A solution for Multiple Withdrawal Attack in ERC20 token

***Abstract*—ERC20 standard defines set of interfaces for standardizing interaction with tokens on the Ethereum blockchain. Tokens in Ethereum ecosystem facilitate creation of digital assets by introducing standard functions that can be reused by ERC20-compliant applications. Being as a subset of smart contracts, makes them vulnerable to security flaws. Particularly, two functions in ERC20 standard allow token transfer on behalf of the owner. Using these two functions in case of front-running could lead to "Multiple Withdrawal Attack" that allows a spender to transfer more tokens than the owner ever wanted. This standard-level issue was initially raised on Github and may even impact security of already deployed tokens. Openness of the issue since October 2017, motivated us to (1) examine ten suggested solutions in accordance with specifications of the standard and being backward compatible with already deployed smart contracts; (2) propose a new solution that mitigates the attack sustainably.**

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

1. Introduction

Ethereum project[1] launched in 2014 and ranked second in terms of market value after Bitcoin.[1] It has the biggest development community to track enhancement and propose new ideas.[2] Ethereum offers an ecosystem to implement decentralized application on the blockchain. Tokens are essential part of this ecosystem which define set of rules—known as API[3]—for standardizing interaction with smart contracts.[4]. Therefore, any ERC20[5]-compliant application can interact with any ERC20 token for trading, swapping, exchanging, etc. For example, shares of company X can be represented as ERC20 tokens to be reusable by other smart contracts (e.g., online exchanges, automated payment systems, decentralized games, etc). Leveraging ERC20 token facilitates implementation of financial assets while raising some security concerns. ERC20 tokens are technically standardized version of smart contracts that could be similarly vulnerable to security flaws.

[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

[3]Advanced Programming Interface.

[4]Types of transactions that execute as they are programmed by Ethereum scripting language (e.g., Solidity or Vyper)

[5]ERC20 is title of the standard and it should be referred as EIP20 (which is the actual proposal for improvement). In this paper we use both ERC20 and EIP20 in one sense for simplicity.

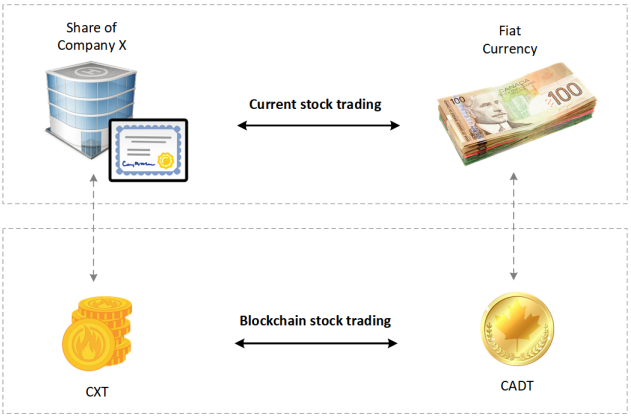


Figure 1. Importance of ERC20 security in case of (1) digitizing share of company X (2) representing a fiat currency. In this case, smart contracts interact with two different ERC20 tokens which are repressing in sequence, a financial instrument and a non-collateralized stablecoin. Any security vulnerability in written code of ERC20 tokens, will impact security of related smart contracts and value of underlying assets.

Since introduction of ERC20 standard in November 2015, several vulnerabilities have been discovered and addressed by the Ethereum community. In October 2017, a new security issue opened on GitHub[2], [3] and known as "Multiple Withdrawal Attack". The attack originating from definition of two interfaces[6] in ERC20 standard for approving and transferring tokens. Using these functions in an undesirable situation (e.g., front-running) could result in conditions that tokens being spent by another third party on behalf of the owner. This issue is still open and several solutions have been made to mitigate it. In this paper we examine 10 suggested solutions in terms of compatibility with the standard and attack mitigation. Since none of them could satisfy constraints of ERC20 standard, we motivated to introduce a new solution to mitigate the attack.

Authors of ERC20 standard [4], provided generic implementations of ERC20 tokens from OpenZeppelin[5] and ConsenSys[6]. OpenZeppelin uses two additional methods and ConsenSys has not attempted to work around the issue. There are other implementations that have different trade-offs. We compared them in the below table to examine whether they violate constraints of the standard or not. Since the attack happens in the event of gap between transactions, we used compare and set (CAS) pattern[15] to cover this gap. CAS is one of the most widely used lock-free synchronization strategy that allows

[6]Interface defines parameters and expected outputs of each function without implementing them. Developers are free to write arbitrary codes that could potentially lead to a security issue.

comparing and setting values in an atomic way. It allows to compare values in one transaction and set new values before transferring control to another one. We leveraged this pattern to remove the gap between transactions and prevent race condition as root cause of the attack.

Two proposals are introduced by this paper to secure one of two vulnerable methods (i.e., either *approve* or *transferFrom*). The first one, mitigates the attack by comparing transferred tokens with the current allowance in the *approve* method. The second proposal secures *transferFrom* method by not allowing token transfers more than allowed.

#	Proposed solution	Is it backward compatible ?	Does it secure 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 still occur
2	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
3	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
4	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 it needs two calls	✓ By default transferFrom method	✗ Race condition can still occur
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	Detecting token transfers	✓ It adds two lines to approve method	✓	✗ It locks allowance in case of any token transfer	✓ By default transferFrom method	✓ By blocking legit and non-legit allowances
8	Keeping track of remaining tokens	✓ It adds three lines to the approve method	✓	✓	✓ By default transferFrom method	✗ Race condition can still occur
9	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
10	New token standard	✗ It introduces new API	✓	✓	✓	✓
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. Proposal 2 uses CAS pattern to mitigate the attack sustainably by (1) comparing transferred tokens through a new local variable in *transferFrom* function and (2) tracking of transferred tokens to prevent more transfers.

2. Background

2.1. How attack could happen

The attack could happen in case of race condition[7]. It allows a spender to transfer more tokens than the owner ever wanted. An attacker can execute *transferFrom* function two times by front-running of *approve* method and transferring more token than authorized. According to ERC20 standard:

(a) *approve*[8] function allows *_spender* to withdraw up to the *_value* amount of tokens from token pool of the approver. If this function is called again, it has to overwrites the current allowance with the new *_value*.

(b) *transferFrom*[9] function grants required rights to the spender (account, wallet or other smart contract) for transferring *_value* amount of tokens from address *_from* to address *_to*.

Attacker can take advantage of gap between execution of *approve* and *transferFrom* functions since the *approve* method overrides current allowance regardless of whether spender already transferred any tokens or not. Moreover, transferred tokens are not trackable and only *Transfer*[10] event will be logged (which is not sufficient in case of transferring tokens to a third parity). Here could be a possible attack scenario:

1- Alice allows Bob to transfer N tokens on her behalf by calling *approve(_Bob, N)*.
2- After a while, Alice decides to change Bob's approval from N to M by executing *approve(_Bob, M)*.
3- Bob notices Alice's second transaction before it was mined and quickly sends another transaction that runs *transferFrom(_Alice, _Bob, 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 techniques) 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 before and gains ability of transferring another M tokens.
7- Before Alice notices that something went wrong, Bob calls *transferFrom* method for the second time and transfers M Alice's tokens by executing *transferFrom(_Alice, _Bob, M)*.

In fact, Alice attempted to change Bob's allowance from N to M, but she made it possible for Bob to transfer N+M of her tokens at most, while Alice never wanted to allow so many transfers to be occurred by Bob. It should be noted that the assumption here is to prevent Bob from withdrawing Alice's tokens multiple times when allowance changes from N to M. If Bob could withdraw N tokens after Alice initial approval, this would be considered as legitimate transfer, since Alice

[7] Execution of two transactions at the same time with undesirable situation or priority.

[8] approve(address *_spender*, uint256 *_tokens*)

[9] transferFrom(address *_from*, address *_to*, uint256 *_tokens*)

[10] Transfer(address indexed *_from*, address indexed *_to*, uint256 *_value*)

has already approved it. In other words, it would be responsibility of Alice to make sure before approving anything to Bob. After that, Bob is allowed to transfer up to N tokens even right before allowance change from N to M. Accordingly, transaction #4 in the below diagram would be considered as multiple withdrawal attack since Bob is able to move more M tokens in addition to already transferred N tokens in step #3.

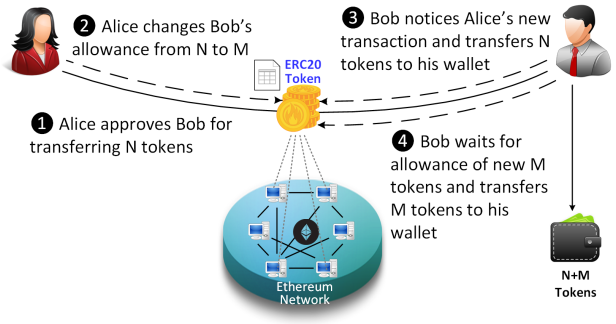


Figure 3. Possible multiple withdrawal attack in ERC20 tokens when Alice changes Bob's allowance from N to M. By front-running, Bob is able to move N+M tokens from Alice's token pool.

2.2. How to prevent the attack

Sustainable solutions have to prevent Bob from multiple withdrawal (N+M) presuming that Alice has more than N+M tokens in her wallet. One of these approaches can be implemented to prevent the attack:

A. **Prevention by owner:** This approach is recommended by ERC20 standard [4] and advises owners to change spender allowance from N to 0 and then from 0 to M (instead of set it directly from N to M). Changing allowance to non-zero values after setting to zero will not prevent the attack since the owner would not be able to distinguish how the allowance is set to zero. Was it because of her previous *approve* transaction for changing allowance from N to 0?, Or it was set to 0 by *transferFrom* method due to token transfers? Although It would be possible to track transferred token through *Transfer* events—logged by *transferFrom* method—tracking of tokens would not be easy in case of transferring to a third-party. For example, if Alice allows Bob and then Bob transfers tokens to Carole, *Transfer* event creates a log showing Carole moved tokens from Alice. As discussed in "MiniMeToken" solution, this approach can not prevent the attack despite being recommended by the standard.

B. **Prevention by *approve* method:** In this approach, *approve* method changes spender allowance from N to M atomically by using compare and set (CAS) pattern [15]. Comparison part of CAS requires knowledge of transferred tokens that reveals any transferred tokens in case of front-running. Hence, we can compare new allowance with transferred token and set it accordingly. Setting new allowance in *approve* method must satisfy ERC20 constraint that says "If this function is called again it overwrites the current allowance with *_value*" [4]. This constraint prohibits any adjustments in allowance which makes the *approve* method vulnerable. For example, considering front-running by Bob when Alice wants to change Bob allowance form 100 to 110, the *approve* method can reveal 100 transferred tokens by Bob. However, based on ERC20 constraints, it must not adjust new allowance to 110-100=10, it has to set it to 110, which is allowing Bob for transferring 100+110=210 tokens in total. We implemented this approach in proposal 1 and as analyzed, securing *approve* method can not prevent the attack while adhering constraints of the standard. The only feasible approach would be prevention it when transferring tokens.

C. **Prevention by *transferFrom* method:** This approach is based on ERC20 constraint that says "approve functions allows *_spender* to withdraw from your account multiple times, up to the *_value*". So, spender must not be able to transfer more than authorized tokens. That being said, *transferFrom* method can be secured in a way that prevents M new token transfer in case of already transferred N tokens. By comparing transferred tokens in *transferFrom* method, spender will be restricted for moving solely remained tokens. In case of trying to transfer token more than allowed, the transaction fails. For example, Alice's new transaction for increasing Bob allowance from 100 to 110, sets Bob allowance to 110. However, *transferFrom* method does not allow Bob from transferring more than 10 tokens. We implanted this approach in proposal 2 and it prevents the attack effectively.

2.3. What are violation constraints

An important criterion for a solution is to adhere the specifications of ERC20 standard. Conforming with the standard, keeps new tokens backward compatible with already deployed smart contracts. So, smart contracts can interact with tokens as defined in the standard without raising any exception. We extracted defined constraints from ERC20 specifications [4] that must be satisfied by any sustainable solution:

1- Calling *approve* function has to overwrite current allowance with new allowance.
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- Introducing new methods violates ERC20 API and it has to be avoided for having compatible token with already deployed smart contracts.
5- Spender will be allowed to withdraw from approver account multiple times, up to the allowed amount.
6- Transferring initial allowed tokens is considered as legitimate transfer. It could happen right after approval or before changing allowance.
7- Race condition MUST not happen in any cases to prevent multiple withdrawal attack account.

3. Examination of solutions

Several solutions have been proposed by Ethereum community (mostly from developers on GitHub) to address this attack. There would be some considerations for each solution that needs to be evaluated in term of compatibility with ERC20 standard and attack mitigation. We have examined technical aspects of each solution in the following sections.

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:

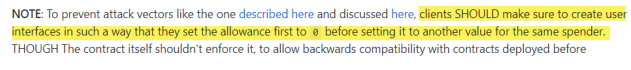


Figure 4. Recommendation of ERC20 standard to mitigate multiple withdrawal attack.

However, if Alice does not use UI and connects directly to the blockchain, there would be a good chance of impacting by this attack. Furthermore, as discussed on Github[8], this approach is not sufficient 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 0.
3- Bob front runs Alice's transaction and transfers N Alice's tokens (*transferFrom* sets Bob's allowance to 0).
4- Alice's transaction is mined and Bob's allowance is set to 0 by *approve* method. This is exactly what she would see if Bob would not transfer any tokens, so she has no reason to think that Bob actually used his allowance before it was revoked.
5- Now Alice publishes a new transaction that changes Bob's allowance to M.
6- Alice's second transaction is mined, Bob now is allowed to transfer 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. This is a legitimate transaction since Alice has already approved it. The issue occurs after Alice's new transaction to set Bob's allowance to 0. 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 will find 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 or Bob already transferred token on her behalf. 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. Additionally, There is no way to see from UI if setting Bob's allowance to 0 is processed before the subsequent non-zero approval [9]. This is because of current methods in Web3.js[11] that do not support such checking[10]. Hence, enforcement should be considered at contract level not UI.[12]

3.2. Using Minimum viable token

As suggested by[10], we can boil down ERC20 standard to a very basic functionalities by implementing only essential methods. this will prevent 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[13]) is one of these examples since it does not implement the *approve*, *allowance* and *transferFrom* functions. According to ERC20 specifications, these methods are not OPTIONAL and must be implemented. Moreover, ignoring them will cause failed function calls from standard smart contracts that expect to interact with these methods. Therefore, we would not consider this solution as a compatible fix although mitigates the attack.

3.3. Approving trusted parties

Approving token transfer to non-upgradable smart contracts can be considered safe. Because they do not contain any logic to take advantage of this vulnerability. However, upgradable smart contracts may add new logic to a new version that needs re-verification before approving token transfer. 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 mitigation for the attack.

3.4. MiniMeToken solution

MiniMeToken[11] also follows ERC20 recommendation by reducing allowance to zero before non-zero values. They added a line of code to the *approve* method. The red clause allows setting approval to 0 and blue condition checks allowance of *_spender* to be 0 before setting to other values (i.e., If *_spender* allowance is 0 then allows non-zero values):

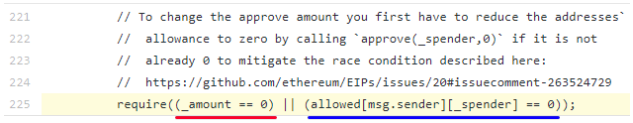


Figure 5. MiniMeToken suggestion for adding new codes to approve method.

[11] JavaScript UI library for interacting with Ethereum blockchain

[12] Interestingly, OpenZeppelin example implements a workaround in contract level that makes it inconsistent with the recommendations of ERC20.

[13] https://etherscan.io/address/0xa74476443119A942dE498590Fe1f2454d7D4aC0d#code

Similar to "Enforcement by User Interface (UI)", this 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. It is more clear in this 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- By considering that Bob has not been transferred any tokens, Alice allows Bob for transferring new M tokens.
6- Bob would be able to transfer new approved tokens.

3.5. MonolithDAO solution

MonolithDAO[7] Token suggests two additional functions for increasing or decreasing allowance. *approve* function will also have an additional code to set allowance to zero before non-zero values. In this case, the default *approve* function should be called when spender's allowance is zero (No approval has been made). If spender's allowance is non-zero, increase and decrease functions will be used:

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 *approve* method with new added code in MonolithDAO token.

These two functions can address race condition and prevent allowance double-spend exploit:

1- Alice allows Bob to transfer N tokens by calling *approve(_Bob, N)*. This will be executed by *approve* function since current Bob's allowance is 0.
2- After a while, Alice decides to decrease Bob's approval by M by running *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 functions will prevent the attack, they have not been defined in the initial specifications of ERC20. Therefore, they can not be used by smart contracts that are already deployed on the Ethereum network since they will still use 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 defines new allowance. 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 20 (100 - 80 = 20). Comparatively, increase method will set new allowance to 180 while it has to set to 80 again. 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[12] is to move security checks to another function like *safeApprove* that sets allowance if it has not been already changed. By using this function, Alice uses the standard *approve* function to set Bob's allowance to 0 and for new approvals, she has to use *safeApprove*. It takes the current expected approval amount as input parameter and calls *approve* method if previous allowance is equal to current expected approval. So, Alice will have one step more and it is reading the current allowance and passing it to the new *safeApprove* method. As mentioned in the last section, this approach is not backward compatible with already implemented smart contracts. The new *safeApprove* method that is not defined in ERC20 standard and existing code would not be able to use this safety feature.

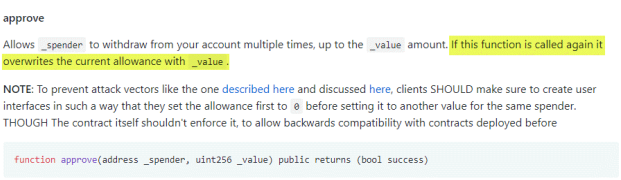
3.7. Detecting token transfers

As suggested by [13],a boolean variable is used to detect whether any tokens have been transferred or not. *transferFrom* method sets a flag to true if tokens are transferred. *approve* method checks the flag to be false before allowing new approvals (i.e., it checks if tokens have been used/transferred since the owner last allowance set). Moreover, it uses a new data structure for keeping track of used/transferred tokens. This approach could prevent race condition as described below:

1- Alice runs *approve(_Bob, N)* to allow Bob for transferring N tokens.
2- Since Bob's initial allowance is 0 and used flag is false, then sanity check passes and Bob's allowance is set to N.
3- Alice decides to set Bob's allowance to 0 by executing *approve(_Bob, 0)*.
4- Bob front-runs Alice's transaction and transfers N tokens. Then, his used flag turns to *true*.
5- Alice's transaction is mined and passes sanity check (because *_value == 0*).
6- Bob's allowance is set to 0 while used flag is still *true*.
7- Alice changes Bob's allowance to M by executing *approve(_Bob, M)*.

8- Since Bob already transferred number of tokens, used flag is *true* and it fails the transaction.
9- Bob's allowance remains as N and he could transfer only N tokens.

Although this approach mitigates the attack, 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 (even 1 token), Alice would not be able to change his approval. Because used flag is *true* now and does not allow changing allowance to any non-zero values. Even setting the allowance to 0, does not flip used flag and keeps Bob's allowance locked down. In fact, the code needs a line like *allowed[_from][msg.sender].used = false;* in *approve* method. But it will cause another problem. After setting allowance to 0, used flag becomes *false* and allows non-zero values event if tokens have been already transferred. In other words, it resembles the initial values of allowance similar when nothing is transferred. Therefore, it makes attack mitigation functionality ineffective. In short, this approach can not satisfy both legitimate and non-legitimate scenarios and violets ERC20 standard that says:


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 7. ERC20 approve method constraint.

Nevertheless, this solution is a step forward by introducing the need for a new variable to track transferred tokens.

3.8. Keeping track of remaining tokens

Inspired by by the previous solution, [14] keeping track of remaining tokens instead of detecting transferred tokens. It uses modified version of data structure that used in the previous solution for storing residual tokens:

```
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;
}

function allowance(address holder, address spender) public view returns (uint) {
    return allowances[holder][spender].residual;
}

function transferFrom(address holder, uint amount) public returns (bool) {
    uint residual = allowance(holder, msg.sender);

    require(amount <= residual);

    allowances[holder][msg.sender].residual = residual - amount;

    // ... do the token transfer

    return true;
}
```

Figure 8. Keeping track of remaining tokens.

At first, it seems that this solution is a sustainable way to mitigate the attack by setting approval to zero before non-zero values. However, the highlighted code resembles the situation that we explained in "Enforcement by User Interface (UI)":

1- Bob's allowance is initially zero (*allowances[_Alice][_Bob].initial=0*, *allowances[msg.sender][spender].residual=0*).
2- Alice allows Bob to transfer N tokens (*allowances[_Alice][_Bob].initial=N*, *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 values.
4- Bob noticed Alice's transaction for setting his allowance to zero and transfers N tokens in advance. transferFrom sets his allowance (residual) to zero consequently (*allowances[_Alice][_Bob].residual=0*).
5- Alice's transaction is mined and sets *allowances[_Alice][_Bob].initial=0* and *allowances[msg.sender][spender].residual=0* (Similar to step 1). This is like that no token has been transferred. So, Alice would not be able to distinguish whether any token have been transferred or not.
6- 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.

3.9. Changing ERC20 API

[9] advised to change ERC20 approve method to compare current allowance of spender and sets it to new value if it has not already been transferred. This allows atomic compare and set of spender allowance to make the attack impossible. So, it will need new overloaded approve method with three parameters:

```
// 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 for approve method.

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 throw an exception. Moreover, one more call is required to read current allowance value and pass it to the new approve method. New events need to be added to ERC20 specification 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 deployed smart contracts and software wallets. Hence, it could not be considered as viable solution.

3.10. New token standards

After recognition of this security vulnerability, new standards like ERC233 [14] and ERC721[15] were introduced to address the issue in addition to improving current functionality of ERC20 standard. They changed approval model and fixed some drawbacks which need to be addressed in ERC20 as well (i.e., handle incoming transactions through a receiver contract, lost of funds in case of calling transfer instead of transferFrom, etc). Nevertheless, migration from ERC20 to ERC223/ERC721 would not be convenient and all deployed tokens needs to be redeployed. This also means update of any trading platform listing ERC20 tokens. The goal here is to find a backward compatible solution instead of changing current ERC20 standard or migrating tokens to new standards. Despite expanded features and improved security properties of new standards, we would not consider them as target solutions.

[14]https://github.com/Dexaran/ERC223-token-standard

[15]https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md

4. Solution proposal

4.1. Proposal 1: Securing *approve* method

As discussed, a feasible solution would be to use CAS pattern to set new allowance atomically. This needs knowledge of transferred tokens that requires adding a new mapping variable to token code. The code would be still compatible with other smart contracts due to internal usage of the variable. Consequently, *transferFrom* method will have an new line of code for tracking transferred tokens:

```
// ------------------------------------------------------------------------
// Special type of Transfer and make it possible to give permission to another address
// to spend token on your behalf.
// It sends _tokens amount of tokens from address _from to address _to
// The transferFrom method is used for a withdraw workflow, allowing contracts to send
// tokens on your behalf, for example to "deposit" to a contract address and/or to charge
// fees in sub-currencies; the command should fail unless the _from account has
// deliberately authorized the sender of the message via some mechanism
// ------------------------------------------------------------------------
function transferFrom(address _from, address _to, uint256 _tokens) public returns (bool success) {
    require(_to != address(0));
    require(balances[_from] >= _tokens);                    // Check if approver has enough tokens
    require(allowed[_from][msg.sender] >= _tokens);      // Check 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* based on added mapping variable.

Similarly, a block of code will be added to the *approve* function to compare new allowance with transferred tokens. It has to work in both cases with zero and non-zero allowance set:

```
// ------------------------------------------------------------------------
// It helps to give permission to another address to spend tokens on your behalf.
// Allow _spender to withdraw from your account, multiple times, up to the _tokens amount.
// If this function is called again it overwrites the current allowance with _tokens.
// ------------------------------------------------------------------------
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 _spender for 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 compare and set new allowance value.

Added code to *approve* function will compare new allowance (*_tokens*) with current allowance of the spender (allowed[msg.sender][_spender]) and with already transferred token (transferred[msg.sender][_spender]). Then it decides to increase or decrease current allowance. If new allowance is less than initial allowance (sum of *allowance* and *transferred* variables), it denotes decreasing allowance, otherwise increasing allowance was intended. For example, we consider these scenarios:

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) will be compared with new allowance (10).
7- Since Bob already transferred more tokens, his allowance will be set to 0.
8- Bob is not able to move more than initial approved tokens.

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).
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*/*transferFrom* functions, we have implemented a standard ERC20 token (TKNv1[16]) along side proposed ERC20 token (TKNv2[17]). 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 50% more than TokenV1.*transferFrom*. This difference is because of maintaining a new mapping variable for tracking transferred tokens:

	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 TKNv1 and TKNv2.

In term of compatibly, working with standard wallets (like MetaMask) have not raised any transfer issue. This shows compatibility of the token with existing wallets.

[16]https://rinkeby.etherscan.io/address/0x8825bac68a3f6939c296a40fc8078d18c2f664c7

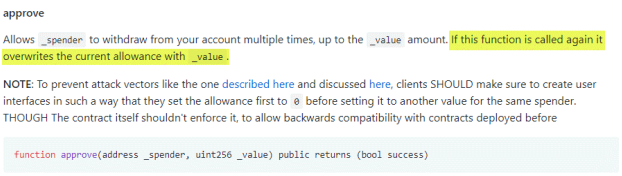
[17]https://rinkeby.etherscan.io/address/0xf2b34125223ee54dff48f71567d4b2a4a0d98d8b

4.2. Proposal 2: Securing *transferFrom* method

Proposal 1 mitigates the attack in all situations, however it adjusts allowance based on transferred tokens. For example, if Alice allowed Bob for transferring 100 tokens and she decides to increase it to 120 tokens, the allowance will not directly set to 120 and the code adjusts it as below:

- If Bob already transferred 100 tokens, the new allowance will be 20 (100+20 = 120).
- If Bob already transferred 70 tokens, the new allowance will be 50 (70+50 = 120).
- If Bob has not already transferred any tokens, the new allowance will be 120 (0+120=120).

Although the final result will be the same and does not allow Bob to transfer more than intended tokens, but ERC20 standard approve method emphasizes that:


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

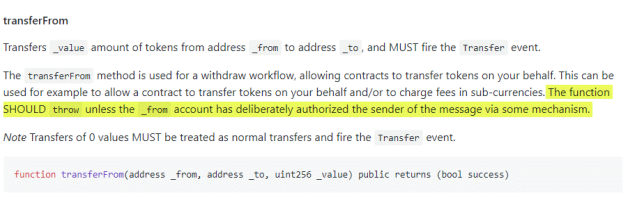
```
function approve(address _spender, uint256 _value) public returns (bool success)
```

Figure 13. ERC20 *approve* method constraint.

Hence, adjusting allowance will violate this constraint. On the other hand, this was the only solution for improving approve method. Because setting allowance securely to new values would need knowledge of transferred tokens. We can get this knowledge by:

1- Tracking what have been transferred and ADJUST allowance accordingly.
2- Passing a new input parameter that is showing what was the allowance before.

We implemented the first approach and the second one would need to modify definition of *approve* method. It seems that there is no feasible implementation to satisfy constraints of ERC20 and mitigating the attack under one solution. Therefore, we would assume API change as final solution of securing *approve* method. As an alternative solution, we can think of securing *transferFrom* method instead of *approve* method. ERC20 standard emphasizes that:


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 14. ERC20 *transferFrom* method constraint.

So, the goal is to prevent spender from transferring more tokens than allowed by the approve. Based on this impression, we should not consider allowance as the main factor. Transferred tokens should be considered as the variable in calculations. For example:

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's transaction is mined and sets Bob allowance to 80.
4- Bob got new allowance and runs *transferFrom(_Bob,80)*. Since he already transferred more than 80, his transaction will fail and prevent multiple withdrawal.
5- Bob's allowance stays as 80, however, he can not use it.

Here allowance 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. In fact, there is no relation between allowance (*allowed[_from][msg.sender]*) and transferred tokens (*transferred[_from][msg.sender]*). The fist variable shows maximum transferable tokens by a spender and can be changed irrelative to transferred tokens (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 is life time variable that accumulates transferred tokens regardless of allowance change. So, by this assumption, we can secure *transferFrom* method instead of *approve* method as below:

```
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 15. Securing *transferFrom* method instead of *approve* method.

This token is implemented as TKNv3 [18] on Rinkby network. Gas consumption of *transferFrom* function is around 37% more than standard implementation which is acceptable for having a secure ERC20 token.

5. Conclusion

Based on ERC20 specifications, token owners should be aware of their approval consequences. If they approve someone to transfer N tokens, the spender can transfer exactly N tokens, even if they change allowance to zero afterward. This is considered a legitimate transaction and responsibility of approver before allowing the spender for transferring any tokens. An attack can happen when changing allowance from N to M, that allows spender to transfer N+M tokens and effect multiple withdrawal attack. This attack is possible in case of front-running by approved side. As we examined possible solutions, all approaches violate ERC20 specifications or have not addressed the attack completely. Proposal 1 uses CAS pattern for checking and setting new allowance atomically. In proposal 2, *transferFrom* function is secured instead of *approve* method. We implemented an ERC20 token for each proposal that solve this security issue while keeping backward compatibly with already deployed smart contracts or wallets. Although these implementations consume more Gas than standard ERC20 implementations, they are secure and could be considered for secure ERC20 token deployment.

[18] https://rinkeby.etherscan.io/address/0x5d148c948c01e1a61e280c8b2ac39fd49ee6d9cd

References

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

[2] 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].

[3] 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].

[4] 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].

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

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

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

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

[9] 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].

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

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

[13] N. Welch, "flygoing/BackwardsCompatibleApprove.sol," https://https://gist.github.com/flygoing/2956f0d3b5e662a44b83b8e4bec6cca6, Feb. 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] Wikipedia, "Compare-and-swap," https://en.wikipedia.org/wiki/Compare-and-swap, Jul. 2018, [Online; accessed 10-Dec-2018].