# Resolving the Multiple Withdrawal Attack in ERC20 Tokens

Reza Rahimian, Shayan Eskandari, Jeremy Clark
*Concordia Univeristy*

***Abstract*—Custom tokens are an integral part of Ethereum decentralized applications. The ERC20 standard defines a token interface that is interoperable with any ERC20-compliant application, which might be another decentralized application or a web service. A security issue, known as the *multiple withdrawal attack*, was raised on GitHub and has been open since October 2017. It exploits ERC20 standard API describing a method called `approve()`, which is used by a token holder to approve other users (or applications) to withdrawal a capped number of tokens. If the token holder makes adjustments to the amount of tokens approved, a malicious entity can front-run this new approval, withdraw the original allotment of tokens, and then wait for the new approval to be confirmed which provides a new allotment of tokens available for withdrawal. In this paper, we (1) examine *ten* suggested solutions in accordance with specifications of the standard and being backward compatible with already deployed smart contracts; and (2) propose two new solutions that mitigate the attack.**

***Index Terms*—Cryptocurrency; Security; ERC20; Token; Smart Contract; Front-Running; Ethereum; Blockchain;**

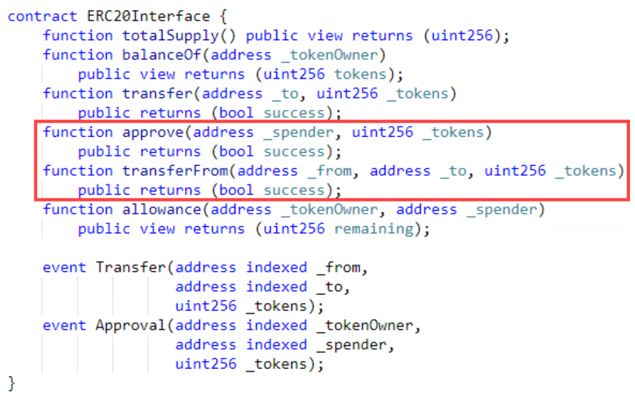
## 1. Introduction

Ethereum is a public blockchain proposed in 2013, deployed in 2015 [7], and has the second largest market cap at the time of writing[1]. It has a large development community which track enhancements and propose new ideas.[2] Ethereum enables decentralized applications (DApps) to be deployed and executed. DApps can accept and transfer Ethereum's built-in currency (ETH) or might issue their own custom currency-like tokens for specific purposes. Tokens might be currencies with different properties than ETH, they may be required for access to a DApp's functionality or they might represent ownership of some off-blockchain asset. It is beneficial to have interoperable tokens with other DApps and off-blockchain webapps, such as exchange services that allow tokens to be traded.

Toward this goal, the Ethereum project accepted a proposed standard (called ERC20 [8]) for a set of methods which standard-compliant tokens should implement. In terms of object oriented programming, ERC20 is an interface that defines abstract methods (name, parameters, return types) and provides guidelines on how the methods should be implemented, however it does not provide a actual concrete implementation (see Figure 1).

Since introduction of ERC20 API in November 2015, several vulnerabilities have been discovered. In October

1. CoinMarketCap - Ethereum currency - Accessed: 2019-02-11 https://coinmarketcap.com/currencies/ethereum/
2. CoinDesk Crypto-Economics Explorer - Accessed: 2019-02-11 https://www.coindesk.com/data



```
contract ERC20Interface {
    function totalSupply() public view returns (uint256);
    function balanceOf(address _tokenOwner)
        public view returns (uint256 tokens);
    function transfer(address _to, uint256 _tokens)
        public returns (bool success);
    function approve(address _spender, uint256 _tokens)
        public returns (bool success);
    function transferFrom(address _from, address _to, uint256 _tokens)
        public returns (bool success);
    function allowance(address _tokenOwner, address _spender)
        public view returns (uint256 remaining);

    event Transfer(address indexed _from,
                   address indexed _to,
                   uint256 _tokens);
    event Approval(address indexed _tokenOwner,
                   address indexed _spender,
                   uint256 _tokens);
}
```

Figure 1. ERC20 standard defines 6 functions and 2 events that have to be implemented as part of ERC20-compliant token. Using `approve` and `transferFrom` functions in existence of race condition may lead to *"multiple withdrawal attack"*

2017, a security issue called *"Multiple Withdrawal Attack"* was opened on GitHub[17], [9]. The attack originates from two methods in the ERC20 standard for approving and transferring tokens. The use of these functions in an undesirable situation (*e.g.,* front-running [6]) could result in more tokens being spent by the allowed third party. This issue is still open and several solutions have been made to mitigate it. We started off by analyzing two implementations of ERC20 token provided by OpenZeppelin[13] and ConsenSys[3]. They are advised by authors of ERC20 standard [8] as sample implementations. OpenZeppelin mitigates the attack by introducing two additional methods as replacement of `approve` function and ConsenSys implementation does not attempt to resolve the attack. Other implementations have a variety of different trade-offs in solving the issue that will be discussed in section 3.

**Contributions.** In this paper, we analyze 10 proposed solutions and evaluate them in accordance with (A) backward compatibility with already deployed smart contracts and (B) attack mitigation while adhering specifications of the standard (see the summary in Figure 2). Since none of them precisely satisfy constraints of ERC20 standard, we were motivated to work on new solutions to mitigate the attack. Specifically, we propose two new approaches: each secures one of the two vulnerable methods (*i.e.,* `approve` or `transferFrom`). The first one, mitigates the attack by comparing transferred tokens with the current allowance in the `approve` method. It uses *Compare and Set (CAS)* pattern[21] to cover gap between transactions as root cause of the attack. CAS is one of the most widely used lock-free synchronization strategy that allows comparing and setting values in an atomic way. By leveraging this pattern, the gap between transactions will be removed and there is no possibility of effecting the attack. We would consider this solution as partially ERC20-compliant solution since it violates one of ERC20 constraints that prohibits any adjustment in the allowance. As discussed in section 4.1, it would not be feasible to secure `approve` method without adjusting spender allowance. Alternatively, securing `transferFrom` method has been considered as effective mitigation approach by not allowing token transfers more than allowed.

## 2. Preliminaries

### 2.1. How *multiple withdrawal attack* works

According to the ERC20 API definition:

- The `approve` function allows spender to withdraw up to allowed tokens from token pool of the approver. If this function is called again, it overwrites the current allowance with the new input value.
- The `transferFrom` function allows the spender (*e.g.,* user, wallet or other smart contracts) to transfer tokens on behalf of the approver. It updates balance of transaction parties accordingly.

An adversary can exploit the gap between execution of `approve` and `transferFrom` functions since the `approve` method overrides current spender allowance regardless of whether the spender already transferred any tokens or not. This functionality of the `approve` method is defined by the standard and cannot be changed. Furthermore, transferred tokens are not trackable and only `Transfer` events are logged, which is not sufficient in case of transferring tokens to a third party (i.e., `Transfer` event will not be related to the spender). To make it clear, the following attack scenario can be applied:

1) Alice allows Bob to transfer N tokens on her behalf by executing `approve(_Bob, N)`.
2) Later, Alice decides to change Bob's approval from N to M by calling `approve(_Bob, M)`.
3) Bob notices Alice's second transaction after it is broadcast to the Ethereum network but before it is added to a block.
4) Bob using the asymmetric insertion method [6] and front-runs the original transaction to run `transferFrom(_Alice, _Bob, N)` first. If a miner is incentivized to add this transaction before Alice's, it will transfer N Alice's tokens to Bob before changing Bob's approval.
5) Alice's transaction will be executed after Bob's.
6) Bob can call `transferFrom` method again and transfers M additional tokens by executing `transferFrom(_Alice, _Bob, M)`.

In summary, in attempting to change Bob's allowance from N to M, Alice makes it possible for Bob to transfer N+M of her tokens. We operate on the assumption that a secure implementation would prevent Bob from withdrawing Alice's tokens multiple times when the allowance changes from N to M (see Figure 3). If Bob could withdraw N tokens after Alice initial approval, this would be

| # | Proposed solution | Is it backward compatible? | Does it remediate vulnerable functions? | Does it allow non-zero allowances? | Does it allows zero token transfers? | Does it mitigate the attack? |
|---|---|---|---|---|---|---|
| | **Suggested Solutions** | | | | | |
| **1** | Enforcement by UI | ✓ It does not change any code | ✗ It does not change any code | ✓ By default approve method | ✓ By default transferFrom method | ✗ Race condition can occur |
| **2** | MiniMe Token | ✓ It adds only one line to approve method | ✗ Only forces allowance to be zero before non-zero values | ✓ If it is already zero, otherwise in two calls | ✓ By default transferFrom method | ✗ Race condition can occur |
| **3** | Minimum viable token | ✗ It does not implement vulnerable functions | ✗ It does not implement approve function | ✗ It does not implement approve function | ✗ It does not implement transferFrom function | ✓ By not addressing vulnerable functions |
| **4** | Approving trusted parties | ✓ It does not change any code | It depends on code verification | ✓ By default approve method | ✓ By default transferFrom method | ✓ It is non-comprehensive solution |
| **5** | Monolith DAO | ✗ It adds two new functions | ✗ It does not change any code | ✗ It adjusts allowance | ✓ By default transferFrom method | ✓ By using two new methods |
| **6** | Alternate approval function | ✗ It adds one new function | ✗ It does not change any code | ✓ By using new method | ✓ By default transferFrom method | ✓ By using new method |
| **7** | Changing ERC20 API | ✗ It adds new overloaded approve method | ✓ By new method with three parameters | ✓ By using new method | ✓ By default transferFrom method | ✓ By using new method |
| **8** | New token standards | ✗ It introduces new API | ✓ | ✓ | ✓ | ✓ |
| **9** | Detecting token transfers | ✓ It adds two lines to approve method | ✓ | ✗ It locks down allowance in case of any token transfer | ✓ By default transferFrom method | ✓ It blocks legit and non-legit allowances as well |
| **10** | Keeping track of remaining tokens | ✓ It adds three lines to the approve method | ✓ | ✓ | ✓ By default transferFrom method | ✗ Race condition can occur |
| | **New Proposals** | | | | | |
| **11** | Proposal 1: Securing approve method | ✓ It adds new codes to the approve method | ✓ | ✗ It adjusts the allowance | ✓ | ✓ |
| **12** | Proposal 2: Securing transferFrom method | ✓ It adds new codes to transferFrom method | ✓ It secures transferFrom method | ✓ By default approve method | ✓ | ✓ |

Figure 2. Comparison of 10 suggested solutions with two proposals contributed by this paper. In proposal 1, CAS pattern is used to mitigate the attack by comparing transferred tokens with new allowance. It is partially ERC20-compliant since it adjusts the allowance. In proposal 2, a new local variable has been defined to keep tracking of transferred tokens and prevent more transfers in case of already transferred tokens. It secures `transferFrom` function and prevents the attack.

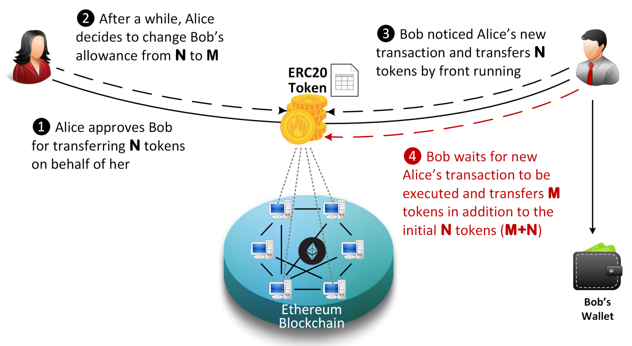

2 After a while, Alice decides to change Bob's allowance from N to M
3 Bob noticed Alice's new transaction and transfers N tokens by front running
ERC20 Token
1 Alice approves Bob for transferring N tokens on behalf of her
4 Bob waits for new Alice's transaction to be executed and transfers M tokens in addition to the initial N tokens (M+N)
Ethereum Blockchain
Bob's Wallet

Figure 3. Possible multiple withdrawal attack in ERC20 tokens when Alice changes Bob's allowance from N to M. By front-running, Bob is able to move N+M tokens from token pool of Alice.

considered as legitimate transfer, since Alice has already approved it. In other words, it would be the responsibility of Alice to make sure tokens have not been transferred before modifying her approval for Bob.

## 2.2. General prevention techniques

In general, there would be off-chain and on-chain transaction to consider when proposing solutions. For this paper, off-chain transactions are out of scope. Therefore, concentration will be on on-chain prevention techniques:

1) **Prevention by token owner**: Considering owner responsibility to mitigate the attack in UI (*e.g.,* Web3.js) instead of contract (*i.e.,* Solidity/Vyper).
2) **Prevention by token author**: Smart contract author(s) have to mitigate the attack by securing either `approve` or `transferFrom` methods.

**Prevention by token owner.** This approach is recommended by ERC20 authors [8] and advises token holder to change spender allowance from N to zero before zero to M (instead of directly from N to M). This will not prevent the attack because changing of allowance from N to zero by the owner is indistinguishable from transferred tokens by the attacker. In other words, after execution of `approve` function, token holder sees allowance=0, but cannot distinguish whether it was because of execution of `approve` method or has been set to zero in `trnasferFrom` function due to token transfer. Although it would be possible to track transferred tokens through `Transfer` events, but it would miss some information in case the transfer happens to a third-party. For example, if Alice allows Bob to transfer N tokens, Bob can transfers N tokens to anyone not just to himself. He might call `transferFrom(_Alice, _Carol, N)` to transfer the N tokens to Carol. In this case, `Transfer`[3] event creates a log showing Carol moved N tokens from Alice with no indication that it was instigated by Bob. As discussed later in 3.4, this approach can not prevent the attack since it is not distinguishable which transaction (*i.e.,* owner or spender transaction) has set the allowance to zero. Consequently, Alice cannot decide whether to change Bob's allowance to M or not. Changing his allowance to M might allow Bob to transfer more M tokens (N+M in total) and not changing it will prevent any legitimate allowance change.

**Prevention by `approve` method.** Securing `approve` method could be a solution and prevent the attack. By using compare and set (CAS) pattern [21], `approve` method can change spender allowance from N to M atomically (*i.e.,* comparing new allowance with transferred token and set it accordingly). Comparison part of CAS requires knowledge of previously transferred tokens that will reveal any token transfer in case of front-running. Although this is promising approach, setting new allowance in `approve` method must satisfy ERC20 constraint that dictates "If this function is called again it overwrites the current allowance with `_value`" [8]. This does not allow any adjustments in allowance while it is a prerequisite for securing `approve` method. For example, considering front-running by Bob when Alice changes his allowance form 100 to 110, the `approve` method can reveal 100 transferred tokens by Bob and has to set Bob allowance to 10 (110-100=10). However, based on ERC20 constraints, it must not adjust the allowance and has to set it literally to 110. Consequently, this allows Bob to transfer 100+110=210 tokens in total. We implemented this approach in 4.1 and concluded that securing `approve` method cannot prevent the attack while adhering constraints of the ERC20 standard.

**Prevention by `transferFrom` method.** Based on ERC20 specifications, "approve functions allows `_spender` to withdraw from your account multiple times, up to the `_value`". Therefore, spender must not be able to transfer more than allowed tokens. That being said, `transferFrom` method can prevent transferring of new tokens in case of already transferred tokens up to allowed amount. By comparing transferred tokens in `transferFrom` method, spender will be restricted to move solely the remaining tokens of his allowance. In case of trying to transfer more tokens than allowed, the transaction fails. For example, Alice's new transaction for increasing Bob allowance from 100 to 110, sets Bob allowance to 110 (`approve` method sets allowance regardless of transferred tokens). However, `transferFrom` method prevent the attack by not allowing Bob to transfer more than 10 tokens if he had already transferred 100 tokens. We implemented this approach in 4.2 and it mitigates the attack effectively.

## 2.3. Properties of acceptable solutions

An important criterion for a solution is to adhere the specifications of ERC20 standard. Conforming with the standard ensures that new tokens are backward-compatible with already deployed smart contracts or web applications. We summarized ERC20 constraints [8] that must be satisfied by any sustainable solution:

1) Calling `approve` function has to overwrite current allowance with new allowance.

3. `Transfer` event is defined as `Transfer(address _from, address _to, uint256 _tokens)`. It does not log `msg.sender` who called `transferFrom` function. Therefore, Bob can easily call `transferFrom` by specifying address of someone else (`_from`=Alice's address, `_to`=Carol's address).

2) `approve` method does not adjust allowance, it sets new value of allowance.
3) Transferring 0 values by `transferFrom` method MUST be treated as normal transfers and fire the `Transfer` event as non-zero transactions.
4) Spender will be allowed to withdraw from approver account multiple times, up to the allowed amount.
5) Transferring initial allowed tokens is considered as legitimate transfer. It could happen right after approval or before changing it.
6) Race condition MUST not happen in any case to prevent multiple withdrawal from the account.
7) Introducing new methods violate ERC20 API and MUST be avoided for having compatible token. In other words, introducing new secure methods that are not defined in the initial ERC20 API, make them unusable with already deployed smart contracts. Therefore, smart contracts cannot use them to prevent the attack.

## 3. Evaluating Proposed Solutions

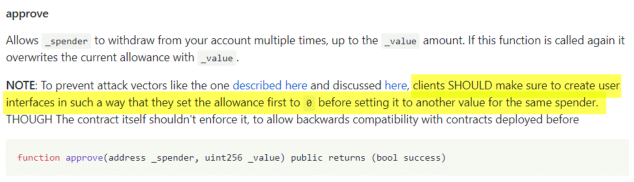
Several solutions have been proposed by Ethereum community—mostly from developers on GitHub[9]—to address the attack. There would be some trad-offs for each solution that needs to be evaluated in term of conforming with standard constraints and attack mitigation. We have examined mitigation approach of each solution and explained possible ERC20 constraint violation.

### 3.1. Enforcement by User Interface (UI)

ERC20 standard recommends to set allowance to zero before any non-zero values and enforce approval processing check in UI instead of smart contract (see Figure 4). If Alice does not use UI and connects directly to the blockchain, there would be a good chance of impacting by this attack. Even if she uses UI, this approach can not prevent the attack [18] and Bob can still transfer N+M tokens in the below scenario:

1) Bob is allowed to transfer N Alice's tokens.
2) Alice publishes a new transaction that changes Bob's allowance from N to 0.
3) Bob front runs Alice's transaction and transfers N Alice's tokens (`transferFrom` sets Bob's allowance to 0 respectively).
4) Alice's transaction is mined and sets Bob's allowance to 0.
5) Now Alice publishes a new transaction that changes Bob's allowance from 0 to M.
6) Alice's second transaction is mined, Bob now is allowed to move M Alice's tokens.
7) Bob transfers M Alice's tokens and in total N+M.

At step 3, Bob is able to transfer N tokens and consequently his allowance becomes 0 by `transferFrom` method. This is considered as a legitimate transaction since Alice has already approved it. The issue occurs after Alice's new transaction in step 5 to set Bob's allowance to M. In case of front-running by Bob, Alice needs to check Bob's allowance for the second time before setting any new value. However, she finds out Bob's allowance 0 in either case. In other words, she can not distinguish whether Bob's allowance is set to 0 because of her transaction in step 2 or Bob already transferred tokens by front-running is step 3. Someone may point out that Alice notices this by checking `Transfer` event logged by `transferFrom` function. However, if Bob had transferred tokens to someone else (like Carol), then `Transfer` event will not be linked to Bob, and, if Alice's account is busy and many people are allowed to transfer from it, Alice may not be able to distinguish this transfer from a legitimate one performed by someone else. Overall, this solution does not prevent the attack while tries to follow ERC20 recommendations for setting Bob's allowance to zero before any non-zero value. Hence, enforcement should be considered at contract level not UI to remove the gap between transactions and mitigate the attack. Interestingly, OpenZeppelin example implements a workaround in contract level that makes it inconsistent with the recommendations of ERC20.



**approve**

Allows `_spender` to withdraw from your account multiple times, up to the `_value` amount. If this function is called again it overwrites the current allowance with `_value`.

**NOTE**: To prevent attack vectors like the one described here and discussed here, clients SHOULD make sure to create user interfaces in such a way that they set the allowance first to `0` before setting it to another value for the same spender. THOUGH The contract itself shouldn't enforce it, to allow backwards compatibility with contracts deployed before

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

Figure 4. Recommendation of ERC20 standard to mitigate multiple withdrawal attack by enforcement in UI.

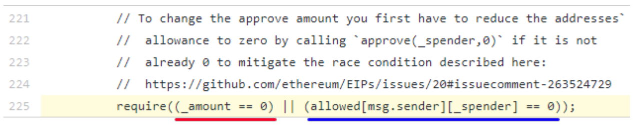
### 3.2. Minimum viable token

As suggested by Ethereum Foundation[15], we can boil down ERC20 standard to a very basic functionalities by implementing only essential methods. This prevents effecting of the attack by skipping implementation of vulnerable functions. While removing `approve` and `transferFrom` functions prevent the attack, it makes the token partially-ERC20-compliant. Golem Network Token (GNT[4]) is one of these examples since it does not implement the `approve`, `allowance` and `transferFrom` functions. According to ERC20 specifications[8], these methods are not OPTIONAL and must be implemented. Moreover, ignoring them will cause failed function calls from standard smart contracts that expect to interoperate with these methods. Therefore, we would not consider it as a backward compatible solution although mitigates the attack by removing vulnerable functions.

### 3.3. Approving trusted parties

Approving token transfer to parties that we trust, reduces risk of impacting by the attack. Non-upgradable smart contracts or trusted accounts can be considered safe for delegating token transfers on behalf of us. Because they do not contain any logic to take advantage of this vulnerability. However, upgradable smart contracts may

4. https://etherscan.io/address/0xa74476443119A942dE498590Fe1f2454d7D4aC0d#code



```
221        // To change the approve amount you first have to reduce the addresses`
222        //  allowance to zero by calling `approve(_spender,0)` if it is not
223        //  already 0 to mitigate the race condition described here:
224        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
225        require((_amount == 0) || (allowed[msg.sender][_spender] == 0));
```

Figure 5. MiniMeToken added code to `approve` method for allowing non-zero allowance values if it is already set to zero.

add new code to their new versions that needs code re-verification before approving token transfers. Similarly, approving token transfer to people that we trust could be considered as a mitigation plan. Nonetheless, this solution would have limited use cases and it could not be considered as a comprehensive prevention mechanism to mitigate the attack.

## 3.4. MiniMeToken

MiniMeToken[11] follows ERC20 recommendation for setting allowance to zero before non-zero values. The owner needs to execute two transactions, setting allowance from N to zero and then from 0 to M. MiniMeToken added a new line of code to the `approve` method for preventing the attack by enforcing this rule. The red clause in line 225 (`_amount == 0`) allows setting of approval to 0 and blue condition checks allowance of `_spender` to be 0 before setting to non-zero values (*i.e.,* If `_spender` allowance is 0 then allows non-zero values-See Figure 5).

Similar to Enforcement by User Interface (UI) and as explained there, this solution will not prevent Bob from transferring N+M tokens. Because Alice would not be able to distinguish whether N tokens have been already transferred or not as described in the below scenario:

1) Alice decides to set Bob's allowance to 0.
2) Bob front-runs Alice's transaction and his allowance sets to 0 after transferring N tokens.
3) Alice's transaction is executed and sets Bob's allowance to 0 (Red clause passes sanity check).
4) Alice checks Bob's allowance and she will find it 0, so, she can not determine whether this was because of her transaction or Bob already transferred N tokens.
5) Alice considers that Bob has not transferred any tokens and allows him for transferring new M tokens.
6) Bob is able to transfer new M tokens and N+M in total.

## 3.5. MonolithDAO

MonolithDAO Token[16] implements two additional functions for allowance increase or decrease. The default `approve` function has additional codes to enforce the owner for setting allowance to zero before non-zero values. It allows non-zero spender's allowance if it is already set to zero. The below table shows functionality of `approve` method based on current spender's allowance and passed input `_value` as new allowance (see Figure 6). If the current spender's allowance is non-zero, `decreaseApproval`[5] and `increaseApproval`[6] functions have to be used for decreasing or increasing the allowance. Using these two functions can prevent race condition and mitigate the attack as explained below:

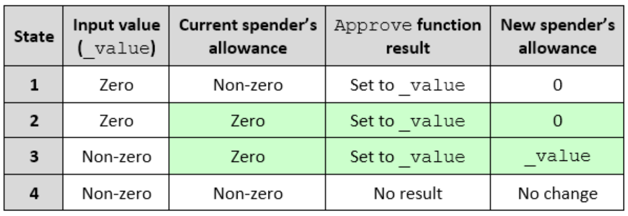
1) Alice allows Bob to transfer N tokens by calling `approve(_Bob, N)`. Alice used the default `approve` function consciously since current Bob's allowance is 0. So, he checked Bob's allowance before calling `approve` method.
2) After a while, Alice decides to decrease Bob's approval by M and runs `decreaseApproval(_Bob, M)`.
3) Bob notices Alice's second transaction and front runs it by executing `transferFrom(_Alice, _Bob, N)`.
4) Bob's transaction will be executed first and transfers N token to his account and the his allowance becomes 0 as result of this transfer.
5) Alice's transaction is mined after Bob's transaction and tries to decrease Bob's allowance by M. If Bob had already transferred more than M tokens, new Bob's allowance becomes negative and it fails the transaction. So, the transaction does not change Bob's remaining allowance and he would be able to transfer the rest (which is legitimate transfer since Alice has already approved it). If Bob had transferred less than M tokens, the new allowance will be applied and reduces Bob's allowance by M.

Although these two new complementary functions prevent the attack, they have not been defined in the initial specifications of ERC20 standard. Therefore, they can not be used by smart contracts that are already deployed on the Ethereum network and still call `approve` method for setting new allowance—and not `increaseApproval` or `decreaseApproval`. Moreover, ERC20 specifications does not define any increase or decrease of allowance. It only allows setting new allowances without adjustment. For example, if Alice has approved Bob for 100 tokens and wants to set it to 80, the new allowance should be 80 while using decrease methods will set it to 20 (100 - 80 = 20). Comparatively, increase method sets new allowance to 180 while it has to set it to 80 again to be in-compliant with ERC20 specification. For these reasons, this solution would not be compatible with ERC20 standard and only is usable if approver or smart contract are aware of these supplementary methods.

## 3.6. Alternate approval function

Another suggestion[2] is to move security checks to another function called `safeApprove`[7] that compare current and new allowance value and sets it if has not been already changed. By using this function, Alice uses the standard `approve` function to set Bob's allowance to

5. `decreaseApproval(address _spender, uint _subtractedValue)`
6. `increaseApproval(address _spender, uint _addedValue)`
7. `safeApprove(address _spender, uint256 _currentValue, uint256 _value`



| State | Input value (`_value`) | Current spender's allowance | `Approve` function result | New spender's allowance |
|---|---|---|---|---|
| 1 | Zero | Non-zero | Set to `_value` | 0 |
| 2 | Zero | Zero | Set to `_value` | 0 |
| 3 | Non-zero | Zero | Set to `_value` | `_value` |
| 4 | Non-zero | Non-zero | No result | No change |

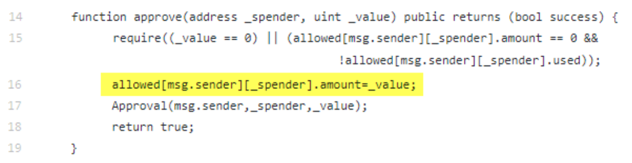
Figure 6. Functionality of default `approve` method in MonolithDAO token that enforces setting spender's allowance to zero before any non-zero values. It implements ERC20 recommendation for changes allowance from N to M in two steps (N→0→M).

0 and for new approvals, she has to use `safeApprove` function. `safeApprove` takes the current expected approval amount as input parameter and calls `approve` method if previous allowance is equal to the current expected approval. By using this function, Alice will have one step more to read the current allowance and pass it to the new `safeApprove` method. Although this approach mitigates the attack by using CAS pattern[21], however, it is not backward compatible with already implemented smart contracts due to their unawareness of this new complementary function. In other words, the new `safeApprove` method is not defined in ERC20 standard and existing smart contracts would not be able to use this new safety feature.

## 3.7. Detecting token transfers

In order to set new allowance atomically, tracking of transferred tokens is required to detect token transfers before setting new allowances. If `approve` method reveals any transferred tokens due to front running, it throws an exception without setting new allowance. As suggested by [20], a flag can be used to detect whether any token has been transferred or not. `transferFrom` method sets this flag to `true` in case of any token transfer. `approve` method checks the flag to be `false` before allowing new approvals. This approach requires new data structure to keep track of used/transferred tokens for each spender. It can prevent race condition as described below:

1) Alice runs `approve(_Bob, N)` to allow Bob for transferring N tokens. Since Bob's initial allowance is 0 and his corresponding flag=`false`, then sanity check passes and Bob's allowance sets to N (line 16).
2) Alice decides to set Bob's allowance to 0 by executing `approve(_Bob, 0)`.
3) Bob front-runs Alice's transaction and transfers N tokens. Then, `transferFrom` turns his flag to `true`.
4) Alice's transaction is mined and passes sanity check because passed value is 0 in line 15.
5) Bob's allowance is set to 0 while his flag remains `true`. (`approve` method does not flip spender flags.)
6) Alice wants to change Bob's allowance to M by executing `approve(_Bob, M)`. Since Bob already transferred N tokens (his flag=`true`), then transaction fails.
7) Bob's allowance does not change and he cannot move more tokens than initially allowed.



```
14      function approve(address _spender, uint _value) public returns (bool success) {
15            require((_value == 0) || (allowed[msg.sender][_spender].amount == 0 &&
                                               !allowed[msg.sender][_spender].used));
16            allowed[msg.sender][_spender].amount=_value;
17            Approval(msg.sender,_spender,_value);
18            return true;
19      }
```

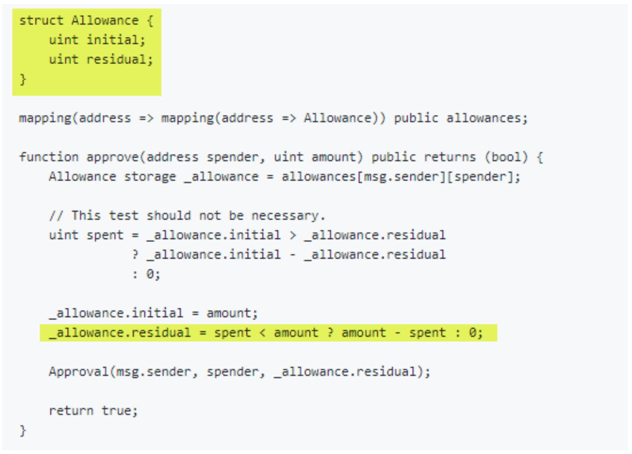
Figure 7. `approve` method needs to be modified by adding a line of code like `allowed[msg.sender][_spender].used = false;` between lines 16 and 17 to unlock spender flag for the next legitimate change.

Although this approach mitigates the attack, but it prevents any further legitimate approvals as well. Considering a scenario that Alice rightfully wants to increase Bob's allowance from N to M (two non-zero values). If Bob had already transferred number of tokens, Alice would not be able to change his approval. Because Bob's flag is set to `true` and line 15 does not allow changing allowance by throwing an exception (see Figure 7). Even setting allowance to 0 and then to M, does not flip the flag to `false` (There is no code for it in the `approve` method). So, It keeps Bob's allowance locked down and blocked further legitimate allowances.

In fact, `approve` method needs a new code between lines 16 and 17 to set the flag to `false`. But it will cause another problem. After setting allowance to 0, spender flag becomes `false` and allows non-zero values event if tokens have been already transferred. It resembles the initial state of allowance similar when nothing was transferred. For example, considering front-running by Bob, before new allowance change from N to 0 by Alice. Bob's flag turns to `true` by `transferFrom` method and turns to `false` by `approve` method afterwards. Now if Alice wants to set allowance from 0 to M, Bob's flag is `false` and his allowance is zero. This is similar to the situation that he did not transfer any tokens. So, Alice cannot distinguish whether Bob moved any token or not. Setting new allowance will allow Bob to transfer more tokens than Alice wanted. Therefore, adding new code makes attack mitigation functionality ineffective. In short, this approach can not satisfy both legitimate and non-legitimate scenarios. Nevertheless, it is a step forward by introducing the need for a new variable to track transferred tokens.

## 3.8. Keeping track of remaining tokens

This approach[14] is inspired by the previous solution and keeping track of remaining tokens instead of detecting transferred tokens. `approve` method uses these variables to set allowance accordingly. It uses modified version of data structure that used in the previous solution for storing residual tokens (see Figure 8). At first look, it seems to be a promising solution by setting approval to zero before non-zero values. However, the highlighted code in `approve` method resembles the situation that we explained in ”Enforcement by User Interface (UI)”. In case of front-running, both `initial` and `residual` variables will be zero and it would not be possible for Alice to distinguish if any token transfer have occurred due to her allowance change or Bob token transfer. To make it more clear, considering the below scenario:



```
struct Allowance {
    uint initial;
    uint residual;
}

mapping(address => mapping(address => Allowance)) public allowances;

function approve(address spender, uint amount) public returns (bool) {
    Allowance storage _allowance = allowances[msg.sender][spender];

    // This test should not be necessary.
    uint spent = _allowance.initial > _allowance.residual
              ? _allowance.initial - _allowance.residual
              : 0;

    _allowance.initial = amount;
    _allowance.residual = spent < amount ? amount - spent : 0;

    Approval(msg.sender, spender, _allowance.residual);

    return true;
}
```

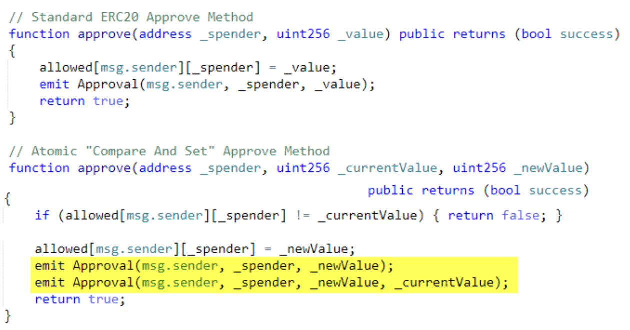
Figure 8. Keeping track of remaining tokens by introducing new data structure.

1) Bob's allowance is initially zero (`allowances[_Alice][_Bob].initial=0`) and his residual is zero as well (`allowances[_Alice][_Bob].residual=0`).
2) Alice allows Bob to transfer N tokens that makes `allowances[_Alice][_Bob].initial=N` and `allowances[_Alice][_Bob].residual=N`.
3) Alice decides to change Bob's allowance to M and has to set it to zero before any non-zero.
4) Bob noticed Alice's transaction for setting his allowance to zero and transfers N tokens in advance. Consequently, `transferFrom` function sets his residual to zero (`allowances[_Alice][_Bob].residual=0`).
5) Alice's transaction for setting Bob's allowance to zero is mined and sets `allowances[_Alice][_Bob].initial=0` and `allowances[_Alice][_Bob].residual=0`. This is similar to step 1 that no token has been transferred. So, Alice would not be able to distinguish whether any token have been transferred or not.
6) Considering no token transfer by Bob, Alice approves Bob for spending new M tokens.
7) Bob is able to transfer new M tokes in addition to initial N tokens.

Someone may think of using `Transfer` event to detect transferred tokens or checking approver balance to see any transferred tokens. As explained in "Enforcement by User Interface (UI)", using `Transfer` event is not sufficient in case of transferring tokens to a third party. Checking approver balance also would not be an accurate way if the contract is busy and there are lot of transfers. So, it would be difficult for the approver to detect legitimate from non-legitimate tokens transfers. Overall, this approach cannot prevent the attack.

## 3.9. Changing ERC20 API

As advised by [19], changing ERC20 API could secure `approve` method by comparing current allowance of the spender and sets it to new value if he has not already been transferred any tokens. This allows atomic compare and set of spender allowance to make the attack impossible. This approach needs a new overloaded `approve` method with three parameters in addition to the standard `approve` method with two parameters (see Figure 9).



```
// Standard ERC20 Approve Method
function approve(address _spender, uint256 _value) public returns (bool success)
{
    allowed[msg.sender][_spender] = _value;
    emit Approval(msg.sender, _spender, _value);
    return true;
}

// Atomic "Compare And Set" Approve Method
function approve(address _spender, uint256 _currentValue, uint256 _newValue)
                                                    public returns (bool success)
{
    if (allowed[msg.sender][_spender] != _currentValue) { return false; }

    allowed[msg.sender][_spender] = _newValue;
    emit Approval(msg.sender, _spender, _newValue);
    emit Approval(msg.sender, _spender, _newValue, _currentValue);
    return true;
}
```

Figure 9. Suggested ERC20 API Change by adding new `approve` method with three parameters to compare and set new allowance atomically.

In order to use this new method, smart contracts have to update their codes to provide three parameters instead of current two, otherwise any `approve` call will use the standard vulnerable version with two parameters. Moreover, one more call is required to read current allowance and pass it to the new `approve` method. Additionally, new event definition needs to be added to ERC20 API to log an approval events with four arguments. For backward compatibility reasons, both three-arguments and new four-arguments events have to be logged. All of these changes makes this token contract incompatible with already deployed smart contracts. Hence, we would not consider it as a sustainable solution.

## 3.10. New token standards

Most of the new token standards are introduced to improve current functionality of ERC20 tokens and do not address the attack (see Table 1). Despite these enhancements, migration from ERC20 to new token standard would not be convenient and all deployed ERC20 tokens—168,092 tokens[8]—have to be redeployed. This also means update of trading platform who already listed ERC20 tokens. Our goal is to find a backward compatible solution instead of changing current ERC20 standard or migrating tokens to a new standards. Despite expanded features and improved security properties of new standards, we would not consider them as target solutions.

## 4. New mitigations

By this point, we have discussed 10 solutions to the multiple withdrawal attack and we evaluated them in terms of compatibility with the standard and attack mitigation (recall the summary in Table 2). Since none of them precisely satisfy the constraints of ERC20 standard, we now propose two new solutions to mitigate the attack.

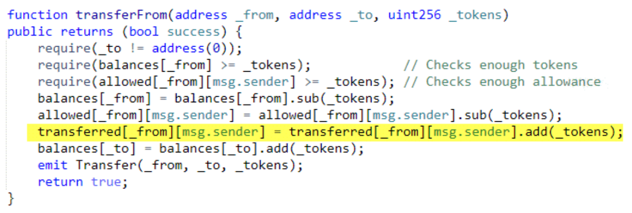
8. https://etherscan.io/tokens, Accessed 18-Feb-2019

| Token Standard | A description of non-compliance with ERC20 |
|---|---|
| ERC 223 [4] | It does not implement ERC20 `approve` + `transferFrom` mechanism by assuming it potentially insecure and inefficient. |
| ERC 667 [5] | It solves the problem of transfer function in ERC223 (i.e., The need to implement `onTokenTransfer` routing in the receiving contract). So, it does address the attack and uses the same code as ERC223 with a supplementary function. |
| ERC 721 [22] | Unlike ERC20 tokens that share the same characteristics, ERC721 tokens represent a non-fungible tokens (NFT) which are unique and non-interchangeable with each other. In addition to this differences, ERC721 does not implement `transfer` method of ERC20 standard and introduces a safe transfer function defined as `safeTransferFrom`. |
| ERC 777 [10] | This standard does not use `transfer` and `transferFrom` methods of ERC20 and implements `send` and `operatorSend` instead. Moreover, costly `approve` + `transferFrom` mechanism is replaced by `tokensReceived` function. Therefore, it would not be backward compatible with ERC20 requirements. Nevertheless a token contract may implement both ERC20 and ERC777 in parallel which still requires attack mitigation. |
| ERC 827 [12] | It uses OpenZeppelin[13] ERC20 implementation and defines three new functions to allow users for transferring data in addition to value in ERC20 transactions. This feature enables ERC20 tokens to have the same functionality as Ether (transferring data and value). In fact, it extends functionality of ERC20 tokens and not addressing the attack. |
| ERC 1155 [23] | It is improved version of ERC721 by allowing each Token ID to represent a new configurable token type, which may have its own metadata, supply and other attributes. ERC1155 aimed to remove the need to "approve" individual token contracts separately. Therefore, it does not implement any code to address the vulnerability |
| ERC 1377 [1] | It implements `approve` method with three parameters in addition to the ERC20 default `approve` with two inputs. Additionally, it uses OpenZeppelin[13] approach for increasing and decreasing approvals. We would consider it as mix of MiniToken and OpenZeppelin approaches that we discussed before. |

TABLE 1. COMPARISON OF ERC STANDARDS IN TERMS OF COMPLIANCY WITH ERC20 SPECIFICATION

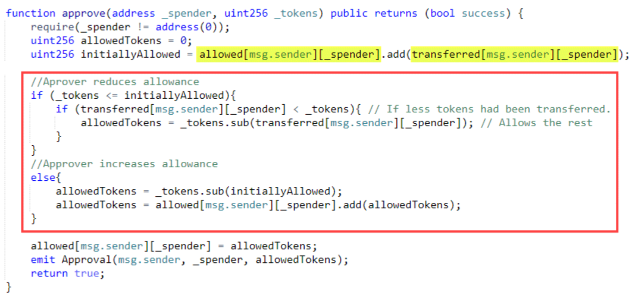
## 4.1. Proposal 1: Securing `approve` method

By implementing CAS [21] in `approve` method, new allowance can be set atomically after comparing with transferred tokens. This tracking requires adding a new variable to `transferFrom` method (see Figure 10). Since this is an internal variable, it is not visible to already deployed smart contracts and keeps the `transferFrom` function compatible. Similarly, a block of code is added to the `approve` function (see Figure 11) to work in both cases with zero and non-zero allowances. Added code to the `approve` function, compares new allowance—passed as `_tokens` argument to the function—with the current allowance of the spender and already transferred tokens—highlighted as `allowed[msg.sender][_spender]` and `transferred[msg.sender][_spender]`. Then it decides to increase or decrease current allowance based on this comparison. If the new allowance is less than initial allowance—sum of `allowance` and `transferred` variables—it denotes decreasing of allowance, otherwise increasing of allowance was intended. Modified `approve` function prevents the attack in either increasing or decreasing of the allowance.



```
function transferFrom(address _from, address _to, uint256 _tokens)
public returns (bool success) {
    require(_to != address(0));
    require(balances[_from] >= _tokens);             // Checks enough tokens
    require(allowed[_from][msg.sender] >= _tokens); // Checks enough allowance
    balances[_from] = balances[_from].sub(_tokens);
    allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_tokens);
    transferred[_from][msg.sender] = transferred[_from][msg.sender].add(_tokens);
    balances[_to] = balances[_to].add(_tokens);
    emit Transfer(_from, _to, _tokens);
    return true;
}
```

Figure 10. Modified version of `transferFrom` for keeping track of transferred tokens per spender.



```
function approve(address _spender, uint256 _tokens) public returns (bool success) {
    require(_spender != address(0));
    uint256 allowedTokens = 0;
    uint256 initiallyAllowed = allowed[msg.sender][_spender].add(transferred[msg.sender][_spender]);

    //Aprover reduces allowance
    if (_tokens <= initiallyAllowed){
        if (transferred[msg.sender][_spender] < _tokens){ // If less tokens had been transferred.
            allowedTokens = _tokens.sub(transferred[msg.sender][_spender]); // Allows the rest
        }
    }
    //Approver increases allowance
    else{
        allowedTokens = _tokens.sub(initiallyAllowed);
        allowedTokens = allowed[msg.sender][_spender].add(allowedTokens);
    }

    allowed[msg.sender][_spender] = allowedTokens;
    emit Approval(msg.sender, _spender, allowedTokens);
    return true;
}
```

Figure 11. Added code block to `approve` function to prevent the attack by comparing and setting new allowance atomically.

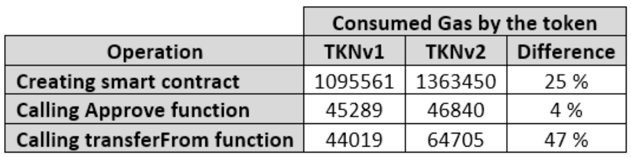
Unlike other solutions, there is no need to set allowance from N to 0 and then to M. Token holder can directly change the allowance from N to M which is saving one transaction accordingly. The following examples make functionality of added codes more clear when decreasing or increasing allowance:

**Scenario A.** Alice approves Bob for spending 100 tokens and then decides to decrease it to 10 tokens.

1) Alice approves Bob for transferring 100 tokens.
2) After a while, Alice decides to reduce Bob's allowance from 100 to 10 tokens.
3) Bob noticed Alice's new transaction and transfers 100 tokens by front-running.
4) Bob's allowance is 0 and `transferred` is 100 (set by `transferFrom` function).
5) Alice's transaction is mined and checks initial allowance (100) with new allowance (10).
6) As it is reducing, `transferred` tokens (100) is compared with new allowance (10). Since Bob already transferred more tokens, his allowance will be set to 0.
7) Bob is not able to move more than initial 100 approved tokens.

**Scenario B.** Alice approves Bob for spending 100 tokens and then decides to increase it to 120 tokens.

1) Alice approves Bob for transferring 100 tokens.
2) After a while, Alice decides to increase Bob's allowance from 100 to 120 tokens.
3) Bob noticed Alice's new transaction and transfers 100 tokens by front-running.
4) Bob's allowance is 0 and `transferred` is 100.
5) Alice's transaction is mined and checks initial allowance (100) with new allowance (120).



| | Consumed Gas by the token | | |
|---|---|---|---|
| **Operation** | **TKNv1** | **TKNv2** | **Difference** |
| **Creating smart contract** | 1095561 | 1363450 | 25 % |
| **Calling Approve function** | 45289 | 46840 | 4 % |
| **Calling transferFrom function** | 44019 | 64705 | 47 % |

Figure 12. Comparison of gas consumption between standard implementation of ERC20 token (TKNv1) and secured implementation of it (TKNv2).

6) As it is increasing, new allowance (120) will be subtracted from transferred tokens (100).
7) 20 tokens will be added to Bob's allowance.
8) Bob would be able to transfer more 20 tokens (120 in total as Alice wanted).

In order to evaluate functionality of the new `approve` and `transferFrom` functions, we have implemented a standard ERC20 token (TKNv1[9]) along side the proposed ERC20 token (TKNv2[10]) on Rinkeby test network. Result of tests for different input values shows that TKNv2 can address multiple withdrawal attack by making front-running gain ineffective. Moreover, we compared these two tokens in term of gas consumption. TokenV2.`approve` function uses almost the same amount of gas as TokenV1.`approve`, however, gas consumption of TokenV2.`transferFrom` is around 47% more than TokenV1.`transferFrom` (see Figure 12). This difference is because of maintaining a new mapping variable for tracking transferred tokens. In term of compatibility, working with standard wallets (*e.g.,* MetaMask) have not raised any transfer issue. This shows compatibility of the token with existing wallets.

In summary, we could use CAS pattern to implement a secure `approve` method that can mitigate the attack effectively. However, it violates one of ERC20 specifications that says "If this function is called again it overwrites the current allowance with `_value`". This implementation of `approve` method adjusts allowance based on transferred tokens. Essentially, it would not be possible to secure the `approve` method without adjusting the allowance. Considering the below scenario:

1) Alice decides to change Bob's allowance from N to M (M less than N in this example).
2) Bob transfers N tokens by front running and `transferred` variable sets to N.
3) Alice's transaction is mined and `approve` method detects token transfer. If `approve` method does not adjust the allowance based on transferred tokens, it has to set it to M—to conform with the standard—which is allowing Bob to transfer more M tokens.

Therefore, `approve` method has to adjust the allowance according to transferred tokens, not based on passed input values to the `approve` method. Overall, there is no solution to secure `approve` method while adhering specification of ERC20 standard.

9. https://rinkeby.etherscan.io/address/0x8825bac68a3f6939c296a40fc8078d18c2f66ac7

10. https://rinkeby.etherscan.io/address/0xf2b34125223ee54dff48f71567d4b2a4a0c9858b

## 4.2. Proposal 2: Securing `transferFrom` method

As an alternative solution to Proposal 1, we can think of securing `transferFrom` method instead of `approve` function. As specified by ERC20 standard (see figure 13), the goal here is to prevent spender from transferring more tokens than allowed. Based on this assumption, we should not consider allowance as the main prevention factor. Transferred tokens can be considered as the main variable in our calculations. For example in the below situation, we can prevent the attack by securing `transferFrom` method and keeping `approve` function untouched to set allowance as specified by the token holder:

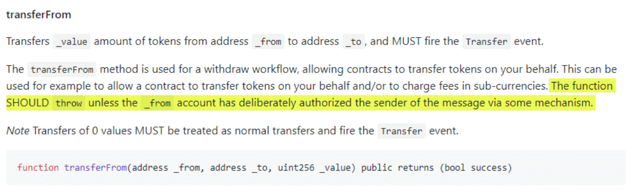
1) Alice allowed Bob for transferring 100 tokens and decides to set it to 70 after a while.
2) Bob front runs Alice's transaction and transfers 100 tokes (legitimate transfer).
3) Alice transaction is mined and sets Bob allowance to 70 by the default `approve` method.
4) Bob noticed new allowance and tries to move new tokens by running `transferFrom(_Bob,70)`. Since he already transferred more than 70, his transaction fails and prevents multiple withdrawal. Additionally, Bob's allowance stays as 70, although transferred tokens shows 100.

Here, `allowance` value can be considered as maximum allowance. It indicates that Bob is eligible to transfer up to specified limit if he has not already transferred any tokens. This impression is completely in accordance with ERC20 standard (see figure 13). In fact, there is no relation between allowance (`allowed[_from][msg.sender]`) and transferred tokens (`transferred[_from][msg.sender]`). The first variable shows maximum transferable tokens by a spender and can be changed irrelative to transferred tokens (*i.e.,* `approve` method does not check transferred tokens). If Bob has not already transferred that much of tokens, he would be able to transfer difference of it—`allowed[_from][msg.sender].sub(transferred[_from][msg.sender])`. In other words, `transferred` variable is a life time variable that accumulates transferred tokens regardless of allowance changes. This token is implemented as TKNv3[11] on Rinkby network and passed compatibility checks by transferring tokens between standard wallets. In terms of gas consumption, `transferFrom` function needs at about 37% more gas than standard `transferFrom` implementation which is acceptable for having a secure ERC20 token.

## 5. Conclusion

ERC20 token holders should be aware of their approval consequences. If they approve someone to transfer N tokens on behalf of them, the spender can transfer exactly N tokens, even if they reduce the allowance afterwards. This is considered as a legitimate transaction and

11. https://rinkeby.etherscan.io/address/0x5d148c948c01e1a61e280c8b2ac39fd49ee6d9c6



**transferFrom**

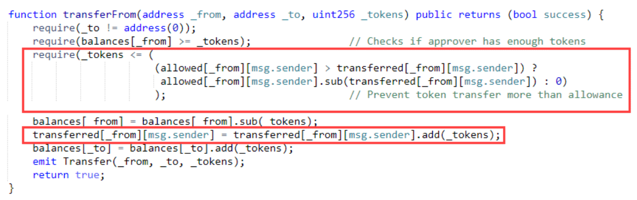
Transfers `_value` amount of tokens from address `_from` to address `_to`, and MUST fire the `Transfer` event.

The `transferFrom` method is used for a withdraw workflow, allowing contracts to transfer tokens on your behalf. This can be used for example to allow a contract to transfer tokens on your behalf and/or to charge fees in sub-currencies. The function SHOULD `throw` unless the `_from` account has deliberately authorized the sender of the message via some mechanism.

*Note* Transfers of 0 values MUST be treated as normal transfers and fire the `Transfer` event.

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

Figure 13. ERC20 `transferFrom` method definition that emphasizes on throwing an exception when the spender is not authorized to move tokens.



```
function transferFrom(address _from, address _to, uint256 _tokens) public returns (bool success) {
    require(_to != address(0));
    require(balances[_from] >= _tokens);                // Checks if approver has enough tokens
    require(_tokens <= (
                    (allowed[_from][msg.sender] > transferred[_from][msg.sender]) ?
                     allowed[_from][msg.sender].sub(transferred[_from][msg.sender]) : 0)
                    );                                  // Prevent token transfer more than allowance

    balances[_from] = balances[_from].sub(_tokens);
    transferred[_from][msg.sender] = transferred[_from][msg.sender].add(_tokens);
    balances[_to] = balances[_to].add(_tokens);
    emit Transfer(_from, _to, _tokens);
    return true;
}
```

Figure 14. Securing `transferFrom` method instead of `approve` method can mitigate the attack by preventing more token transfer than allowed.

responsibility of the approver before allowing the spender for transferring any tokens. Multiple withdrawal attack can occur when allowance changes from N to M, that allows a spender to transfer N+M tokens in total by front running of `approve` method. In this paper, we examined 10 suggested solutions and explained their violation constraints. Since all approaches violated ERC20 specifications or could not address the attack effectively, we introduced two new proposals for securing vulnerable methods—`approve` and `transferFrom`. Proposal 1 incorporates CAS pattern for comparing and setting new allowances atomically. It mitigates the attack by adjusting new allowance based on transferred tokens. This allowance adjustment violates one of ERC20 constraints which requires allowance set to new value instead of adjusting it. We concluded that securing `approve` method is not feasible while adhering this standard constraint. Therefore, we secured `transferFrom` function instead of `approve` method in the second proposal. Proposal 2 mitigates the attack effectively and is backward compatible with already deployed smart contracts. Although it consumes more gas compared to standard ERC20 implementations, but it is not vulnerable and could be considered for future secure ERC20 token deployments.

## References

[1] J. C. Atkins Chang, Noel Bao, L. Chou, and D. Goh. Service-Friendly Token Standard. https://github.com/fstnetwork/EIPs/blob/master/EIPS/eip-1376.md, Sept. 2018. [Online; accessed 12-Jan-2019].

[2] E. Chavez. StandardToken.sol. https://github.com/kindads/erc20-token/blob/40d796627a2edd6387bdeb9df71a8209367a7ee9/contracts/zeppelin-solidity/contracts/token/StandardToken.sol, Mar. 2018. [Online; accessed 23-Dec-2018].

[3] ConsenSys. ConsenSys/Tokens. https://github.com/ConsenSys/Tokens/blob/fdf687c69d998266a95f15216b1955a4965a0a6d/contracts/eip20/EIP20.sol, Apr. 2018. [Online; accessed 24-Dec-2018].

[4] Dexaran. ERC223 token standard. https://github.com/ethereum/EIPs/issues/223, Mar. 2017. [Online; accessed 12-Jan-2019].

[5] S. Ellis. transferAndCall Token Standard. https://github.com/ethereum/EIPs/issues/677, July 2017. [Online; accessed 12-Jan-2019].

[6] S. Eskandari, S. Moosavi, and J. Clark. Sok: Transparent dishonesty: front-running attacks on blockchain. 2019.

[7] Ethereum. Ethereum project repository. https://github.com/ethereum, May 2014. [Online; accessed 10-Nov-2018].

[8] V. B. Fabian Vogelsteller. ERC-20 Token Standard. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md, Nov. 2015. [Online; accessed 2-Dec-2018].

[9] T. Hale. Resolution on the EIP20 API Approve / TransferFrom multiple withdrawal attack #738. https://github.com/ethereum/EIPs/issues/738, Oct. 2017. [Online; accessed 5-Dec-2018].

[10] J. B. Jacques Dafflon and T. Shababi. EIP 777: A New Advanced Token Standard. https://eips.ethereum.org/EIPS/eip-777, Nov. 2017. [Online; accessed 12-Jan-2019].

[11] D. N. Jordi Baylina and sophiii. minime/contracts/MiniMeToken.sol. https://github.com/Giveth/minime/blob/master/contracts/MiniMeToken.sol#L225, Dec. 2017. [Online; accessed 23-Dec-2018].

[12] A. Lemble. ERC827 Token Standard (ERC20 Extension). https://github.com/ethereum/eips/issues/827, Jan. 2018. [Online; accessed 12-Jan-2019].

[13] OpenZeppelin. openzeppelin-solidity. https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/token/ERC20/ERC20.sol, Dec. 2018. [Online; accessed 23-Dec-2018].

[14] outofgas. outofgas comment. https://github.com/ethereum/EIPs/issues/738#issuecomment-373935913, Mar. 2018. [Online; accessed 25-Dec-2018].

[15] E. Project. Create your own crypto-currency). https://www.ethereum.org/token, Dec. 2017. [Online; accessed 01-Dec-2018].

[16] P. Vessenes. MonolithDAO/token. https://github.com/MonolithDAO/token/blob/master/src/Token.sol, Apr. 2017. [Online; accessed 23-Dec-2018].

[17] M. Vladimirov. Attack vector on ERC20 API (approve/transferFrom methods) and suggested improvements. https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729, Nov. 2016. [Online; accessed 18-Dec-2018].

[18] M. Vladimirov. Implementation of 'approve' method violates ERC20 standard #438. https://github.com/OpenZeppelin/openzeppelin-solidity/issues/438#issuecomment-329172399, Sept. 2017. [Online; accessed 24-Dec-2018].

[19] M. Vladimirov and D. Khovratovich. ERC20 API: An Attack Vector on Approve/TransferFrom Methods. https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM/edit#heading=h.m9fhqynw2xvt, Nov. 2016. [Online; accessed 25-Nov-2018].

[20] N. Welch. flygoing/BackwardsCompatibleApprove.sol. https://gist.github.com/flygoing/2956f0d3b5e662a44b83b8e4bec6cca6, Feb. 2018. [Online; accessed 23-Dec-2018].

[21] Wikipedia. Compare-and-swap. https://en.wikipedia.org/wiki/Compare-and-swap, July 2018. [Online; accessed 10-Dec-2018].

[22] J. E. William Entriken, Dieter Shirley and N. Sachs. ERC-721 Non-Fungible Token Standard. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md, Jan. 2018. [Online; accessed 12-Jan-2019].

[23] P. C. Witek Radomski, Andrew Cooke, J. Therien, and E. Binet. ERC-1155 Multi Token Standard. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1155.md, June 2018. [Online; accessed 12-Jan-2019].