Johns Hopkins University

Blockchain and Cryptocurrency

Final Project Report

Security Analysis for IOTA, Beauty Chain, and SmartMesh

Author:
Zhiyuan Li, Hong Ma, Miao Zhang, Chenfeng Nie

1 Introduction

If you are the one who enjoy reading everyday news, in the technology section, you always see some news about the concerns of currently hottest technology—blockchain and cryptocurrency, and you may always see some news about there are some security vulnerabilities lie inside some cryptocurrencies discovered by researchers. Based on the concerns and worrisome about cryptocurrency and blockchain, in this project, our group are trying to analyze some discovered vulnerabilities. The motivations for us doing the security analysis for the cryptocurrencies are that we are willing to analyze and address some security problems inside the blockchain and cryptocurrency technology and that meanwhile understanding this new technology better.

2 Security Analysis for IOTA

2.1 Problem Statement

Vulnerabilities in the Curl Hash Function and IOTA Signature Scheme: The IOTA team developed their own cryptographic hash function called Curl. It is applied to a number of purposes in IOTA including transaction address creation, message digest creation, Proof-of-Work and hash-based signatures. However, Curl cannot withstand the well-known differential cryptanalysis which can successfully generate a practical hash collision of Curl. Because of the failure of collision-resistance, it is easy to forge a signature on IOTA payments as well.

Fail to Achieve Decentralization and Potential risks of Coordinator: Centralization is an inherent property since IOTA came out. IOTA currently relies on a trusted party called Coordinator to approve transactions. The coordinator makes decisions, where the tangle needs to grow and to coordinate the next steps. It also marks transactions that are already confirmed. That's the reason why IOTA is not decentralized.

2.2 Motivation

As a vanguard to go beyond blockchain to get rid of transaction fee and resolve scalability, IOTA is a revolutionary new, next-generation public distributed ledger that utilizes a novel invention. Additionally, it devotes to IoT's lightweight requirements and focuses on tiny transaction scenario comparing to conventional cryptocurrency. Out of these considerations, I believe that IOTA will outperform traditional blockchain.

Right now, the implementation of IOTA is flawed. However, there is definitely a solution to solve the problems of IOTA. If all is fixed, IOTA is still one of the most promising cryptocurrencies. Thus, I hope all the analysis and research on its vulnerabilities can finally benefit IOTA. As a potential next-generation cryptocurrency, I will spare no expense to help IOTA be better and more tailored for IoT industry.

2.3 Existing Solutions

The existing solutions to tackle with the vulnerability of Curl is to stop relying on Curl for cryptography purposes and instead using those well-studied cryptography hash functions like $MD6$, $SHA3$ and $SHA256$.

The shortcomings lie in the innate ternary-based system design of IOTA while those well-studied cryptography hash functions are binary-oriented. Curl is a function which is fast, energy efficient and tiny codebase-wise for IOTA operation. That is why and how Curl is invented and tailored for IoT. This suggestion is not compatible with the intention and will cause a top-down reconstruction of the whole IOTA project. Thus, it is not reasonable.

The existing solution to solve the issues of Coordinator is replacing it with a random walk Monte Carlo method. It is a random algorithm for tips to decide on which transactions to approve. It provides a more decentralized environment for IOTA.

The random walk Monte Carlo method is not thoroughly studied. No one can ensure its resistance to attacks like a double-spend attack.

2.4 Our Observations

Observation of Curl and Signature Scheme: Curl include a message, breaks it into blocks and then iteratively copies each message block into a current state. Then Transform function is called to transform and update the state with "sbox".

```
Curl(message)
    # the state consists of 729 trits. It is initalized to all zero.
    state = [0]*729
    # The message is broken into message blocks of size 243
    MB_0, MB_1, ... MB_n = split(message)
    for MB_i in MB_0, MB_1, ... MB_n:
    # The current message block is copyed into the first 243 trits of the state
      state[0:243] = MB_i
      state = Transform(state)

    # The output is the first 243 trits of the state
    output = state[0:243]
    return output

Transform(state)
    for round in 27
        i = 0
        new_state = [0]*729
        for pos in 729
            i = j
            j += (364 if j < 365 else -365)
            x = state[i]; y = state[j]
            z = sbox[x, y]
            new_state[pos] = z
        state = new_state
return new_state

sbox:
    y:  -1, 0, 1
    x: -1 [1, 1, -1]
    x:  0 [0, -1, 1]
    x:  1 [-1, 0, 0]
```

However, Curl is vulnerable to a well-known crypto analysis technique called differential cryptanalysis. Following the analysis, we can generate hash collisions in Curl. These collisions appear in all rounds of Curl and can be generated in seconds on commodity hardware. Here are three examples of hash collisions.

```
from iota import TryteString
from iota.crypto.pycurl import Curl
from random import randint
```

```

def H(msg):
   out = []
   sponge = Curl()
   sponge.absorb(TryteString(msg).as_trits())
   sponge.squeeze(out)
   return TryteString.from_trits(out)

def simplecol():
      msg1 =
          b'999999999999999999999999999999999999999999999999999999999999999999999999999999999'
   msg2 =
      b'9999999999999999999999999999999999999999999999999999999999999999999999999999999999999'
   hash1 = H(msg1)
   hash2 = H(msg2)
   print hash1
   print hash2
   print hash1 == hash2, msg1 == msg2
// True False

def paddingshortcol():
   msg1 = b'RSWWSFXPQJUBJROQBRQZWZXZJWMUBVIVMHPPTYSNW9YQIQQF9RCSJJCVZG9ZWITXNCSBBDHEEKDRBHV'
   msg2 = b'RSWWSFXPQJUBJROQBRQZWZXZJWMUBVIVMHPPTYSNW9YQIQQF9RCSJJCVZG9ZWITXNCSBBDHEEKDRBHV9'
   msg3 =
      b'RSWWSFXPQJUBJROQBRQZWZXZJWMUBVIVMHPPTYSNW9YQIQQF9RCSJJCVZG9ZWITXNCSBBDHEEKDRBHV99'
   hash1 = H(msg1)
   hash2 = H(msg2)
   hash3 = H(msg3)
   print hash1
   print hash2
   print hash3
   print hash1 == hash2 == hash3, msg1 == msg2, msg2 == msg3
// True False False

def shiftedmsgs():
   msg1 =
      b'9999KENNYLOGGINS99999999999999999999999999999999999999999999999999999999999999999'
   msg2 =
      b'9KENNYLOGGINS99999999999999999999999999999999999999999999999999999999999999999999'
   hash1 = H(msg1)
   hash2 = H(msg2)
   print "".join(["9"]*81)
   print hash1
   print hash2
//OBIVYFRBFRPRVNYMJFUGYYYFHREPKJWGLWCUSKDBQVOZSFYOPPOYGFTPQQA9ZMSYE99CMCYEELGEFRDTU

//VYFRBFRPRVNYMJFUGYYYFHREPKJWGLWCUSKDBQVOZSFYOPPOYGFTPQQA9ZMSYE99CMCYEELGEFRDTUXMW
```

The first function is an all-zero collision that different lengths of all zero messages will hold the same Curl value.

The second function shows a collision that the same message padded with different numbers of zero messages will hold the same Curl value.

The third function reveals that Curl is not satisfied with pre-image resistance because two messages which are bit rotations of each other while Curled results are also bit rotations of each other.

Seeing the vulnerabilities in Curl hash function of IOTA, we can exploit these collisions to break IOTA's Signature Scheme. The signature scheme of IOTA is based on Winternitz One-Time Signatures (WOTS) with an important difference.

```
IOTA_Sign(SK, msg):
    h_msg = CURL_Hash(msg)
    sig = WOTS_Sign(SK, h_msg)
    return sig
```

As we can see in the code, IOTA's signature scheme operates on the Curl hash value of message rather than the original message as WOTS does. This tiny difference leads to a critical impact that WOTS only requires the hash function to be a one-way function while IOTA requires the hash function to be collision resistant.

Thus, if two different messages, m1 and m2, hash to the same output, a signature on m1 will also verify as a signature on m2. Since we are able to find a collision in Curl, it means a forgeability in IOTA's signature scheme.

Here is an example of stealing money attack due to the collision in Curl: I can use the Curl collision to generate two valid IOTA bundles which collide twice on different trits. First, Bob requests a signature of Alice to sign a bundle, bundle1, which pays Bob 100 IOTA from Alice's funds. Bob can use the signature on bundle1 as a valid signature on bundle2 which pays Bob 129140263 IOTA from Alice's funds. Alice wants to pay Bob:

1. Bob creates two specially constructed colliding bundles; bundle1 and bundle2. Bundle1 spends 939211930 IOTA from Alice's address and pays 100 IOTA to Bob with 939211830 IOTA in change back to Alice. Bundle2 also spends some of Alice's funds but differs in that it pays 129140263 IOTA to Bob and only 810071667 IOTA change back to Alice.

2. Bob asks Alice to pay Bob by signing bundle1, that is paying 100 to Bob and send the change 939211830 back to Alice.

3. Bob takes the signature off of bundle1 and uses it on bundle2 which she then broadcasts to the network.

4. When Bundle2 is confirmed, Bob will have stolen 129140163 IOTA from Alice.

This is a serious attack as a signature attesting that an attacker can steal the balance in victim's account without the victim ever agreeing to pay. Therefore, signatures using Curl can no longer be trusted to authorize transactions.

Observation on Coordinator: As the definition, Coordinator is used to prevent double-spend attacks. However, Coordinator itself is open to double-spend attack. Since the source code of Coordinator is closed to the public, basically we do not know how it performs. Although the co-founder of IOTA explains that the conspiracy of Coordinator is not rational, it is still open to question.

Meanwhile, Coordinator is in charge of the approval and confirmation of transactions. The existence of Coordinator greatly reduces the Decentralization of IOTA which is a significant property of cryptocurrency.

2.5 Our Solutions

A golden rule of encryption says , don't roll your own crypto. In order to achieve the similar goal of Curl with least risks, I suggest an indirect way to include both well-studied hash function SHA-3 and Curl. In the Proof-of-Work and signature process, which hash function is needed, the trinary function Curl can be applied to a middle calculation process of SHA-3 hash calculation such as:

$$Hash(m) = SHA3(m||Curl(m))$$

In this way, the weakness of Curl hash function can be transferred and elevated to the level of SHA3 which is collision-resistant. Thus making the scheme at least as secure as SHA3.

The existence of Coordinator is a crucial property in IOTA. Furthermore, even if Coordinator is doomed to be optional for IOTA one day, we cannot currently provide a comprehensive algorithm for transaction selection process. Monte Carlo method is not sufficiently tested by IOTA therefore decentralization of IOTA is nothing more than just ambiguous theory. For the consideration of tangle security, Coordinator is irreplaceable in IOTA right now.

3 Security Analysis for Beauty Chain (BEC)

3.1 Problem Statement

This is a recently discovered security problem in a cryptocurrency, the Beauty Chain (BEC), which is a comparatively new cryptocurrency founded in China. Based on its website and its issued whitepaper, its creator set the goal to be “a truly decentralized and beauty-themed ecosystem”. It started trading on OKEx on Feb. 23, 2018 and spiked 4000 percent on the first trading day. From its peak market cap of around 70 billion USD, it has gradually come down to around two billion USD as of April 22.

However, at the April 22, its trading value suddenly dropped to zero, because of an implementation vulnerability. The vulnerability was exploited by the hacker–

57896044618658100000000000000000000000000000000000000000.792003956564819968

BEC were transferred to the hacker’s addresses. And then, it was discovered by researchers that there are multiple Ethereum-based (ERC-20) token contracts are susceptible to this similar problem. This vulnerability can be categorized as an integer overflow vulnerability.

3.2 Motivation

If you are the one who enjoy reading everyday news, in the technology section, you always see some news about the concerns of currently hottest technology—blockchain and cryptocurrency, and you may always see some news about there are some security vulnerabilities lie inside some cryptocurrencies discovered by researchers. Based on the concerns and worrisome about cryptocurrency and blockchain, in this project, our group are trying to analyze some discovered vulnerabilities. The motivations for us doing the security analysis for the cryptocurrencies are that we are willing to analyze and address some security problems inside the blockchain and cryptocurrency technology and that meanwhile understanding this new technology better. In specific, we are trying to dive deep into the BEC integer overflow vulnerability and based on the vulnerability the cryptocurrency developers should be more careful and should pay more attention to the implementation.

3.3 Existing Solutions

Actually, integer overflow vulnerability is existing for a very long time. There are many effective solutions proposed by researchers and there are some famous libraries the developers can use to prevent these kinds of vulnerabilities. In Solidity language, $SafeMath$ would be a reliable library that can be used in this case.

3.4 Our Observation

So how did this hacking happen? It turns out there is a buffer overflow vulnerability in the implementation code, specifically, it is an integer overflow vulnerability. When we review the implementation code of Beauty Chain, we find that there is a method called $batchTransfer()$, which is the vulnerable function the hacker taking advantage of. The function is designed for users to transfer token, in this case the BEC, to multiple addresses at once.
Based on the $batchTransfer()$ function:

```
    batchTransfer(address[] _receivers, uint256 _value);
```

Argument *_receivers*: an array contains the addresses that you want to transfer the token at once.
Argument *_value*: an unsigned 256 bits integer contains the value of token you want to transfer to each address.

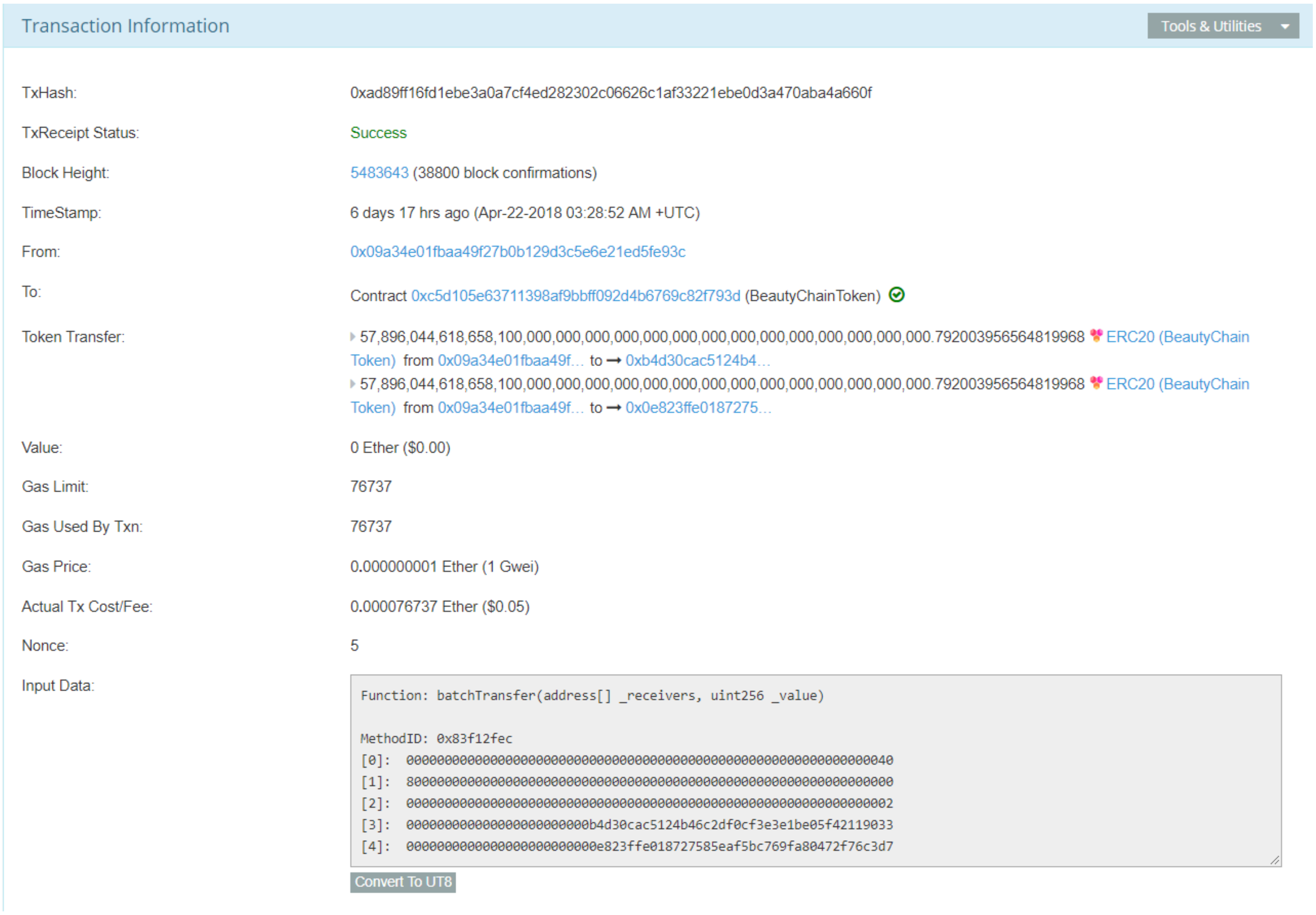
The logic for this function is that when the user specifies the receivers and value, it will check the total token of value the user sent, that is: $amount = numberofreceivers * value$. And then, it will check the user's balance to see if this transfer transaction should be verified, if the user's balance is more than the total value of token he or she wants to send, this transaction is valid. And then, it will subtract the user's balance and add tokens to the receivers. This logic is correct, but when it comes to implementing, the developers make a lethal mistake. The source code to implement the $batchTransfer$ is that:

```
    function batchTransfer(address[] _receivers, uint256 _value) public whenNotPaused
     returns (bool){
      uint cnt = _receivers.lenght;
      uint256 amount = uint256(cnt) * _value;
      require(cnt > 0 && cnt <= 20);
      require(_value > 0 && balances[msg.sender] >= amount);

      balances[msg.sender] = balances[msg.sneder].sub(amount);
      for(uint i = 0; i < cnt; i++){
        balances[_receivers[i]] = balances[_receivers[i]].add(_value);
        Transfer(msg.sender, _receivers[i], _value);
      }
      return true;
    }
```

Based on the source code, The *cnt* here is the number of recipients to transfer the tokens to, the *_value* is the number of BECs each recipient should receive, and the *amount* is the computed amount the contract should withdraw from the sender's account for the transfer.

We can see that variable amount is a 256-bit unsigned integer, so it is ranging from 0 to $2^{256} - 1$ in terms of computer storage. What if the *amount* is a very large integer, which is larger than $2^{256} - 1$, what would happen? It would result in the very infamous vulnerability, called integer overflow vulnerability. If someone can manipulate the variable *amount* and set it to be 2^{256}, it would cause the variable *amount* to be zero. Now you can transfer any number of BECs to anyone without withdrawing anything from the sender's account. So a sender with 1 BEC can now send trillions (in fact, vigintillions) of BECs to other recipients.



Transaction Information

Tools & Utilities

TxHash:	0xad89ff16fd1ebe3a0a7cf4ed282302c06626c1af33221ebe0d3a470aba4a660f
TxReceipt Status:	Success
Block Height:	5483643 (38800 block confirmations)
TimeStamp:	6 days 17 hrs ago (Apr-22-2018 03:28:52 AM +UTC)
From:	0x09a34e01fbaa49f27b0b129d3c5e6e21ed5fe93c
To:	Contract 0xc5d105e63711398af9bbff092d4b6769c82f793d (BeautyChainToken)
Token Transfer:	57,896,044,618,658,100,000,000,000,000,000,000,000,000,000,000,000,000,000,000.792003956564819968 ERC20 (BeautyChain Token) from 0x09a34e01fbaa49f... to ➞ 0xb4d30cac5124b4... 57,896,044,618,658,100,000,000,000,000,000,000,000,000,000,000,000,000,000,000.792003956564819968 ERC20 (BeautyChain Token) from 0x09a34e01fbaa49f... to ➞ 0x0e823ffe0187275...
Value:	0 Ether ($0.00)
Gas Limit:	76737
Gas Used By Txn:	76737
Gas Price:	0.000000001 Ether (1 Gwei)
Actual Tx Cost/Fee:	0.000076737 Ether ($0.05)
Nonce:	5
Input Data:	

```
Function: batchTransfer(address[] _receivers, uint256 _value)

MethodID: 0x83f12fec
[0]:  0000000000000000000000000000000000000000000000000000000000000040
[1]:  8000000000000000000000000000000000000000000000000000000000000000
[2]:  0000000000000000000000000000000000000000000000000000000000000002
[3]:  000000000000000000000000b4d30cac5124b46c2df0cf3e3e1be05f42119033
[4]:  000000000000000000000000e823ffe018727585eaf5bc769fa80472f76c3d7
```

Figure 1: The hacker exploiting transaction.

Let's take a deeper look at this transaction. Based on the transaction website, this transaction is shown as Figure 1. We can see that the hacker transfer the tokens to two addresses:

$$0x0e823ffe018727585eaf5bc769fa80472f76c3d7$$

and

$$0xb4d30cac5124b46c2df0cf3e3e1be05f42119033$$

And the hacker set the parameter $\_value$ to be

$$8000000000000000000000000000000000000000000000000000000000000000$$

, which is a hexadecimal number. Converting it to a decimal number, it would be:

$$57896044618658097711785492504343953926634992332820282019728792003956564819968$$

Since the number of addresses, in this case, is 2, so we have

$$amount = \_value * 2$$

So the *amount* would be

$$115792089237316195423570985008687907853269984665640564039457584007913129639936$$

That it is the value of 2^{256} and exceed the $uint256$ maximum integer value by 1. So the *amount* now would be overflowed and set to be 0, and the hacker would transfer a ton of tokens to the addresses he or she specified without losing a simple token.

Ironically, as you can see from the code, the developers of Beauty Chain did use *SafeMath*, except in that one instance! Why the developers leave only that line of code wide open to integer overflow? No one would know except those developers themselves. In term of the reason why the developers of the Beauty Chain use *SafeMath* in other line of codes except that one vulnerable line of code, we guess that maybe the creator of Beauty Chain was in the gold rush of ICOs, so the developers might just copy and past some *ERC*20 contract code from the Internet without fully understanding their implications. Therefore, it leads us to think about the fact that blockchains and cryptocurrencies are all the rage in the current information technology industry, even beyond the information technology industry. It seems like that creating a new cryptocurrency and then put it on the market in a rush would easily get a ton of fundings from the investors so that the creators could make a big fortune overnight. Since cryptocurrency and blockchain are so hot right now, some new cryptocurrencies are invented just for making some quick money, not for the innovation of new technology. If making quick money is the primary goal, security would definitely not be guaranteed since security needs patience to testing and reviewing the implementations.

3.5 Our Solutions

Our solutions would be in accordance with the existing solutions. When there is a vulnerability that is exploited by the hackers, the compromised cryptocurrency company needs to works with the exchange, for example, *OKEx*, to roll back to the status before the hacking activity happening. This is far not enough to solve the problem from the bottom line, the key solution is that the best security and software development process practices should apply through the whole development cycle. Using *SafeMath* is a small but very important way to secure the implementation from integer overflow vulnerability. We can take a look at the implementation of *SafeMath* to see how it can help.

```
    pragma solidity ^0.4.21;
/**
 * @title SafeMath
 * @dev Math operations with safety checks that throw on error
 */
library SafeMath {

  /**
  * @dev Multiplies two numbers, throws on overflow.
  */
  function mul(uint256 a, uint256 b) internal pure returns (uint256 c) {
    if (a == 0) {
      return 0;
    }
    c = a * b;
    assert(c / a == b);
    return c;
  }

  /**
  * @dev Integer division of two numbers, truncating the quotient.
  */
  function div(uint256 a, uint256 b) internal pure 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 a / b;
  }

  /**
  * @dev Subtracts two numbers, throws on overflow (i.e. if subtrahend is greater than
     minuend).
  */
  function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    assert(b <= a);
    return a - b;
  }

  /**
  * @dev Adds two numbers, throws on overflow.
  */
  function add(uint256 a, uint256 b) internal pure returns (uint256 c) {
    c = a + b;
    assert(c >= a);
    return c;
  }
}
```

So the bottom line is that the developers should keep in mind these potential problems when programming. Also, conducting the code reviewing and testing are necessary before the ICO.

Other crucial things we need to take into consideration are that we need to carefully think about the bubble in the blockchains and cryptocurrencies. What we really want to have, is a new generation of smart contract or a program with a ton of vulnerabilities?

4 Security Analysis for SmartMesh (SMT)

4.1 Problem Statement

Recently, it is reported that multiple ERC-20 smart contracts have vital vulnerabilities, which can cause huge damage and loss. For example, in the early morning of April 25th, the SmartMesh Team discovered an "Ethereum smart contract overflow vulnerability" and immediately contacted the major exchanges where SMT is listed such as Huobi and OKEX. They have suspended SMT trades and transfers since then.
The total number of "counterfeit tokens" generated by the ETH smart contract vulnerability:

$$65133050195990400000000000000000000000000000000.891004451135422463$$

The attacker traded a part of these "counterfeit tokens" on some exchanges. Then these abnormal trades have attracted the exchanges' attention and the counterfeit untraded tokens have been completely frozen by the exchanges.

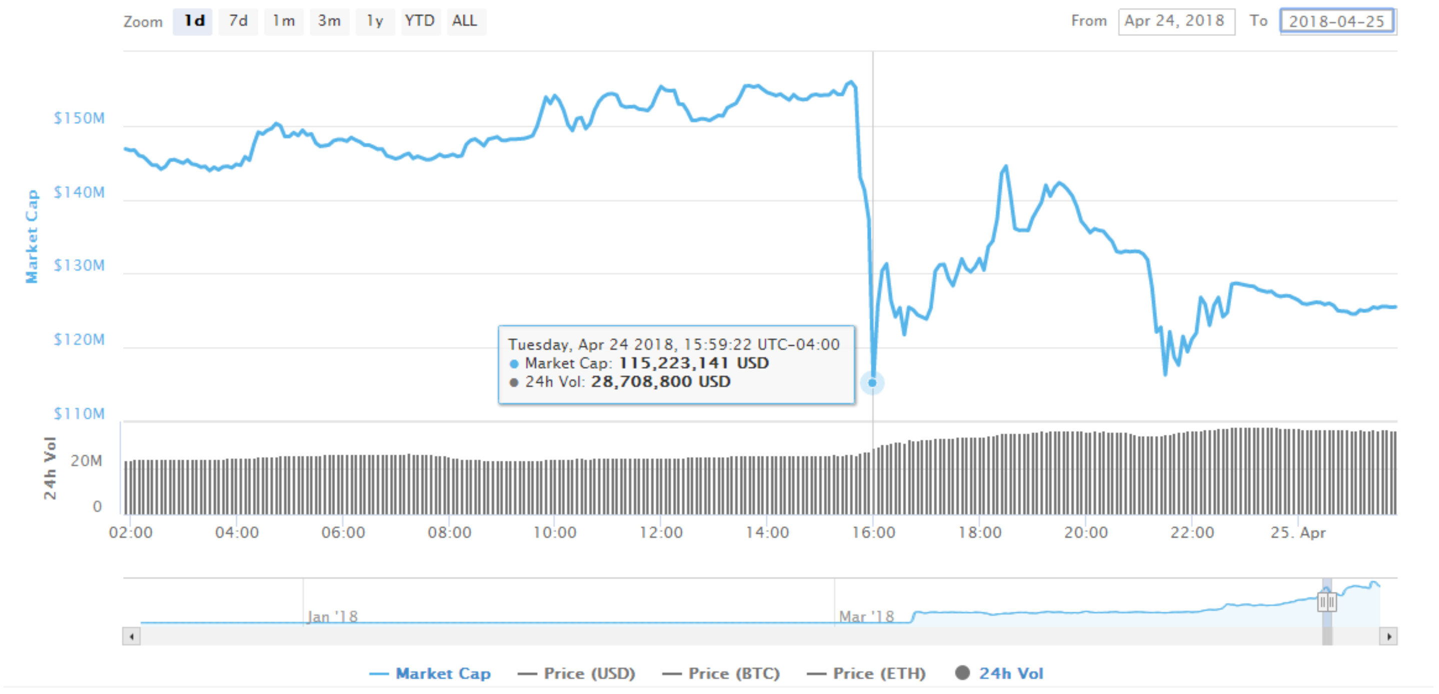
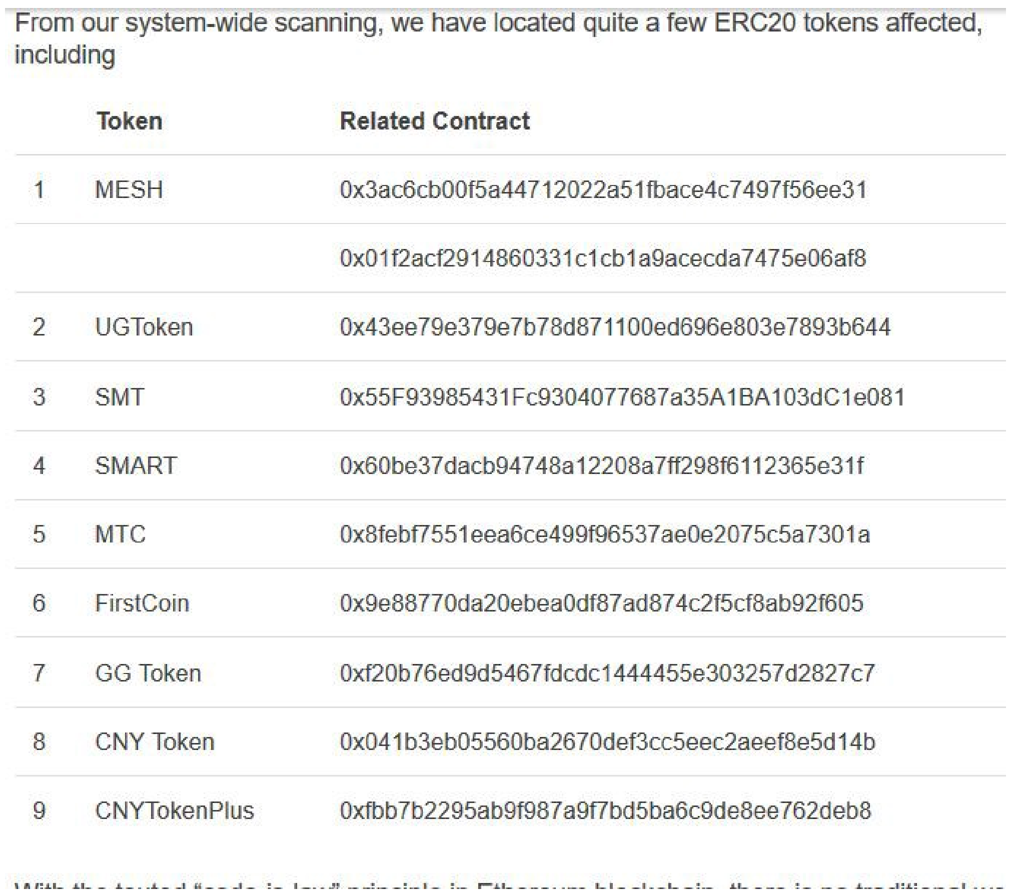


Figure 2: The loss of SMT.

Reopening exchange trade: After this event, the SmartMesh team has already communicated and coordinated with several exchanges including Huobi and OKEX and conducted in-depth communications with them regarding the specific timing for re-opening SMT transactions and other related measures. The team initially has reached an agreement with a number of exchanges, and will jointly work together to resolve losses caused by this loophole, and make every effort to maintain the interests of the community and the value of SMT.

Not all ERC-20 tokens have been affected by this exploit. Peckshield did however trace other tokens that have been affected by this vulnerability.



From our system-wide scanning, we have located quite a few ERC20 tokens affected, including

	Token	Related Contract
1	MESH	0x3ac6cb00f5a44712022a51fbace4c7497f56ee31
		0x01f2acf2914860331c1cb1a9acecda7475e06af8
2	UGToken	0x43ee79e379e7b78d871100ed696e803e7893b644
3	SMT	0x55F93985431Fc9304077687a35A1BA103dC1e081
4	SMART	0x60be37dacb94748a12208a7ff298f6112365e31f
5	MTC	0x8febf7551eea6ce499f96537ae0e2075c5a7301a
6	FirstCoin	0x9e88770da20ebea0df87ad874c2f5cf8ab92f605
7	GG Token	0xf20b76ed9d5467fdcdc1444455e303257d2827c7
8	CNY Token	0x041b3eb05560ba2670def3cc5eec2aeef8e5d14b
9	CNYTokenPlus	0xfbb7b2295ab9f987a9f7bd5ba6c9de8ee762deb8

Figure 3: Vulnerable ERC-20 smart contracts.

We can see that these vulnerabilities can not only cause a huge financial loss but also huge loss of reputation to the enterprise. So, it is necessary to avoid these issues.

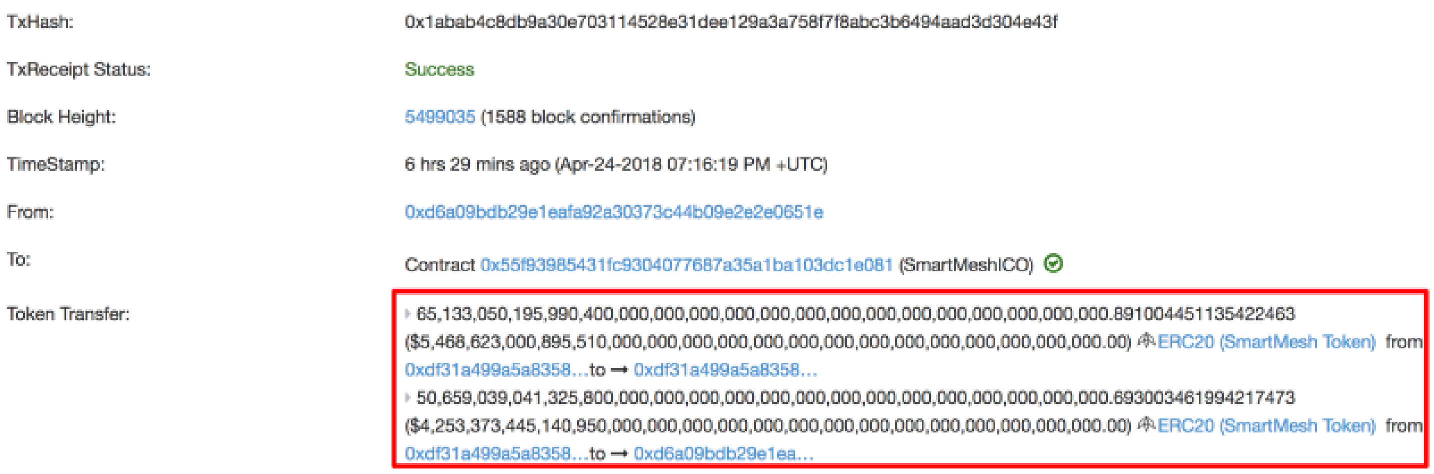
4.2 Motivation

With the rapid development of cryptocurrency, there are more and more enterprises entering this field in order to make huge profit. However, if one enterprise overlook the importance of solving the vulnerabilities in all of the aspects, it will definitely be a disaster. So our motivation of doing this research in this project is to find some solutions to solve these vulnerabilities and avoid loss because of them.

4.3 Existing Solutions

Enterprise: Find the vulnerability and modify the code to solve it. “Counterfeit money” destruction: the SmartMesh Foundation will take the equivalent amount of SMT to the counterfeit amount and destroy it to make up for the losses caused, and keep the total supply of SMT at the value of $3,141,592,653$.

Exchanges: Monitor the transactions and once discover some abnormal trades, respond timely. Centralized exchanges can play a large part in the prevention of malicious attacks by hackers or other people that have bad intentions. In this case people that knew about the exploit tried to profit of it by selling illegally obtained funds. Huobi Pro was one of the first cryptocurrency exchanges to respond. They immediately disabled all deposits and withdrawals of all tokens. After confirmation only ERC-20 tokens could be affected they enabled deposits and withdrawals on all non-ERC-20 tokens. Later Huobi Pro announced the deposit and withdrawal of all ERC-20 tokens were resumed. OKEx was also quick to respond with the suspension of all ERC-20 token deposits.



TxHash: 0x1abab4c8db9a30e703114528e31dee129a3a758f7f8abc3b6494aad3d304e43f

TxReceipt Status: Success

Block Height: 5499035 (1588 block confirmations)

TimeStamp: 6 hrs 29 mins ago (Apr-24-2018 07:16:19 PM +UTC)

From: 0xd6a09bdb29e1eafa92a30373c44b09e2e2e0651e

To: Contract 0x55f93985431fc9304077687a35a1ba103dc1e081 (SmartMeshICO)

Token Transfer:
65,133,050,195,990,400,000,000,000,000,000,000,000,000,000,000,000,000,000,000.891004451135422463 ($5,468,623,000,895,510,000,000,000,000,000,000,000,000,000,000,000,000,000,000.00) ERC20 (SmartMesh Token) from 0xdf31a499a5a8358...to → 0xdf31a499a5a8358...
50,659,039,041,325,800,000,000,000,000,000,000,000,000,000,000,000,000,000,000.693003461994217473 ($4,253,373,445,140,950,000,000,000,000,000,000,000,000,000,000,000,000,000,000.00) ERC20 (SmartMesh Token) from 0xdf31a499a5a8358...to → 0xd6a09bdb29e1ea...

Figure 4: The unusual SMT token transaction.

4.4 Our Observations

On 4/24/2018, 07:16:19 UTC, PeckShield, a blockchain security startup, detected an unusual SMT token transaction. In this particular transaction, someone transferred a large amount of MESH token:

$$0x8fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff$$

to herself along with a huge amount fee:

$$0x7000000000000000000000000000000000000000000000000000000000000001$$

to the address issuing this transaction.

We first check the function $transfer$. We found that this function is correct and has no obvious vulnerability, so we were looking at other functions.

```
function transfer(address _to, unit256 _value)
  public transferAllowed(msg.sender) returns(bool success){
   if(balances[msg.sender] >= _value && balances[_to] + _value > balances[_to]){
      balances[msg.sender] -= _value;
      balances[_to] += _value;
      Transfer(msg.sender, _to, _value);
      return true;
   }
   else {return false;}
}
```

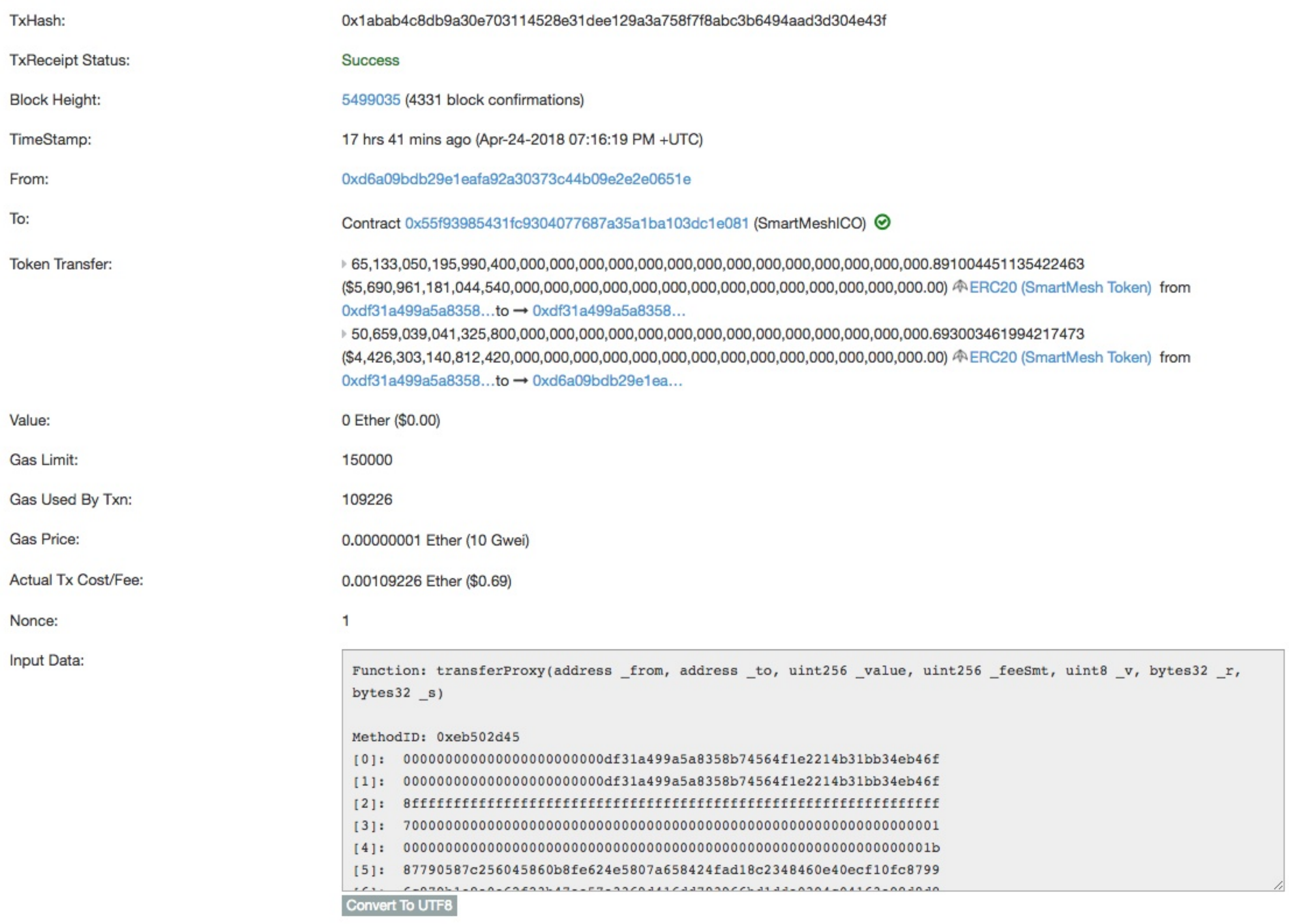
This function first checks whether the balance of sender is bigger than the amount he wants to send. After checking, it will decrease the amount of the balance of the sender and increase the amount of the balance of the receiver. Then return true. It seems like there is no problem. Then let's look at the record of token transfer. Here the hacker called the transferProxy method and passed in six parameters. The value of the third parameter is

$$8fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff$$

The value of the fourth parameter is

$$7000000000000000000000000000000000000000000000000000000000000000$$

The corresponding variables are $\_value$ and $\_fee$. This is so abnormal because the value is so huge. Then let's look at the suspicious function $transferProxy()$.



TxHash: 0x1abab4c8db9a30e703114528e31dee129a3a758f7f8abc3b6494aad3d304e43f

TxReceipt Status: Success

Block Height: 5499035 (4331 block confirmations)

TimeStamp: 17 hrs 41 mins ago (Apr-24-2018 07:16:19 PM +UTC)

From: 0xd6a09bdb29e1eafa92a30373c44b09e2e2e0651e

To: Contract 0x55f93985431fc9304077687a35a1ba103dc1e081 (SmartMeshICO)

Token Transfer: 65,133,050,195,990,400,000,000,000,000,000,000,000,000,000,000,000,000,000,000.891004451135422463 ($5,690,961,181,044,540,000,000,000,000,000,000,000,000,000,000,000,000,000,000.00) ERC20 (SmartMesh Token) from 0xdf31a499a5a8358...to ➞ 0xdf31a499a5a8358...
50,659,039,041,325,800,000,000,000,000,000,000,000,000,000,000,000,000,000,000.693003461994217473 ($4,426,303,140,812,420,000,000,000,000,000,000,000,000,000,000,000,000,000,000.00) ERC20 (SmartMesh Token) from 0xdf31a499a5a8358...to ➞ 0xd6a09bdb29e1ea...

Value: 0 Ether ($0.00)

Gas Limit: 150000

Gas Used By Txn: 109226

Gas Price: 0.00000001 Ether (10 Gwei)

Actual Tx Cost/Fee: 0.00109226 Ether ($0.69)

Nonce: 1

Input Data:

```
Function: transferProxy(address _from, address _to, uint256 _value, uint256 _feeSmt, uint8 _v, bytes32 _r,
bytes32 _s)

MethodID: 0xeb502d45
[0]:  000000000000000000000000df31a499a5a8358b74564f1e2214b31bb34eb46f
[1]:  000000000000000000000000df31a499a5a8358b74564f1e2214b31bb34eb46f
[2]:  8fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
[3]:  7000000000000000000000000000000000000000000000000000000000000001
[4]:  000000000000000000000000000000000000000000000000000000000000001b
[5]:  87790587c256045860b8fe624e5807a658424fad18c2348460e40ecf10fc8799
```

Figure 5: The record of token transfer.

```
function transferProxy(address _from, address _to, unit256 _value, unit256 _fee, unit8 _v, bytes32 _r, bytes32 _s) public transferAllowed(_from) returns(bool){
    if(balances[_from] < _fee + _value) revert();

    unit256 nonce = nonces[_from];
    bytes32 h = keccak256(_from, _to, _value, _fee, nonce);
    if(_from != ecrecover(h,_v,_r,_s)) revert();

    if(balances[_to] + _value < balances[_to] || balances[msg.sender] + _fee < balances[msg.sender]) revert();
    balances[_to] += _value;
    Transfer(_from, _to, _value);

    balances[msg.sender] += _fee;
    Transfer(_from, msg.sender, _fee);

    balances[_from] -= _value + _fee;
    nonces[_from] = nonce + 1;
    return true;
}
```

Now we have found the problem here is that this function has a classic integer overflow problem. As shown in the code, both $\_fee$ and $\_value$ are input parameters which could be controlled by the attacker.

If $\_fee + \_value$ happens to be 0 (the overflow case), the sanity checks could be passed. It means the attacker could transfer a huge amount of tokens to an address with zero balance. Also, a huge amount fee would be transferred to the $msg.sender$.

4.5 Our Solutions

First, enterprises should scrutinize the code of smart contract carefully and audit the smart contract extensively before the ERC-20 token is tradable. Second, to avoid overflow from happening, developers should use SafeMath in Solidity. Third, only use respectable exchanges that respond quickly and handle in the best interest of the customer, like Huobi Pro and OKEx. Forth, users and enterprises should be aware that trading on decentralized exchanges poses a challenge, because these exchanges might not be able to stop attackers from selling the tokens and can't just revert transactions. Fifth, With trading comes risk, enterprises should make sure to put themselves at as low of a risk as possible.