Resolving the Multiple Withdrawal Attack in ERC20 Tokens

***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.**

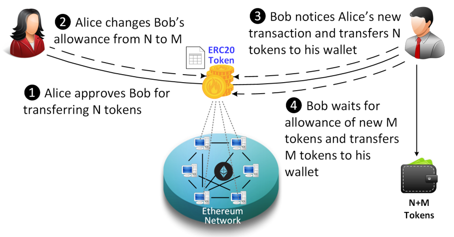
***Index Terms*—Cryptocurrency; Security; ERC20; Token; Smart Contract; Front-Running; Ethereum; Blockchain;**

1. Introduction

Ethereum is an open blockchain proposed in 2013 and deployed in 2015 [3]. At the time of writing, it has the second largest market value behind Bitcoin.[1] It has a large development community which track enhancements and propose new ideas.[2] Ethereum enables decentralized applications (DApps) to be deployed and run on it—DApps can accept and transfer Ethereum's built in currency (ETH).

Commonly, DApps will issue their own custom currency-like tokens for specific purposes. Tokens might be currencies with different properties than ETH, they may be tokens that are required for access to a DApp's functionality, or they might represent ownership of some off-blockchain asset. It is beneficial to have tokens be interoperable with other DApps and off-blockchain webapps, such as exchange services that allow tokens with some market value to be traded. Toward this goal, the Ethereum project accepted a proposed standard (EIP20) for a set of methods that standard-compliant tokens (called ERC20 tokens) should implement. In terms of object oriented programming (most DApps are developed in a Java-like language called Solidty), ERC20 is an interface that defines abstract methods (name, parameters, return types) and provides guidance on how the methods should be implemented, but does not provide an actual concrete implementation.

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



❷ Alice changes Bob's allowance from N to M
❸ Bob notices Alice's new transaction and transfers N tokens to his wallet
ERC20 Token
❶ Alice approves Bob for transferring N tokens
❹ Bob waits for allowance of new M tokens and transfers M tokens to his wallet
Ethereum Network
N+M Tokens

Figure 1. 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.

Since its introduction in November 2015, several vulnerabilities have been discovered in ERC20's design. In October 2017, the security issue called the “Multiple Withdrawal Attack” was opened on GitHub[11], [5]. The attack originates from how two methods in the ERC20 standard, methods for approving and transferring tokens, are described. The use of 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 (see Figure 1). This issue is still open and several solutions have been made to mitigate it. Authors of the ERC20 standard [4] provided generic implementations of ERC20 tokens from OpenZeppelin[7] and ConsenSys[2]. OpenZeppelin uses two additional methods and ConsenSys has not attempted to work around the issue. There are other implementations that have a variety of different trade-offs in solving this issue.

Contributions. In this paper, we discuss 10 solutions to the multiple withdrawal attack and evaluate them according to a set of criteria we develop in terms of compatibility with the standard and attack mitigation (see the summary in Table 2). Since none of them precisely satisfy the constraints of the ERC20 standard, we were motivated to propose 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. The second proposal secures *transferFrom* method by not allowing token transfers more than allowed. Since the attack happens in the event of gap between transactions, we used compare and set

#	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.

(CAS) pattern[15] to cover this gap. CAS is one of the most widely used lock-free synchronization strategy that allows 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.

2. Preliminaries

2.1. How attack could happen

According to the ERC20 API definition:

- The `approve` function[3] allows `_spender` to withdraw up to the `_value` amount of tokens from token pool of the approver. If this function is called again, it overwrites the current allowance with the new `_value`.
- The `transferFrom` function[4] grants required rights to the spender (account, wallet or other smart contract) for transferring `_value` amount of tokens from address `_from` to address `_to`.

An adversary can exploit the gap between execution of `approve` and `transferFrom` functions since the `approve` method overrides current allowance regardless of whether the spender already transferred any tokens or not. Futher, transferred tokens are not trackable and only `Transfer` events[5] will be logged, which is not sufficient in case of transferring tokens to a third party. To make it more clear, the following attack scenario can be considered:

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 after it is broadcast to the Ethereum network but before it is added to a block.
4) Bob quickly sends another transaction with high gas costs to bribe the miners, that runs `transferFrom(_Alice, _Bob, N)`. If a miner adds this transaction before Alice's, it will transfer N Alice's tokens to Bob.
5) Alice's transaction will be executed after Bob's.
6) Before Alice notices that something went wrong, Bob calls `transferFrom` method for the second time 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. If Bob could withdraw N tokens after Alice initial approval, this would be 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.

3. `approve(address _spender, uint256 _tokens)`

4. `transferFrom(address _from, address _to, uint256 _tokens)`

5. `Transfer(address indexed _from, address indexed _to, uint256 _value)`

2.2. General prevention techniques

Solutions can be categorized into three general techniques: securing `approve`, securing `transferFrom`, or punting the issue to the contract owner.

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 was set to zero. Was it because of 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, 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 later in with the MiniMeToken, this approach can not prevent the attack since it is not distinguishable which transaction (*i.e.,* owner or attacker transaction) has set allowance to zero.

Prevention by `approve` method. By using compare and set (CAS) pattern [15] in this approach, `approve` method can change spender allowance from N to M atomically. 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. Although this is promising approach, but 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]. In other words, any adjustments in allowance is prohibited 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 literally to 110, which is allowing Bob for transferring 100+110=210 tokens in total. We implemented this approach in proposal 1 and conlcuded that securing `approve` method can not prevent the attack while adhering constraints of the ERC20 standard.

`transferFrom` method. Based on ERC20 constraint, "approve functions allows `_spender` to withdraw from your account multiple times, up to the `_value`". Hence, 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 tokens transfer in case of already transferred N tokens. By comparing transferred tokens in `transferFrom` method, spender will be restricted to move solely remained 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 (since the `approve` method does not adjust allowance). However, `transferFrom` method does not allow Bob from transferring more than 10 tokens if he had already transferred 100 tokens. We implemented this approach in proposal 2 and it prevents 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 backwards compatible with already deployed smart contracts and web applications using ERC20 tokens. We summarize 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 MUST 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 case to prevent multiple withdrawal from the account.

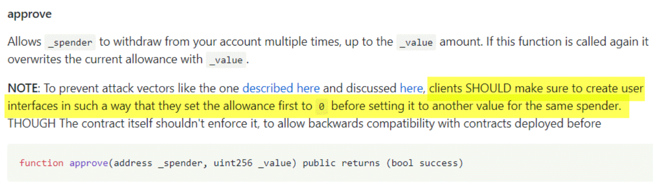
3. Evaluating Proposed Solutions

Several solutions have been proposed by Ethereum community—mostly from developers on GitHub[5]—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:
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. As discussed on GitHub[12], Bob still can transfer N+M tokens in the below scenario:

1) Bob is allowed to transfer N Alices tokens.



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)
```

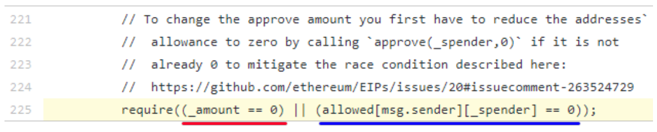
Figure 3. Recommendation of ERC20 standard to mitigate multiple withdrawal attack by enforcement in UI.

2) Alice publishes a new transaction that changes Bobs allowance from N to 0.
3) Bob front runs Alices transaction and transfers N Alices tokens (*transferFrom* sets Bobs allowance to 0 respectively).
4) Alices transaction is mined and sets Bobs allowance to 0.
5) Now Alice publishes a new transaction that changes Bobs allowance from 0 to M.
6) Alices second transaction is mined, Bob now is allowed to move M Alices tokens.
7) Bob transfers M Alices 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 Alices new transaction in step 5 to set Bob's allowance to M. In case of front-running by Bob, Alice needs to check Bobs allowance for the second time before setting any new value. However, she finds out Bobs allowance 0 in either case. In other words, she can not distinguish whether Bobs allowance is set to 0 because of her transaction in step 2 or Bob already transferred tokens by front-running is step 3 and that made his allowance 0. 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 Alices 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 Bobs 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.

3.2. Minimum viable token

As suggested by Ethereum Foundation[9], 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[6]) is one of these examples since it does not implement the *approve*, *allowance* and *transferFrom* functions. According to ERC20 specifications[4], 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 it as backward compatible solution although mitigates the attack by removing vulnerable functions.



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

Figure 4. MiniMeToken added code to *approve* method for allowing non-zero allowance values if it is already set to zero.

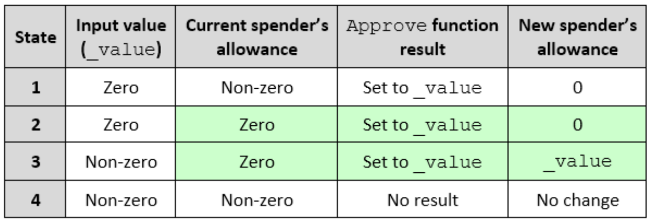
3.3. Approving trusted parties

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 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 mitigation for the attack.

3.4. MiniMeToken

MiniMeToken[6] followed ERC20 recommendation by setting allowance to zero before non-zero values. They added a line of code to their *approve* method. The red clause (*_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): 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

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



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 5. 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).

whether N tokens have been already transferred or not. It is more clear in the below scenario:

1) Alice decides to set Bobs allowance to 0.
2) Bob front-runs Alices transaction and his allowance sets to 0 after transferring N tokens.
3) Alices transaction is executed and sets Bobs allowance to 0 (Red clause passes sanity check).
4) Alice checks Bobs 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 been 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[10] implements two additional functions for allowance increase or decrease. The default *approve* function has additional codes to enforce 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: If the current spenders allowance is non-zero, *decreaseApproval*[7] and *increaseApproval*[8] 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 Bobs allowance is 0. So, he checked Bob's allowance before calling *approve* method.
2) After a while, Alice decides to decrease Bobs approval by M and runs *decreaseApproval(_Bob, M)*.
3) Bob notices Alices second transaction and front runs it by executing *transferFrom(_Alice, _Bob, N)*.
4) Bobs transaction will be executed first and transfers N token to his account and the his allowance becomes 0 as result of this transfer.
5) Alices transaction is mined after Bobs transaction and tries to decrease Bobs allowance by M. If Bob had already transferred more than M tokens, new Bobs allowance becomes negative and it fails the transaction. So, the transaction does not change Bobs 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 Bobs allowance by M.

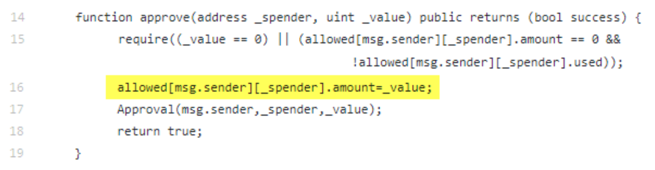
Although these two new functions will prevent the attack, they have not been defined in the initial specifications of ERC20. Therefore, they can not be used by smart contracts that are already deployed on the Ethereum network and still call approve method for setting new allowance—and not *increaseApproval* 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 20 (100 - 80 = 20). Comparatively, increase method will set 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[1] is to move security checks to another function called *safeApprove*[9] that compare current and new allowance values and sets it if has not been already changed. By using this function, Alice uses the standard *approve* function to set Bobs allowance to 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[15], 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.

7. decreaseApproval(address *_spender*, uint *_subtractedValue*)
8. increaseApproval(address *_spender*, uint *_addedValue*)

9. safeApprove(address *_spender*, uint256 *_currentValue*, uint256 *_value*)



```
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      }
```

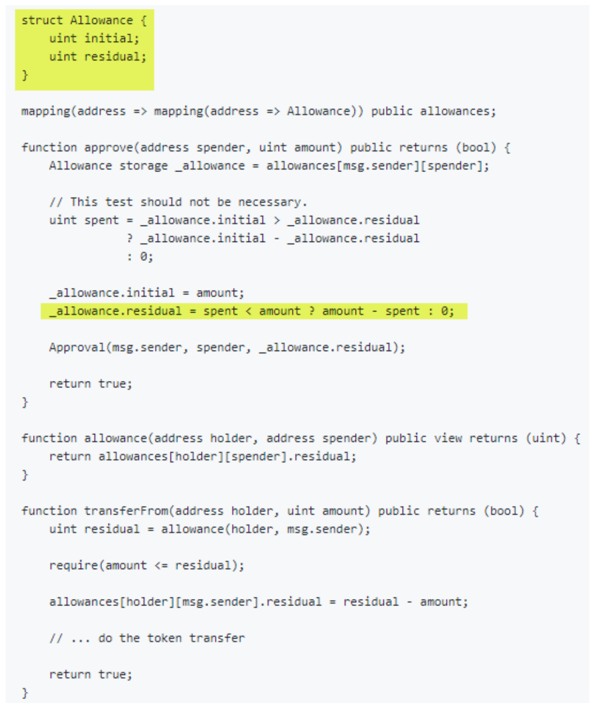
Figure 6. *approve* method needs to be modifed by adding a line of code like *allowed[msg.sender][_spender].used = false;* between lines 16 and 17 to unlock spender flag for the next legitimate change.

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 [14], a flag can be used to detect whether any tokens have 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 Bobs initial allowance is 0 and his corresponding flag=*false*, then sanity check passes and Bobs allowance sets to N in line 15.
2) Alice decides to set Bobs allowance to 0 by executing *approve(_Bob, 0)*.
3) Bob front-runs Alices transaction and transfers N tokens. Then, *transferFrom* turns his flag to *true*.
4) Alices transaction is mined and passes sanity check because passed value is 0.
5) Bobs allowance is set to 0 while his flag remains *true*. (*approve* method does not flip spender flags.)
6) Alice wants to change Bobs allowance to M by executing *approve(_Bob, M)*. Since Bob already transferred N tokens (his flag=*true*), the transaction fails.
7) Bobs allowance does not change and he cannot move more tokens than initially allowed.

Although this approach mitigates the attack, it prevents any further legitimate approvals as well. Considering a scenario that Alice rightfully wants to increase Bobs 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: 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 Bobs 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*.



```
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 7. Keeping track of remaining tokens.

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

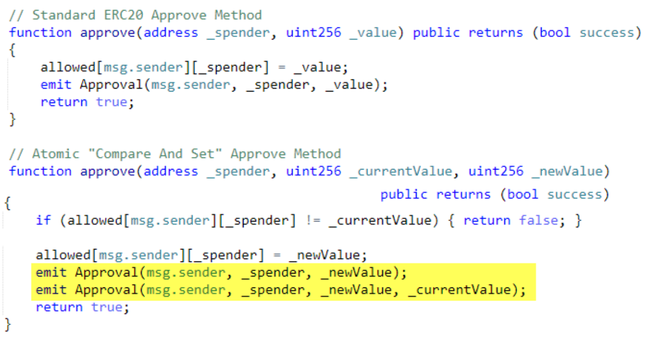
This approach[8] is inspired by the previous solution and keeping track of remaining tokens instead of detecting transferred tokens. It uses modified version of data structure that used in the previous solution for storing residual tokens: At first, it seems to be a promising solution by setting approval 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)”. It would not be possible for Alice to distinguish if any token transfer have occurred when changing allowance. To make it more clear, considering the below scenario:

1) Bobs 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 Bobs allowance to M and has to set it to zero before any non-zero values.
4) Bob noticed Alices 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) Alices 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 [13], changing ERC20 API could secure *approve* method by comparing current allowance of the spender and sets it to new value if it has not already been transferred any tokens. 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 in addition to the standard *approve* method with two parameters: 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. Moreover, one more call is required to read current allowance and pass it to the new *approve* method. 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



```
// 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 8. Suggested ERC20 API Change by adding new *approve* method with three parameters to compare and set new allowance atomically.

these changes makes this token contract incompatible with already deployed smart contracts.

3.10. New token standards

After recognition of this security vulnerability, new standards like ERC233 [10], ERC721[11] and ERC777[12] 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* function instead of *transferFrom*, etc). Nevertheless, migration from ERC20 to ERC223/ERC721/ERC777 would not be convenient and all deployed ERC20 tokens (168,092[13] tokens as of 18 February 2019) 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 a new standards. Despite expanded features and improved security properties of new standards, we would not consider them as target solutions.

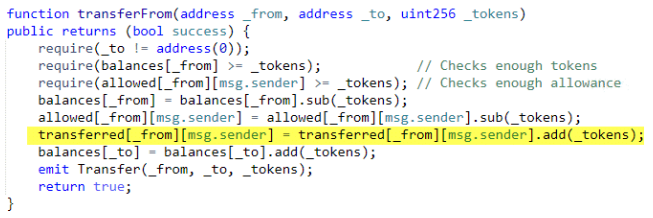
4. New mitigations

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 the ERC20 standard, we now propose two new solutions to mitigate the attack.

4.1. Proposal 1: Securing *approve* method

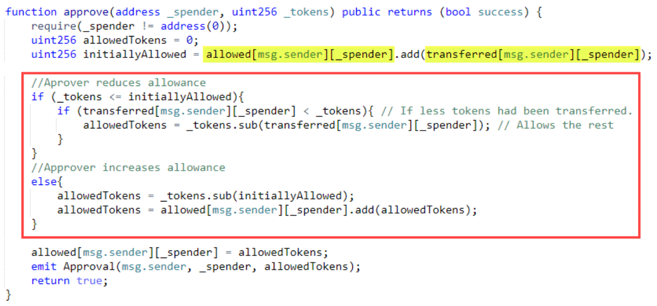
As discussed in the previous section, a sustainable solution should use CAS pattern[15] to set new allowance atomically. This needs knowledge of transferred tokens that

10. https://github.com/Dexaran/ERC223-token-standard
11. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md
12. https://eips.ethereum.org/EIPS/eip-777
13. https://etherscan.io/tokens



```
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 9. 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 10. Added code block to *approve* function to prevent the attack by comparing and setting new allowance atomically.

requires adding a new mapping variable to the token code. The code is still compatible with other smart contracts due to internal usage of the variable. Modified version of *transferFrom* method can track transferred tokens by storing them in this new variable (*transferred*):

Similarly, a block of code is added to the *approve* function to work in both cases with zero and non-zero allowances. Added code compares new allowance (passed as *_tokens* to the function) with the current allowance of the spender and already transferred token (highlighted as *allowed[msg.sender][_spender]* and *transferred[msg.sender][_spender]* respectively). 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 allowance, otherwise increasing allowance was intended.

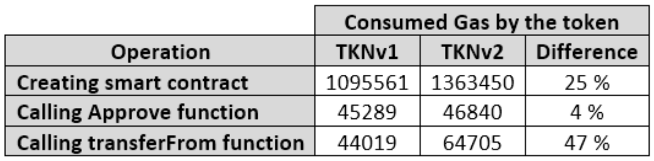
Modified *approve* function prevents the attack in either increasing or decreasing of the allowance. Unlike other solutions, there is no need to set allowance from N to 0 and then to M. The owner can directly change the allowance from N to M which is saving one transaction accordingly. Considering the below scenarios 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 Bobs allowance from 100 to 10 tokens.
3) Bob noticed Alices new transaction and transfers 100 tokens by front-running.
4) Bobs allowance is 0 and *transferred* is 100 (set by *transferFrom* function).
5) Alices 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 Bobs allowance from 100 to 120 tokens.
3) Bob noticed Alices new transaction and transfers 100 tokens by front-running.
4) Bobs allowance is 0 and *transferred* is 100.
5) Alices 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 Bobs allowance.
8) Bob would be able to transfer more 20 tokens (120 in total as Alice wanted).



	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 11. Comparison of gas consumption between standard implementation of ERC20 token (TKNv1) and secured implemetation (TKNv2).

In order to evaluate functionality of the new *approve*/*transferFrom* functions, we have implemented a standard ERC20 token (TKNv1[14]) along side proposed ERC20 token (TKNv2[15]) 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*. This difference is because of maintaining a new mapping variable for tracking transferred tokens:

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

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

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

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¡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 decrease. If *approve* method does not adjust the allowance based on transferred tokens, it has to set it to M (to conform 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. Overall, there is no solution to secure *approve* method while adhering specification of ERC20 standard.

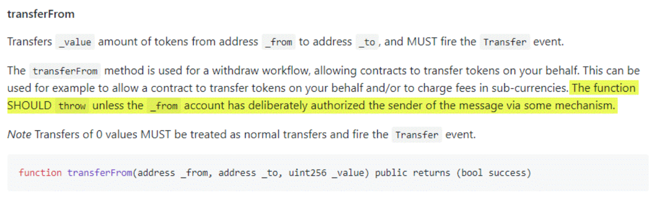
4.2. Proposal 2: Securing *transferFrom* method

As an alternative solution, we can think of securing *transferFrom* method instead of *approve* function. 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 factor. Transferred tokens can be considered as the main variable in calculations. For example in the below situation we can prevent the attack by securing *transferFrom* method and keeping *approve* function as default:

1) Alice allowed Bob for transferring 100 tokens and decides to set it to 70 after a while.
2) Bob front runs Alices transaction and transfers 100 tokes (legitimate transfer).
3) Alices 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, Bobs allowance stays as 70, although transferred tokens shows 100.

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. This impression is completely in accordance with ERC20 standard that emphasizes:

y this assumption, we secured *transferFrom* method instead of *approve* method by adding required codes to prevent more token transfer that allowed:


transferFrom

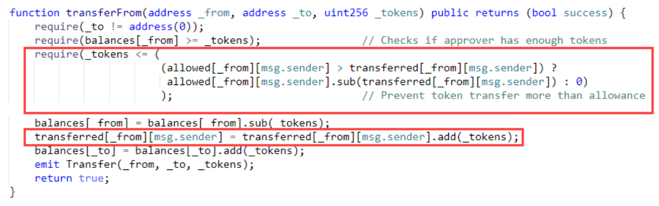
Transfers `_value` amount of tokens from address `_from` to address `_to`, and MUST fire the `Transfer` event.

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 12. ERC20 *transferFrom* method that emphasizes on throwing exception if 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 13. Securing *transferFrom* method instead of *approve* method.

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 (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 is life time variable that accumulates transferred tokens regardless of allowance change. This token is implemented as TKNv3[16] on Rinkby network and passed compatibility by transferring tokens. 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

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 afterwards. This is considered a legitimate transaction and 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 spender to transfer N+M tokens in total. 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 effectively. In this paper we introduced two proposals for securing vulnerable methods—`approve` and `transferFrom`. Proposal 1 incorporates CAS pattern for comparing and setting new

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

allowance atomically. It adjusts new allowance based on transferred tokens which is violating one of ERC20 constraint that says setting new allowance instead of adjusting it. As discussed securing `approve` method is not feasible while adhering ERC20 specifications. Therefore, we secured `transferFrom` function instead of `approve` method in the second proposal. Each proposal has been implemented on Rinkeby network and tested in terms of backward compatibly with already deployed smart contracts and conforming with the standard. Although new proposals consume more gas compared to standard ERC20 implementations, they are secure and could be considered for future secure ERC20 token deployments.

References

[1] 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].

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

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

[6] 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].

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

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

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

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

[11] 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].

[12] 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].

[13] 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].

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

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