focuses on finding out any information about the smart contracts as well as Ethereum and blockchain. The literature search was extensive and detailed in order to get the most relevant and most accurate information, since platforms such as Ethereum are relatively new. The resources used included Google scholar, ScienceDirect, IEEE, DL, Scopus, ACM and others. The search took place over the period of research, and the researcher stayed up to date on any developments on the web articles. Several cases were also identified and used as part of the literature to showcase the existing vulnerabilities.

### **6.2 Taxonomies**

The second part of the method involved researching on taxonomies and building a dataset based on the work proposed by [36]. These taxonomies were collected from different research papers that can be classified as EVM, Solidity, and blockchain. The second part of the taxonomy involved finding a secure website that can be used identity bugs and test for security vulnerabilities. Some of the tools selected included Oyente and Remix, a web based IDE.

### **6.3 Audited smart contracts**

For the testing and identifying security vulnerabilities, the research had to find a secure contract that has been tested and is confirmed to be free of bugs. Thus, the research chose to utilise smart contracts from companies that are reputable and have tested these contracts. The company selected was Zeppelin, because of their record as a reputable company in having clean and debugged smart contracts. The programming language chosen was Solidity as mentioned in the taxonomy part.

### **6.4 Justification of the selected methods**

The choice of utilising literature from online sources as well as taxonomies and smart contracts previous audited in various other researches is critical, because it provided for a way to analyse security vulnerabilities with the best available resources.

Time and cost are such resources that are highly important and never nearly adequate in a research. The use of previous researches that are current and up to date on vulnerabilities in smart contracts are a less costly way to conduct the project research as well as save on time. while it required a significant period of observation and gathering of information for several months, it was less time consuming and more deliverable within the expected time limit of the project. It also saved costs because the internet is highly affordable. Personalised access to some of the tools like SmartCheck, Oyente, Remix, and others would have been very costly, since some of them require a membership and a fee to be able to utilise the services, which could be too costly over the period of the project.

Another reason for the choice of audited smart contracts as well as taxonomies that have been verified in previous work was to ensure total security and mitigation of bugs that might come with unchecked smart contracts. As mentioned, there is a real and ongoing threat of vulnerabilities in online platforms infringing the privacy and security of a person, and some of these could be passed on through unsecure smart securities.

The choice of literature and previous research allows for a wide degree of data to be collected and analysed for proper evaluation, this wide degree of data can be crucial to identifying all the important areas that needed to be research, it means that data and information was not limited to one source or aspect, but rather explored in detail through different approaches and ideas presented in other projects.

## 7 Result and Evaluation

This project focuses on an exploration of smart contracts, by analysing their vulnerabilities. Essentially, smart contracts are computerized transaction protocols that can be used to execute the terms of contract. Smart contracts are supported on two existing platforms of cryptography known as blockchain and Ethereum. However, in a world of growing insecurities online, smart contracts are vulnerable in many ways, and can be exploited. This project has contributed new information to the field of technology, specifically in cryptography, which heavily relies on smart contracts, which as mentioned can be exploited, through evaluating and analysing all the existing forms of vulnerabilities, and how to possibly prevent them from occurring.

The project has utilised methods like literature search, case study, and analysis of taxonomies and audited smart contracts from previous studies. Some of the reasons for selecting such processes was due to time, cost, availability of information, and security from possible bugs if smart contracts selected were not audited.

The current rate of technology development has resulted into more applications on the internet through computers. The regular innovations being introduced into the world have had significant impacts in pushing the limits of software and systems today. However, with the development of such technologies and applications to make business and learning easier, there have been numerous setbacks as well, one of which is the security issues that arise. Thus this study analysed and focused on blockchains and Ethereum platforms to further understand the vulnerabilities.

A blockchain is a ledger which is shared and it is used to keep track all the transactions that have taken place and all the data that has been reciprocated from the time it was created. It basically keeps a continuous emerging list of catalogues known as blocks. Ethereum serves as both a payment system and a computing platform. Ethereum is termed as the global computer and as the future internet which is capacitated by the blockchain technology.

Ethereum bugs cause integers overflow and change of ownership of the contracts which result in halting of ERC-20 token exchanges and denial of service respectively. These bugs attack relevant information to the owner accessing their account information, changing account balances and bypassing intended restrictions to manipulate digital assets. Peckshield, a smart contracts security company has been able to identify the vulnerabilities that smart contracts are prone to when incorrect codes are written. The case studies have not yet identified effective remedies although experts have suggested performing auditing and formal

verification to ensure that security holes in contracts are eliminated. Smart contracts functionality should be considered critical to the success of the market since the presence of bugs lead to significant hits to the value of tokens. Developers should be more rigorous on how to execute codes by avoiding functions that enable attackers to bypass security and sanity checks.

Some of the vulnerabilities identified included transaction ordering dependence, attacks on the contract, timestamp dependence, botched exceptions, reentrancy vulnerability, greedy contracts, suicidal contracts, posthumous contracts, and others. These vulnerabilities level can also be classified as low, medium, and high, depending on how they affect a secure system.

Some of the mitigations include making smart security checks to a contract before executing it. It is essential for Solidity to integrated overflow-protected arithmetic types that can assist systems to detect and curb with integer related vulnerabilities. However, Smith has argued that it is impossible for humans to administer codes that are free from security holes completely and as such they should ensure that the codes are fully audited before deployment to minimize the risks of vulnerabilities[37].

## **8 Conclusion**

### **8.1 Limitations of the project**

The project faced challenges such as difficulties in acquiring materials to obtain comprehensive details on the subject at hand leading to a time consuming and energy intensive task. The study had to address several case studies that were crucial to the research which were not widely accessible, hence limiting the amount of information that was needed. The project was also not able to access security analysis tools such as SmartCheck, Remix, and Oyente directly due to incremental costs. The project also had to utilize examples of previous audited smart contracts in order to avoid the risk of bugs without the right security analysis tools, and thus did not analyse any of their own smart contracts.

### **8.2 Future Work**

Despite the effort that has been put into this project, however, as said above, there are some limitations of the system that were not implemented due to time restriction. The future work can improve and prepare for work: analyse some model or template of vulnerabilities; do some research how to get data from blockchain by some website like Etherscan to monitor transaction with abnormal data; try to design plugin to detect vulnerabilities from a contract. What's more, it is good to take blockchain security as employment direction, learning on go language is necessary. In general, keep study on go and solidity, then try to seek intern job in the future.

## 9 References

- [1] Ahire, J, *Blockchain: the future?*. Lulu. Com, 2018.
- [2] Tsankov, P, Dan, A., Cohen, D.D., Gervais, A., Buenzli, F. and Vechev, M. Securify: Practical Security Analysis of Smart Contracts. *arXiv preprint arXiv:1806.01143*,2018.
- [3] Dannen, C, Use Cases. In *Introducing Ethereum and Solidity* (pp. 165-172). Apress, Berkeley, CA, 2017.
- [4] Christidis, K. and Devetsikiotis, M. Blockchains and smart contracts for the internet of things. *Ieee Access*, 4, pp.2292-2303,2016
- [5] Muntean, P. and Grossklags, J, Practical Integer Overflow Prevention. *arXiv preprint arXiv:1710.03720*,2017.
- [6] Bashir, I, *Mastering Blockchain*. Packt Publishing Ltd, 2017.
- [7] Fontein, R, Comparison of static analysis tooling for smart contracts on the EVM,2018.
- [8] Mougayar, W *The business blockchain: promise, practice, and application of the next Internet technology*. John Wiley & Sons,, 2016.
- [9] Alharby, M. and van Moorsel, A, Blockchain-based smart contracts: A systematic mapping study. *arXiv preprint arXiv:1710.06372*, 2017.
- [10] Zheng, Z., Xie, S, Dai, H.N. and Wang, H, Blockchain challenges and opportunities: A survey. *Work Pap.–2016*, 2016.
- [11] Li, X., Jiang, P, Chen, T., Luo, X. and Wen, Q, A survey on the security of blockchain systems. *Future Generation Computer Systems*, 2017.
- [12] Szabo, N, Formalizing and securing relationships on public networks. *First Monday*, 1997.
- [13] Buterin, V.A next-generation smart contract and decentralized application platform. *white paper*, 2014.
- [14] Wood, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151, pp.1-32, 2014.
- [15] Bartoletti, M, Carta, S, Cimoli, T. and Saia, R. Dissecting Ponzi schemes on Ethereum: identification, analysis, and impact. *arXiv preprint arXiv:1703.03779*, 2017.
- [16] Gerard, D *Attack of the 50 foot blockchain: Bitcoin, blockchain, Ethereum & smart contracts*. David Gerard.', 2017.

- [17] Iyer, K. and Dannen, C, Building Games with Ethereum Smart Contracts: Intermediate , 2018.
- [19] Iyer, K. and Dannen, C, Building Games with Ethereum Smart Contracts: Intermediate Projects for Solidity Developers, 2018.
- [20] Wang, T., Wei, T., Lin, Z. and Zou, W, February. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution. In *NDSS.*’ , 2009.
- [23] Bryans, D, Bitcoin and money laundering: mining for an effective solution. *Ind. LJ*, 89, p.441, 2014.
- [24] Bennett. S, *Bitcoin & Blockchain: 2 Manuscripts - Understanding Bitcoin & Understanding*. Cryptomasher via PublishDrive, 2017.
- [25] Hofmann, E., Strewe, U.M. and Bosia, N, *Supply Chain Finance and Blockchain Technology: The Case of Reverse Securitisation*. Springer, 2017.
- [30] Miller, A., Juels, A., Shi, E., Parno, B. and Katz, J, May. Permacoin: Repurposing bitcoin work for data preservation. In *2014 IEEE Symposium on Security and Privacy (SP)* (pp. 475-490). IEEE, 2014.
- [31] Luu, L., Chu, D.H., Olickel, H., Saxena, P. and Hobor, A, October. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (pp. 254-269). ACM, 2016.
- [36] Atzei, N., Bartoletti, M., & Cimoli, T, A survey of attacks on ethereum smart contracts (sok). In *International Conference on Principles of Security and Trust*, 164–186. Springer ,2017
- [37] Smith, S.W, Humans in the loop: Human-computer interaction and security. *IEEE Security & Privacy*, 99(3), pp.75-79, 2003.

URLs:

- [18] A Medium Corporation, [https://medium.com/@Jon\\_A\\_Haas/quantstamps-assessment-of-the-recent-batchoverflow-and-proxyoverflow-vulnerabilities-1f9562fa7340](https://medium.com/@Jon_A_Haas/quantstamps-assessment-of-the-recent-batchoverflow-and-proxyoverflow-vulnerabilities-1f9562fa7340), 2018.
- [21] Implementing an attack against the ERC20 digital currency ProxyOverflow vulnerability on a private Ethereum  
<http://www.91ri.org/17734.html/>, May 2018.

- [22] New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts  
<https://peckshield.com/2018/05/18/burnOverflow/>, May 2018.
- [26] Blockchain Validation Protocol  
<https://chain.com/docs/1.2/protocol/specifications/validation>, June 2018.
- [27] BitDegree Solidity Docs  
<https://www.bitdegree.org/learn/solidity-function-modifiers/>, May 201
- [28] Peckshield New ceoAnyone Bug Identified in Multiple Crypto Game Smart Contracts.  
<https://www.peckshield.com/2018/05/21/ceoAnyone/>, May 2018
- [29] Peckshield New ownerAnyone Bug Allows For Anyone to "Own" Certain  
<https://www.peckshield.com/2018/05/03/ownerAnyone/>, May 2018
- [32] PeckShield: Highly-Manipulatable ERC20 Tokens Identified in Multiple Top Exchanges (including Binance, Huobi, and OKex)  
<https://www.peckshield.com/2018/06/11/tradeTrap/>, June 2018.
- [33] Zhihu/DappReview Analysis "redefined" the hash world of the blockchain  
<https://zhuanlan.zhihu.com/p/37305444>, May 2018
- [34] Bill Gleim Recommendations for Smart Contract Security in Solidity  
<https://consensys.github.io/smart-contract-best-practices/recommendations/#timestamp-dependence>, March 2017
- [35] David Wong, Mason Hemmel. DASP TOP 10.  
<https://dasp.co/#item-7>, April 2018.