PAYSURA

IPC ERC20 Token Report

contact@paysura.com

February 20, 2018

Abstract

In this IPC Report we describe some possbile attack scenarios on ERC20 Ethereum token standard. Until now there are almost 40000 ERC20 token contracts [1] but even though this standard is strong in circulation it contains some security issues for the token holders. Since the presented vulnerabilites relate to the ERC20 interface itself and no concrete implementation every ERC20 token can be potentially vulnerable. The community is often not even aware of such security issues, which is why this report introduces some important attack scenarios on ERC20 tokens and shows how the IPC is protected against them.

I. Introduction

The Ethereum foundation defines an Ethereum-based token as followed:[2]

"Tokens in the Ethereum ecosystem can represent any fungible tradable good: coins, loyalty points, gold certificates, IOUs, in-game items, etc. Since all tokens implement some basic features in a standard way, this also means that your token will be instantly compatible with the Ethereum wallet and any other client or contract that uses the same standards."

ERC20 is a standard for smart contracts of an Ethereum-based token as described above and supports some useful functionalities besides transferring tokens from one address to another address. The interface[3] of the standard is shown in Fig. 1.

```
// ----------------------------------------------------------------------------
// ERC Token Standard #20 Interface
// https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-standard.md
// ----------------------------------------------------------------------------
contract ERC20Interface {
    function totalSupply() public constant returns (uint);
    function balanceOf(address tokenOwner) public constant returns (uint balance);
    function allowance(address tokenOwner, address spender) public constant returns (uint remaining);
    function transfer(address to, uint tokens) public returns (bool success);
    function approve(address spender, uint tokens) public returns (bool success);
    function transferFrom(address from, address to, uint tokens) public returns (bool success);

    event Transfer(address indexed from, address indexed to, uint tokens);
    event Approval(address indexed tokenOwner, address indexed spender, uint tokens);
}
```

Figure 1: *ERC20 interface*

For a better understanding of this IPC Report we recommend reading the specification of the ERC20 standard at `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md`.

IPC as an Ethereum-based token is implementing the ERC20 standard.

[1] `https://etherscan.io/tokens`

[2] `https://www.ethereum.org/token`

[3] `https://theethereum.wiki/w/index.php/ERC20_Token_Standard`

However it does not only implement the ERC20 specification to act as a refund for a blockchain-based worldwide available reward system but also attaches great importance to safety and security and mitigates currently known attacks on ERC20 tokens or even makes them impossible.

II. Attack Scenario 1

i. Multiple Withdrawal Attack

1. Alice executes the approve method by calling *approve(Bob, N)* to allow Bob to transfer N of Alice's tokens (N>0).
2. After a certain time, Alice wants to change the allowed amount. First she makes sure that Bob didn't take any of the N tokens from her wallet by calling *allowance(Alice, Bob)*.
3. Now she changes the allowance by calling *approve(Bob, M)* to allow Bob to transfer M of Alice's tokens (M>0).
4. Before Alice's transaction is mined Bob sends N tokens from Alice to his wallet by calling *transferFrom(Alice, Bob, N)*.
5. If Bob manages to do his transaction in time, his transaction will be mined before Alice's second transaction.
6. Now Alice's transaction will be mined, and allows Bob to transfer M of Alice's tokens.
7. Bob uses *transferFrom(Alice, Bob, M)* to send M of Alice's tokens to his wallet again.
8. Before Alice noticed that Bob took N of her tokens, Bob took away another M tokens.

By changing Bob's allowance from N to M (N, M > 0), Bob got M+N tokens, while Alice only wanted to allow him to use M tokens.

ii. Which protecive mechanism is IPC using?

The IPC smart contract is using the following line from MiniMeToken.sol[4] within the approve method to mitigate the multiple withdrawal attack:

```
require(newAllowance == 0) || (allowance[msg.sender][sender] == 0));
```

Now the allowance for Bob can only be changed if either the current allowance N is 0 or the new allowance M is 0.

1. **Current allowance N = 0:** Alice hasn't allowed Bob to use any of her tokens yet, so Bob can't use the transferFrom method. Multiple Withdrawal Attack is not possible.
2. **New allowance M = 0:** If Alice wants to change the current allowance N (N>0) to M, the implemented method forces Alice to set M to 0. After the transaction has been succesfully mined Alice will have some time to check if Bob used *transferFrom(Alice, Bob, N')* (N' <= N) to send some of Alice's tokens. Depending on Bob's actions Alice can now execute the approve method to give Bob a new allowance. By doing this Alice can't accidentally allow Bob to use more tokens than she wanted.

[4] https://github.com/Giveth/minime/blob/master/contracts/MiniMeToken.sol

Yet this protection creates one problem for the token holder. Changing the allowed amount N to a new amount M (N, M > 0) requires two function calls thereby creating two transactions which leads to higher costs for the user. To fix this the IPC token provides the following methods in Fig. 2 and Fig. 3 (first envisioned by MonolithDAO token) to directly change the allowance without being vulnerable to the described attack and without paying the gas price for the execution of the *approve()* method two times.

The first method (Fig. 2) makes it possible to increase the allowed amount. By calling this method, Bob's current allowance will be directly increased by N (N>0). A multiple withdrawal attack wouldn't make sense in this case since Bob's amount is being increased by just adding N tokens to the current allowance.

```
145     function increaseApproval(address _spender, uint _addedValue) onlyPayloadSize(2) public returns (bool) {
146         allowance[msg.sender][_spender] = safeAdd(allowance[msg.sender][_spender], _addedValue);
147         Approval(msg.sender, _spender, allowance[msg.sender][_spender]);
148         return true;
149     }
```

Figure 2: *Method to increase the allowance in one step*

The second method (Fig. 3) does the opposite. Alice can decrease Bob's allowance directly by subtracting N (N>0) from the current allowance. So if Bob is allowed to use M tokens and withdraws M′ (M>=M′>=0) tokens while Alice's transaction is being mined, Alice's transaction won't increase the total amount but either set it to zero (if N > M-M′) or set the new allowance to M-M′-N.

```
160 function decreaseApproval(address _spender, uint _subtractedValue) onlyPayloadSize(2) public returns (bool) {
161     uint256 currentValue = allowance[msg.sender][_spender];
162     require(currentValue > 0);
163     if (_subtractedValue > currentValue) {
164         allowance[msg.sender][_spender] = 0;
165     } else {
166         allowance[msg.sender][_spender] = safeSub(currentValue, _subtractedValue);
167     }
168     Approval(msg.sender, _spender, allowance[msg.sender][_spender]);
169     return true;
170 }
```

Figure 3: *Method to decrease the allowance in one step*

III. Attack Scenario 2

i. Short-Address-Attack

1. Let's assume Alice wants to send N (e.g. N = 1) tokens to Bob's deployed smart contract.
2. Alice asks Bob for his contract address. Bob's contract address has trailing 0's like *0xbbbbbb00*.
3. Bob is aware of the functionality and vulnerabilities of Ethereum's EVM. So he decides to give Alice a wrong address. He removes the trailing 0's of his real address and sends the manipulated address *0xbbbbbb* to Alice.
4. Now Alice calls the method *transfer(0xbbbbbb, 1)* to send 1 token to Bob.
5. Translating the method and the given arguments can be as followed: a. Method signature: *0x0124567* b. First argument: *0xbbbbbb* c. Second argument: *0x00000001*

6. Concatenating them creates the following transaction: *0x01234567bbbbbbbb00000001*.

7. Ethereum Virtual Machine (EVM) interprets the created transaction like this: a. Method: *0x0124567* b. First argument: *0xbbbbbb00* c. Second argument: *0x000001xx*

8. Since arguments are 4-byte words, the EVM appends the leading zeros of the second argument to the first argument.

9. At the same time the second argument shifts to the left which increases its value considerably.

10. Now the EVM fills the remaining input data with 0's. So the transaction looks like this: *0x01234567bbbbbbbb0000000100*

11. The contract interprets those extra zeros as part of the correct value, provoking the described issues.

By using a short-address-attack Bob managed to receive 256 (or *0x0000000100*) tokens while Alice only wanted to send 1 (or *0x0000000001*) token.

ii. Which protective mechasim is IPC using?

IPC is using a modifier which performs a length check on *msg.data* (first suggested by redditor izqui9, slighty modified).

```
3  modifier onlyPayloadSize(uint numwords) {
4      assert(msg.data.length >= numwords * 32 + 4);
5  }
6
7  function transfer(address _target, uint256 _value) onlyPayloadSize(2) public {}
8  function transferFrom(address _from, address _targe, uint256 _value) onlyPayloadSize(3) public {}
9  function approve(address _target, uint256 _value) onlyPayloadSize(2) public {}
10
```

Figure 4: *Check payload size*

As you can see the modifier is added to the vulnerable methods to check if the given data input has the correct length which is expected by the contract. Since each argument has a default length of 32 bits and the method signature is 4 bits, the modifier uses this rule to make sure that the given data input has at least a length of 4 + 32 * (number of arguments). Otherwise the transaction will not be mined.

Using this protective mechanism usually entails one risk which affects some smart contracts. Since the modifier requires a minimum length depending on the input parameters of the called function the *onlyPayloadSize* check will fail if used by a subcontract function with fewer arguments. The output value of msg.data.length would be different from the expected minimum length and so the call of the protected function can be refused.

There are also some demands to remove the short-address-attack check because it shouldn't be handled by the smart contracts but at different layers.

The short-address-attack is an issue that can lead to damaging consequences for the token holder. The EVM fills missing input datas with zeros instead of throwing the transaction and with certain clients not serializing the message data properly best way to protect the community from a mistake is to mitigate. It is still recommended to use secure clients.

IV. Token Loss

Apart from the two issues above there is still another one. Many token holders keep sending their tokens to the token contract itself. They accidentally keep mixing up the actual target address with the address of the token smart contract. These leads to a huge amount of token loss and since most token contracts are not able to transfer those accidentally sent tokens back they are stuck in it and the community can never get them back. Github user Dexaran published the following list[5] of lost ERC20 tokens in smart contracts:

1. QTUM, **$1,204,273** lost. watch on Etherscan

2. EOS, **$1,015,131** lost. watch on Etherscan

3. GNT, **$249,627** lost. watch on Etherscan

4. STORJ, **$217,477** lost. watch on Etherscan

5. Tronix , **$201,232** lost. watch on Etherscan

6. DGD, **$151,826** lost. watch on Etherscan

7. OMG, **$149,941** lost. watch on Etherscan

8. STORJ, **$102,560** lost. watch on Etherscan

Figure 5: *Token Loss (Dec. 2017)*

The current ERC20 standard is not able to protect users from sending their tokens to smart contracts which are not intended to work with them. This is why a new ERC standard is being discussed that may protect the user from this. But since it is neither finalized nor in circulation the community and the token developers still must rely on the ERC20 standard. Even though ERC20 standard is not able to protect the user from this the IPC token mitigates this issue a lot in a very simple way.

By simply adding the following line to the *transfer()/transferFrom()* method:

```
require(receiver != address(this));
```

Before sending the tokens the transfer/transferFrom method in the IPC smart contract checks if the target address is the token contract address itself and if so the transaction will fail. This makes it impossible to send IPC to its smart contract by mistake.

This obviously does not replace a new ERC standard because the token can still be sent to other smart contracts that are not developed to work with the tokens so it is still recommended to always check the target address more than one time. However it protects the community from a serious mistake and limits token loss very much since in most cases people keep mixing up their target with the address of the token smart contract itself and not with other different smart contracts. This saves a lot of costs for everyone.

[5] https://github.com/ethereum/EIPs/issues/223