Homework 3

Programming an ERC20 Token Contract

108321033 吳明騰
108321056 唐泳烽
108321062 許恩慈

Introducing our ERC20 Contract

Introduction

When design our ERC20 Contract, we do not directly inherit the complete class ERC20.sol from OpenZeppelin, but use its interface IERC20.sol. The main reason is that we want to understand the design concept of each function of ERC20 and try to solve ERC20 vulnerability Multiple Withdrawal Attack (see P@@@)

constructor()

We directly write the token's symbol, name, decimals, _totalSupply into constructor(), because we think that a smart contract is to generate only one ERC20 token, so it is not necessary to use the function mint(). In addition, after we generate the token, we directly deposit all the token into an EOA and record a Transfer event.

```
constructor(){
   symbol = "TC";
   name = "Tim Coin";
   decimals = 2;
   _totalSupply = 100000;
   balances[_walletAddr] = _totalSupply;
   emit Transfer(address(0), _walletAddr, _totalSupply);
 }
```

Contract SafeMath

In the below figure, for using basic mathematical operations, including addition, subtraction, multiplication and division, we are directly designing a new object SafeMath.

In the example of addition, the variable a must be a non-negative number, so it is only necessary to judge whether b is the positive integer or zero.

In the case of subtraction, it must be guaranteed that the returned value must be a non-negative integer.

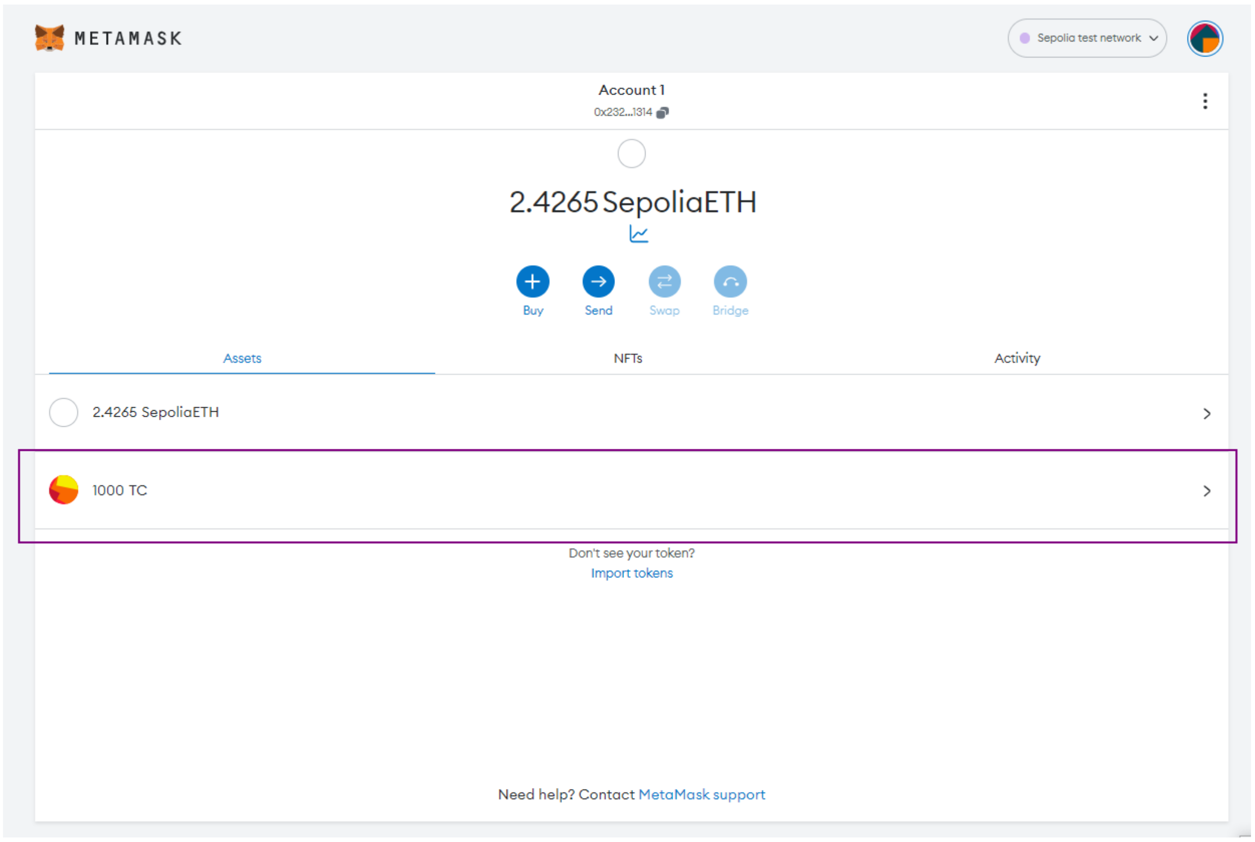
In the example of multiplication, it is necessary to judge the potential problem of the multiplicand and the multiplier being too large that may cause integer overflow.

In the example of division, it is necessary to judge whether the divisor is greater than zero.

```
contract SafeMath {

 function safeAdd(uint a, uint b) public pure returns (uint c) {
  c = a + b;
  require(c >= a);
 }

 function safeSub(uint a, uint b) public pure returns (uint c) {
  require(b <= a);
  c = a - b;
 }

 function safeMul(uint a, uint b) public pure returns (uint c) {
  c = a * b;
  require(a == 0 || c / a == b);
 }

 function safeDiv(uint a, uint b) public pure returns (uint c) {
  require(b > 0);
  c = a / b;
 }
}
```

Using Sepolia

In Figure 1: we mint our own token TC and we import it to Metamask.



Account 1
0x232...1314

2.4265 SepoliaETH

2.4265 SepoliaETH

1000 TC

Don't see your token?
Import tokens

Need help? Contact MetaMask support

Figure 1: The screen of our metamask.

Function approve, transferFrom, allowance

Introduction

In this chapter we will discuss the function approve(), transferFrom(), allowance() Implementation. The reason why we just list three functions is that it is a little difficult to understand these three functions when we first read the ERC20 OpenZeppelin document. In addition to simply introducing the usage of these three functions, we also design a pseudocode to experiment with these functions.

Transfer tokens from A to B

```
function transferFrom(address _from, address _to, uint256 _value) returns (bool success)
```

Parameter 1 is the address for sending tokens, parameter 2 is the address for receiving tokens, and parameter 3 is the quantity. The return value is a Boolean value — 1 for success and 0 for failure.

Approve own token transfer

```
function approve(address _spender, uint256 _value) returns (bool success)
```

Parameter 1 is the address of the object that can claim your own tokens, parameter 2 is the quantity, and the return value is a Boolean value — 1 means success, 0 means failure. By calling this function to approve that an object can take away its own maximum _value tokens through the transferFrom function.

Amount of tokens approved by A to B

```
function allowance(address _owner, address _spender) constant returns (uint256 remaining)
```

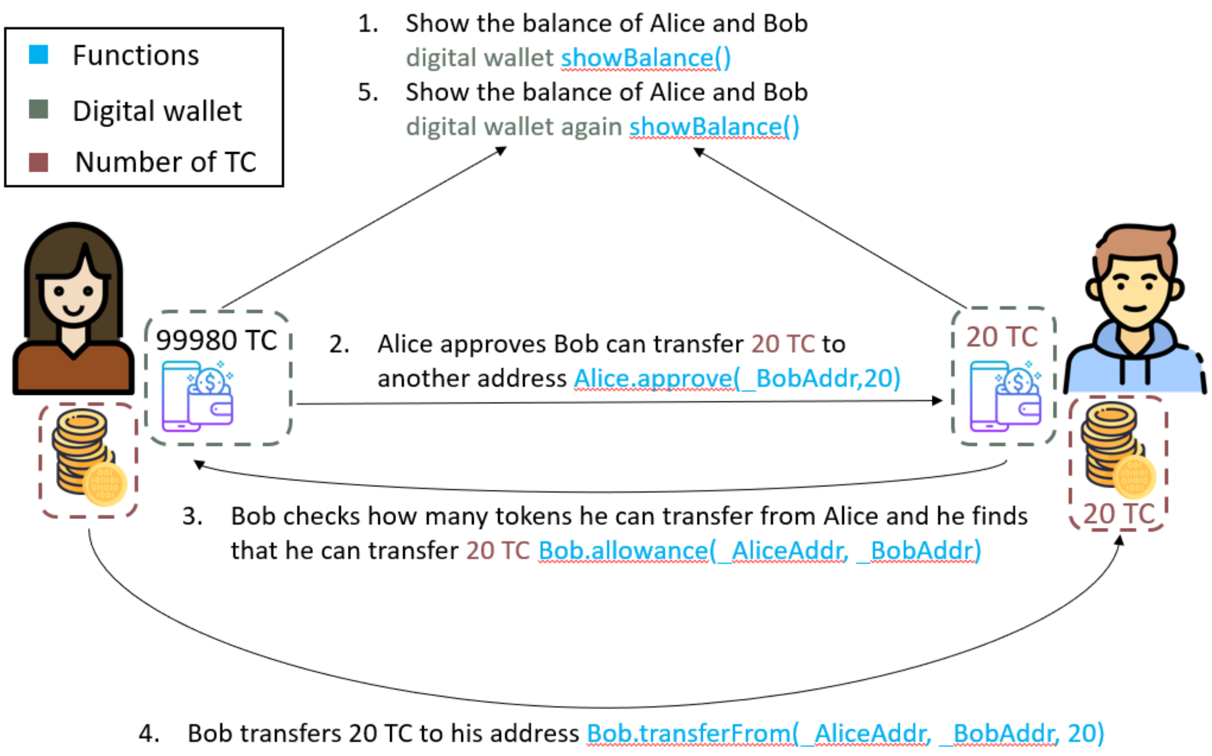
Parameter 1 is the address of the token owner, parameter 2 is the address where the token can be claimed, and the returned value is a positive integer.

Implementation

The figure is the pseudocode we designed, and then we explain the meaning of each line of code one by one.

- Line 1, 10: showBalance() is to show the balance of Alice and Bob
- Line 4: Alice approves Bob can transfer 20 TC to another address.
- Line 5: Bob checks how many tokens he can transfer from Alice and he finds that he can transfer 20 TC (in line 6).
- Line 7: Bob transfers 20 TC to his address.
- Line 8: Bob checks again how many tokens he can transfer from Alice and he finds that he can’t transfer any tokens (in line 9).

```
> showBalance()
==> Alice: 100000 TC
==> Bob: 0 TC
 > Alice.approve(_BobAddr, 20)
 > show Bob.allowance(_AliceAddr, _BobAddr)
 ==> Bob can transfer 20 TC from Alice.
 > Bob.transferFrom(_AliceAddr, _BobAddr, 20);
 > show Bob.allowance(_AliceAddr, _BobAddr)
 ==> Bob can transfer 0 TC from Alice.
 > showBalance()
==> Alice: 99980 TC
 ==> Bob: 20 TC
```
Figure 2: The pseudocode of our implementation.



🟦 Functions
🟩 Digital wallet
🟥 Number of TC
1. Show the balance of Alice and Bob digital wallet showBalance()
5. Show the balance of Alice and Bob digital wallet again showBalance()
99980 TC
2. Alice approves Bob can transfer 20 TC to another address Alice.approve(_BobAddr,20)
20 TC
3. Bob checks how many tokens he can transfer from Alice and he finds that he can transfer 20 TC Bob.allowance(_AliceAddr, _BobAddr)
20 TC
4. Bob transfers 20 TC to his address Bob.transferFrom(_AliceAddr, _BobAddr, 20)

Figure 3: the flowchart of our implementation.

Multiple Withdrawal Attack

Description

This issue has persisted since October 2017, and several proposals have been made that need to be evaluated in terms of compatibility with standards and attack mitigation. This could be a possible attack scenario. [Github issue link]

1. Alice allows Bob to transfer N tokens by calling `approve(_BobAddr, N)`.
2. After a while, Alice decides to change approval from N to M by executing `approve(_BobAddr, M)`.
3. Bob notices Alice's second transaction before it was mined and quickly sends another transaction that runs `transferFrom(_AliceAddr, _BobAddr, N)`. This will transfer N Alice's tokens to Bob.
4. Bob's transaction will be executed before Alice's transaction (because of higher transaction fee, miner's policy or other prioritization ways) and Bob front-runs Alice's transaction.
5. Alice's transaction will be executed after Bob's and allows Bob to transfer more M tokens.
6. Bob successfully transferred N Alice's tokens and gained the ability of transferring another M tokens.
7. Before Alice notices that something went wrong, Bob calls `transferFrom` method again and transfers M Alice's tokens by executing `transferFrom(_AliceAddr, _BobAddr, M)`.

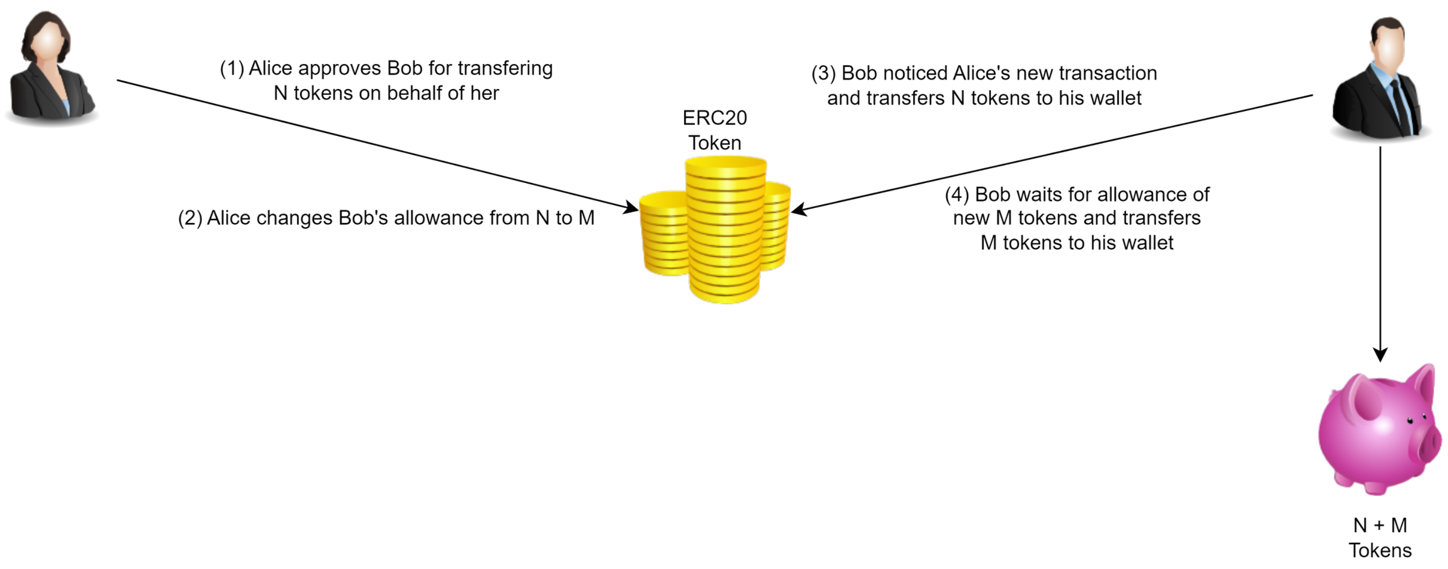

(1) Alice approves Bob for transfering N tokens on behalf of her
(2) Alice changes Bob's allowance from N to M
ERC20 Token
(3) Bob noticed Alice's new transaction and transfers N tokens to his wallet
(4) Bob waits for allowance of new M tokens and transfers M tokens to his wallet
N + M Tokens

Figure 4: Multiple withdrawal attract in ERC20 tokens

```
> showBalance()
==>  Alice : 100000 TC
==>  Bob : 0 TC
> Alice.approve(_BobAddr, 20);
==> Alice’s transaction is waiting to be mined…
> Alice.approve(_BobAddr, 30);
==> Alice’s transaction is waiting to be mined…
> Bob.transferFrom(_AliceAddr, _BobAddr, 20)
==> Bob starts to call tranferFrom()
==> Alice’s transaction is mined successfully
>  Bob.transferFrom(_AliceAddr, _BobAddr, 30)
> showBalance()
==>  Alice : 99950 TC
==>  Bob : 50 TC
```
Figure 5: The pseudocode of multiple withdrawal attract

Some Solutions

Several solutions have been proposed by the Ethereum community (mainly from developers on GitHub) to deal with this attack. Each solution will have some considerations that need to be discussed in terms of compatibility with the ERC20 standard and attack mitigation. We have examined the technical aspects of each of the solutions below:

1. Setting Zero First

The standard recommends setting allowances to zero before any non-zero values. But, as discussed here, this approach is insufficient and still allows Bob to transfer N+M tokens:

1. Bob is allowed to transfer N Alice's tokens
2. Alice publishes transaction that changes Bob's allowance to zero
3. Bob front runs Alice's transaction and transfers N Alice's tokens
4. Alice's transaction is mined
5. Alice sees that her transaction was mined successfully, that Bob's allowance is now zero and that proper Approval event was logged. This is exactly what she would see if Bob would not transfer any tokens from her, so she has no reason to think that Bob actually used his allowance before it was revoked
6. Now Alice publishes transaction that changes Bob's allowance to M

7. Alice's second transaction is mined so now Bob is allowed to transfer M Alice's tokens
8. Bob transfers M Alice's tokens.

2. removing the approve and transferFrom functions
This will prevent the effects of the attack by skipping the implementation of the vulnerable function. While removing the approve and transferFrom functions prevents multiple withdrawal attacks, it makes the token partially ERC20 compliant. Golem Network Token (GNT) is one of these examples as it does not implement the approve, allowance and transferFrom functions. According to the ERC20 specification, these methods are not optional and must be implemented. Also, ignoring them will cause function calls from standard wallets that expect to interact with them to fail. Therefore, although the attack is mitigated, most people do not consider this solution a compatibility fix.

Conclusion
Based on ERC20 specifications, token owners should be aware of their approval consequences. But it is somewhat difficult to complete the multiple withdrawal attack. First, Bob must first know when Alice is going to send a transaction to change Bob's allowance. Second, he has to pay a higher gas fee. When we use Ganache to implement this kind of attack, it is not easy to meet the above two conditions, so we do only add a delay function in Line 4 ~ 7 (In figure 5), so that Bob has time to transfer the first amount of token from Alice.Therefore, we think it is really not easy for this kind of attack to happen in real life. If We need to develop your own ERC20 token in the future and prevent multiple withdrawal attacks, you may have to refer to other predecessors for how to rewrite the content of ERC20 to increase security.

Reference

1. Multiple withdrawal attack
https://blockchain-projects.readthedocs.io/multiple_withdrawal.html
2. ERC20 Token使用手冊
https://medium.com/taipei-ethereum-meetup/erc20-token%E4%BD%BF%E7%94%A8%E6%89%8B%E5%86%8A-3d7871c58bea