Resolving the Multiple Withdrawal Attack on ERC20 Tokens

Reza Rahimian, Shayan Eskandari, Jeremy Clark
Concordia University

***Abstract*—Custom tokens are an integral component of decentralized applications (dapps) deployed on Ethereum and other blockchain platforms. For Ethereum, the ERC20 standard is a widely used token interface and is interoperable with many existing dapps, user interface platforms, and popular web applications (*e.g.,* exchange services). An ERC20 security issue, known as the *multiple withdrawal attack*, was raised on GitHub and has been open since October 2017. The issue concerns ERC20's defined method `approve()` which was envisioned as a way for token holders to give permission for other users and dapps to withdraw a capped number of tokens. The security issue arises when a token holder wants to adjust the amount of approved tokens from N to M (this could be an increase or decrease). If malicious, a user or dapp who is approved for N tokens can front-run the adjustment transaction to first withdraw N tokens, then allow the approval to be confirmed, and withdraw an additional M tokens. In this paper, we evaluate 10 proposed mitigations for this issues and find that no solution is fully satisfactory. We then propose 2 new solutions that mitigate the attack, one of which fully fulfills constraints of the standard, and the second one shows a general limitation in addressing this issue from ERC20's approve method.**

***Index Terms*—Ethereum; ERC20 tokens; Blockchain;**

1. Introduction

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

Toward this goal, the Ethereum project accepted a proposed standard (called ERC20 [2]) for a set of methods which ERC20-compliant tokens should implement. In terms of object oriented programming, ERC20 is an interface that defines abstract methods (name, parameters, return types) and provides guidelines on how the methods should be implemented, however it does not provide an actual concrete implementation (see Figure 1).

1. [2019-02-11] https://coinmarketcap.com/currencies/ethereum/
2. [2019-02-11] https://www.coindesk.com/data

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

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

Figure 1. The ERC20 standard defines 6 methods and 2 events that must be implemented. Using `approve` and `transferFrom` in the existence of a race condition may lead to a *"multiple withdrawal attack"*. Note that `transferFrom` augments the more basic `transfer` method.

Since the introduction of ERC20 in November 2015, several vulnerabilities have been discovered. In October 2017, a security issue called *"Multiple Withdrawal Attack"* was opened on GitHub [3], [4]. The attack originates from two methods in the ERC20 standard for approving and transferring tokens. The use of these functions in an adverse environment (*e.g.,* front-running [5]) could result in more tokens being spent than what was intended. This issue is still open and several solutions have been made to mitigate it. The authors of the ERC20 standard [2] reference two sample implementations: OpenZeppelin [6] and ConsenSys [7]. OpenZeppelin mitigates the attack by introducing two additional methods to replace the `approve` method (see Section 3.3), and the ConsenSys implementation does not attempt to resolve the attack. Additional implementations have a variety of different trade-offs in mitigating the issue (see Section 3).

Contributions. In this paper, we evaluate 10 proposed mitigations for the *"multiple withdrawal attack"*. We develop a set of criteria that encompass backwards compatibility, interoperability, adherence to the ERC20 standard, and attack mitigation. The summary is provided in Figure 2. Since no mitigation is fully satisfactory, we develop two additional solutions based on the *Compare and Set (CAS*[3]*)* pattern[8]. We study in detail possible implementations of ERC20's `approve` and `transferFrom` methods. We argue that a CAS-based approach can never adequately deploy a secure `approve` method while adhering to the ERC20 standard. We then propose a secure implementation of the `transferFrom` method that mitigates the attack and fully satisfies the ERC20 standard.

3. A widely used lock-free synchronization strategy that allows comparing and setting values atomically.

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

Figure 2. Comparison of 10 proposals and 2 contributed by this paper. In our first proposal, CAS is used to mitigate the attack by comparing transferred tokens with new allowance. It is not fully ERC20-compliant since the allowance result does not always match what is requested. In our second proposal, a new local variable is defined to keep track of transferred tokens and prevents transfers in the case of already transferred tokens.

2. Preliminaries

2.1. How the *multiple withdrawal attack* works

According to the ERC20 API definition, the `approve` function[4] allows a spender (*e.g.,* user, wallet or other smart contracts) to withdraw up to an allowed amount of tokens from token pool of the approver. If this function is called again, it overwrites the current allowance with the new input value. On the other hand, the `transferFrom` function allows the spender to actually transfer tokens from the approver to anyone they choose (importantly: not necessarily themselves). The contract updates balance of transaction parties accordingly.

An adversary can exploit the gap between the confirmation of the `approve` and `transferFrom` functions since the `approve` method replaces the current spender allowance with the new amount, regardless of whether the spender already transferred any tokens or not. This functionality of the `approve` method is shaped by the language of the standard and cannot be changed. Furthermore, while variables change and events are logged, this information is ambiguous and cannot fully distinguish between possible traces. Consider the following illustration:

4. We use the term method and function interchangeably.

1) Alice allows Bob to transfer N tokens on her behalf by broadcasting `approve(_Bob, N)`.
2) Later, Alice decides to change Bob's approval from N to M by calling `approve(_Bob, M)`.
3) Bob notices Alice's second transaction after it is broadcast to the Ethereum network but before it is added to a block.
4) Bob front-runs (using an asymmetric insertion attack [5]) the original transaction with a call to `transferFrom(_Alice, _Bob, N)`. If a miner is incentivized (*e.g.,* by Bob offering high gas) to add this transaction before Alice's, it will transfer N of Alice's tokens to Bob.
5) Alice's transaction will then be executed which changes Bob's approval to M.
6) Bob can call `transferFrom` method again and transfer M additional tokens by broadcasting `transferFrom(_Alice, _Bob, M)`.

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

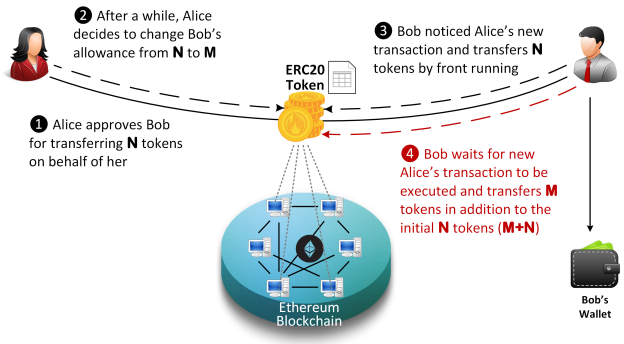


Figure 3. Possible multiple withdrawal attack in ERC20 tokens when Alice changes Bob's allowance from N to M. By front-running, Bob is able to move N+M tokens from token pool of Alice. Solutions should consider Bob's initial transfer of N tokens (step 3) to be legitimate and try to prevent the second transfer of the additional M tokens (step 4).

2.2. Why mitigation is important

ERC20 tokens are important component of Ethereum's supplementary financial system that have many financial (as well as non-financial) uses and could hold considerable value (potentially exceeding the value of Ether itself). There has been more than 64,000 functional ERC20 tokens as of early 2019 [9] that might be vulnerable to this attack. Furthermore, ERC20 tokens that have already been issued cannot easily migrate to a new secure implementation and should these tokens appreciate in value in the future. Resolving the attack also serves as basis for other extended standards, such as ERC-223, ERC-621, and even ERC-777 to be backward compatible with ERC20 interface [10] while offering new enhancements. Finally, firms that hold ERC20 tokens require assurance of their security, particularly in the case that they require their financial statements to be audited—an issue like this could lead to further hesitation by auditors.

2.3. Where to prevent this attack

There are a few logical places to address this attack. Ideally the token author (instead of the token holders) would mitigate the attack within the ERC20 implementation itself. Since two methods are involved in the attack, it could be addressed within the `approve` and/or `transferFrom` method. By contrast, token owners have no control over the implementation of the contract and are relegated to mitigating the attack by carefully monitoring the contract around the time allowance changes are made.

Prevention by token holder. Consider Alice, a token holder using a web app (*e.g.,* a user interface deployed with web3) to adjust Bob's allowance. If this UI is written to the ERC20 standard, Alice will only have `approve` available to her. The ERC20 authors [2] advise Alice against directly changing Bob's allowance from N to M. Instead, she should set the approval to 0 and then it to M. Presumedly, Alice will not do the second approval (setting it from 0 to M) if she sees that Bob withdraws N tokens before her N to 0 adjustment is confirmed.

How will Alice know whether Bob withdraws first? The answer depends on how deeply her webapp can monitor the blockchain. One option is to monitor the variable that records Bob's allowance (technically an on-blockchain helper dapp could also do this). However if she sees Bob's allowance at 0 after initiating the first adjustment, it does not tell her that Bob did not withdraw N—both methods result in Bob having an allowance of 0 so either (or both) could have been executed.

Next, she might rely on events passed from the contract to her web3 app. `Transfer` events as specified in ERC20 will log the transfer parameters (*i.e.,* `address _from, address _to, uint256 _tokens`). Alice's webapp could filter the events and only display transfers matching her address in the `_from` field. The displayed transfers will include any transfer Bob makes, however Bob can provide any address in `_to`, not just his own, and the event does not report who authorized the transaction (*i.e.,* `msg.sender`). If Alice has many authorizations, she cannot determine if a transfer was initiated by Bob or someone else she has authorized.[5] Therefore Alice cannot always unambiguously rely on events to determine Bob did not transfer funds (More details in Section 3.1). If Alice's web app is beyond web3 and runs a full node maintaining blockchain state, it could correctly detect and attribute all transfers initiated by Bob.

The takeaway from all these options is that prevention by token owners has some undesirable properties: (1) it splits adjustments into two transactions, (2) because the first transaction needs to be confirmed before the second is initiated, it takes time to complete, and (3) to precisely mitigate this attack, a heavyweight web app is needed to inspect deeper than variable state changes and events. For these reasons, we concentrate on mitigating the attack in the contract itself. If mitigation at the contract level works, allowances can be adjusted with a single function call from any existing lightweight ERC20 user interface, and no additional monitoring of the contract is necessary.

Prevention by token author in `approve`. The next logical place to tackle multiple withdrawal is in the implementation of the `approve` method. In particular, an `approve` implementation might be engineered to fail under the conditions of multiple withdrawal, to adjust the approval amount, treat adjustments as relative offsets from the current amount, or other techniques. As we review the 10 solutions, we will see different proposals along these lines, as well as our own proposal in Section 4.1. For now, we emphasize that adherence to the standard is a core challenge as it unequivocally states that: "If this function is called again, it overwrites the current allowance with `_value`" [2].

Prevention by token author in `transferFrom`. Recall from Figure 3 that step 4 is the offending function call and it is to `transferFrom`. If we add new state to the contract to track the number of tokens that have been transferred, we can allow approval to work exactly as specified while interpreting it as a "lifetime" allowance. We will explain this in more detail in Section 4.2.

5. Even if Bob is listed in `_to`, another authorized spender might have transferred tokens to Bob to make Alice believe Bob attempted a multiple withdrawal when he actually did not.

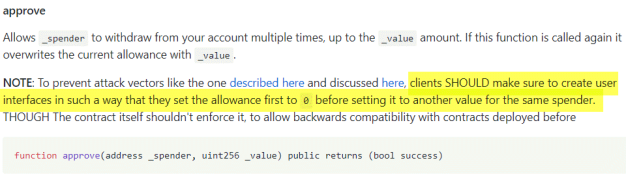


Figure 4. Recommendation of ERC20 standard to mitigate *"multiple withdrawal attack"* by enforcement in UI.

2.4. What an ideal solution looks like

We prioritize adherence to the ERC20 standard. While deviating from the standard might become acceptable if there is no possible way to conform with it and maintain security, we consider that a last resort. Indeed, as we will show, it is possible to secure an ERC20 contract within the constraints of the standard, which we summarize here [2]:

1) The input to `approve` is a new allowance and not a relative adjustment.
2) The result of `approve` will overwrite the current allowance with the new allowance.
3) A call to `transferFrom` on an input of 0 tokens will execute as a normal transfer and emit a `Transfer` event.
4) A spender can call `transferFrom` multiple times up to the allowed amount.
5) Transferring up to any initial allowance is always a legitimate transfer.
6) An ideal solution cannot rely on overloading existing methods or introducing new methods outside of ERC20, as existing ERC20 dapps and web apps would have to be modified to interoperate.
7) A solution must eliminate all race conditions.

3. Evaluating Proposed Solutions

In this section, we evaluate 10 solutions that have been proposed by Ethereum community—mostly from developers on GitHub—to address the multiple withdrawal attack. We examine each solution in detail and evaluate them against the criteria established in the previous section (see Section 2.4). The summary is presented in Figure 2.

3.1. Enforcement by User Interface (UI)

The first solution is enforcement at the user interface level. We discussed this previously in Section 2.3 but we reiterate the main points here again. The exact recommendation from the ERC20 standard is shown in Figure 4 and is essentially to set an allowance to zero before any non-zero values. Presumedly, it will also enforce that the new approval is not allowed if proceeded by a token transfer by the approved spender. We consider the UI to be a lightweight web app that can reference a contract's state variables and emitted events but does not maintain a full copy of the blockchain. Consider (again) the most basic attack sequence:

1) Alice allows Bob to transfer N of Alice's tokens.
2) Alice's client broadcasts an allowance of 0 for Bob.

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

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

3) Bob broadcasts a competing transaction to transfer N of Alice's tokens.
4) Bob's transaction front-runs Alice's, is confirmed, and sets Bob's allowance to 0 (from N).
5) Alice's transaction is confirmed and sets Bob's allowance to 0 (from 0).
6) Alice's client broadcasts an allowance of M for Bob.
7) Alice's second transaction is confirmed and sets Bob's allowance to M.
8) Bob transfers M of Alice's tokens for a total of N+M tokens.

The key mitigation to this attack is for Alice's client to pause at step 6 and determine if a transaction sequence like 3&4 has occurred or not. This cannot be determined by monitoring the integer that records Bob's allowance because it will be 0 regardless of whether steps 3&4 occurred or not.

It also cannot always be determined by monitoring the events emitted from the contract. Step 4 will log a transfer from Alice's address to Bob's address of N tokens. If no event is emitted, Alice can know for certain no transfer was made. However if an event is emitted, Alice must decide it was a transfer initiated by Bob or a transfer by someone else she has authorized. If she has not authorized anyone else, she can know for certain it was Bob. However if she has a busy account with multiple authorized spenders of her tokens, the event is not verbose enough to determine what happened. Importantly, it does not record who initiated the transfer, only who received the tokens, and these are not necessarily the same entity. Bob can send Alice's tokens to his accomplice Carol, or some other authorized spender can send Alice's tokens to Bob (which looks like Bob is attacking when he is not). Since a UI is automated and does not use human discretion, it cannot decide circumstantially whether something looks like an attack or not — it must either specify exact rules, which it cannot do here because of the ambiguity of the events, or it can ask for Alice's human input which introduces usability issues. In conclusion, it is better for enforcement to happen at the contract level.

3.2. MiniMeToken

MiniMeToken [11] enforces the recommendation that allowances are first set to zero before setting to a non-zero value. The enforcement is done within the ERC20 implementation itself by adding a check to the `approve` method (see Figure 5). The red clause in line 225 (`_amount == 0`) allows approvals to be set to 0, and the blue condition requires the allowance of `_spender` to be 0 before it can be set to a non-zero value. This solution fails to prevent Bob from transferring N+M tokens, as the

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 the `approve` method in MiniMeToken and MonolithDAO tokens, which are implemented slightly differently but effectively enforce setting spender's allowance in two steps: first to zero and then to any non-zero value (*e.g.,* N→0→M).

contract will be unable to determine whether N tokens have been already drained by Bob or not. Recall the attack sequence specified in the previous section (Section 4). Step 5 will pass the red clause and Step 7 will pass the blue clause.

3.3. MonolithDAO

MonolithDAO Token [12] (and its extension in OpenZeppelin [6]) implement two additional functions for allowance increases and decreases: `increaseApproval` and `decreaseApproval`. Both take two parameters: the address of the approved spender and the amount to be added/subtracted from the spender's current approval. Additionally, the `approve` method is ERC20 compliant but has additional logic to enforce that owners either set the allowance to zero before non-zero values (see Figure 6) or they use `decreaseApproval` and `increaseApproval` instead of `approve`. Forcing a set to zero is enforced the same way as in MiniMeToken 3.2). Consider the following transaction sequence:

1) Alice allows Bob to transfer N of Alice's tokens by broadcasting `approve(_Bob, N)`.
2) Remark: Alice can use the default `approve` method since Bob's allowance was initially 0.
3) Alice increases Bob's allowance to M. `approve(_Bob, M)` will fail because Bob's allowance is non-zero. Alice broadcasts `increaseApproval(_Bob, M-N)`.
4) Bob front-runs this by broadcasting `transferFrom(_Alice, _Bob, N)`.
5) If Bob's transaction is confirmed first, he will transfer N tokens but this is legitimate as he was approved for N, and his approval is set to 0.
6) Alice's transaction will adjust Bob's approval from 0 to M-N. In total, Bob can spend N+M-N = M tokens as intended.
7) Remark: for decreases, decreaseApproval will similarly prevent the attack by reducing Bob's allowance instead of setting it to a particular value.

Although these two new complementary functions do prevent the attack, they have not been defined in the initial specifications of ERC20 standard. Therefore, these safer functions will not be used by ERC20-compliant web apps and smart contracts that are already deployed. Such deployments will continue to use `approve` which suffers the same issue as MiniMeToken: set to zero does not always mitigate the attack.

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

Figure 7. Detecting token transfers: `approve` method needs to be modified by adding a line of code like `allowed[msg.sender][_spender].used = false;` between lines 16 and 17 to unlock spender flag for the next legitimate change. However, this change makes attack mitigation ineffective.

3.4. Detecting token transfers

The next approach [14] maintains a small state machine to enforce prevent approvals preceded by transfers. We revisit and elaborate on this approach in our own solution in Section 4.1. In this proposal, the implementation of `approve` is augmented with a flag (one flag for each pair of owners and approved spenders) that can be set in `transferFrom`. `transferFrom` sets the flag to `true` after any token transfer and the `approve` method requires the flag to be `false` before allowing new approvals. This approach requires a new data structure, can prevent front-running, but also has a deadlock scenario. Consider the following transaction sequence:

1) Alice allows Bob to transfer N of Alice's tokens by broadcasting `approve(_Bob, N)`.
2) Remark: This succeeds because Bob's allowance was initially 0 and his corresponding flag=`false` (see line 16 in Figure 7).
3) Alice decreases Bob's allowance to 0 by broadcasting `approve(_Bob, 0)`.
4) Bob front-runs this by broadcasting `transferFrom(_Alice, _Bob, N)`.
5) If Bob's transaction is confirmed first, Bob's `transferFrom` turns his flag to `true`.
6) Alice's transaction is confirmed, passes the check because passed value is 0 (line 15), and Bob's allowance is set to 0 while his flag remains `true`.
7) Remark: Critically the `approve` method does not flip the spender's flag.
8) Alice wants to change Bob's allowance to M and broadcasts `approve(_Bob, M)`.
9) Since Bob already transferred N tokens and his flag=`true`, the transaction fails.
10) Remark: Bob's allowance is 0 and it cannot not change so he is locked out of further approvals.

Although this approach mitigates the attack, it prevents any further legitimate approvals. Consider a scenario where Alice wants to increase Bob's allowance from N to M (two non-zero values). If Bob has 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 in Figure 7 does not allow changing the allowance, an exception is thrown. A potential bypass is to set the allowance to 0 and then to M. However this transaction sequence never flips the flag to `false` (there is no code for it in the `approve` method). So it keeps Bob locked out of any further legitimate allowances. A quick fix might be having `approve` set the flag to `false` (*i.e.,* between lines 16 and 17). But this will cause another problem:

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

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

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

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

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

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

    return true;
}
```

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

after setting the allowance to 0, the spender flag becomes `false`, and allows non-zero values even if tokens have been already transferred. This will no longer prevent multiple withdrawals as demonstrated in the following transaction sequence:

1) Alice changes Bob's allowance from N to 0.
2) Bob transfers N tokens before the allowance change and his `used` flag turns to `true`.
3) Alice's transaction is successful since `_value=0`.
4) Remark: The second condition is not evaluated although `used` flag is `true`.
5) Alice's transaction turns `used` flag to `false` and sets Bob's allowance to 0.
6) Now Alice wants to set Bob's allowance from 0 to M, his flag is `false` and allowance is 0.
7) Remark: Alice cannot distinguish whether Bob moved any token or not. Setting a new allowance will allow Bob to transfer more tokens than Alice wanted.

In fact, resetting the flag in `approve` method will not fix the issue and makes attack mitigation 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 state to track transferred tokens if a solution is to be found.

3.5. Keeping track of remaining tokens

This approach [15] is inspired by the previous solution of detecting token transfers by introducing new state in the contract. It keeps track of the remaining tokens, and an ERC20-compliant `approve` method uses these variables to set allowances accordingly (see Figure 8).

At a first glance, it seems to be a promising solution by more effectively enforcing approvals to zero before non-zero values. However, the highlighted code in `approve` method resembles the situation that is explained in 3.1. In case of front-running, both `initial` and `residual` variables will be zero and it would not be possible for Alice to distinguish if any token transfer have occurred due to her allowance change or due to Bob transferring a

```
// 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. An overloaded approve method adds a method with three parameters to compare and set new allowance atomically.

token. To illustrate this, considering the following transaction sequence:

1) Remark: For formatting reasons, we abbreviate `allowances[_Alice][_Bob].initial` as `AB.initial`.
2) Bob's allowance is initially zero (`AB.initial=0`) and his residual is zero as well (`AB.residual=0`).
3) Alice allows Bob to transfer N tokens and makes `AB.initial=N` and `AB.residual=N`.
4) Alice decides to change Bob's allowance to M and has to set it to zero before M.
5) Bob notices Alice's broadcast for setting his allowance to 0 and front-runs it with a transfer of N tokens.
6) Consequently, the `transferFrom` function sets his residual to zero (`AB.residual=0`).
7) Alice's transaction for setting Bob's allowance to 0 is confirmed and sets `AB.initial=0` and `AB.residual=0`.
8) Remark: at this stage, the state is indistinguishable from Step 2. Alice cannot distinguish whether any token have been transferred or not based on `AB.initial` and `AB.residual`.
9) Alice approves Bob for spending new M tokens and Bob is able to transfer new M tokes in addition to initial N tokens.

It is true that at Step 8, a transfer event as been recorded as a result of step 5. However transfer events are ambiguous as described in Section 3.1. Thus it is not always possible for the approver to detect legitimate from non-legitimate tokens transfers. Overall, this approach cannot prevent the attack in all scenarios.

3.6. Overloading Approve

As advised by [16], a secure `approve` method could take one additional parameter: the expected allowance of the spender when the adjustment to the approval is made. Under this proposal, the adjustment succeeds only if the passed expected allowance matches the spender's actual allowance, and fails otherwise. Consider a multiple withdrawal attack where Bob is approved for N of Alice's tokens, Alice adjusts his approval from N to M tokens, and Bob front-runs the approval with a transfer of N tokens. Alice's approval will specify N as the current expected

allowance when adjusting it to M. Because Bob's transfer of N tokens changes his approval to 0, Alice's approval will fail because it expects an allowance of N when the allowance is 0. This allows atomic compare and set of the spender's allowance to make the attack impossible.

While this approach mitigates the attack, it requires a new overloaded `approve` method with three parameters, in addition to the standard ERC20 `approve` method with two parameters (see Figure 9). Additionally it defines a new event. Existing ERC20 web apps and smart contracts will be unaware of the overloaded method and continue to call the insecure two parameter method. Thus it does not provide interoperability.

3.7. Alternate approval function

Another suggestion [13] is to move the security check to a new function called `safeApprove`[6] that compares the current and new allowance values (like the overloaded `approve` in Section 3.6). The adjustment is only allowed if the allowance has not been changed by the time the function is executed. In this case, Alice uses the standard `approve` function to set Bob's allowance to 0 and for new approvals, she has to use `safeApprove` function. As above, `safeApprove` takes the current expected approval amount as input parameter and calls `approve` method if previous allowance is equal to the current expected approval. Although this approach mitigates the attack by using the CAS pattern [8], it is not interoperable with ERC20-compliant web apps and smart contracts that will be unaware of `safeApprove`.

3.8. Minimum viable token

Rather than adding new methods to ERC20, methods can be also be taken away. As suggested by the Ethereum Foundation[17], we can reduce the ERC20 standard to a set of core functionalities and implement only the essential methods. The attack can be side-stepped if methods like `approve` and `transferFrom` are simply not implemented (recall that `transferFrom` is in addition to the more commonly used `transfer`) or are implemented to always throw an exception and revert. Golem Network Token (GNT[7]) is one of these examples since it does not implement the `approve`, `allowance` and `transferFrom` functions. According to the ERC20 specification [2], these methods are not OPTIONAL and must be implemented. Moreover, ignoring them can cause failed function calls from smart contracts or web apps that expect these methods to work as specified. Therefore, we categorize this approach as successfully mitigating the attack but not offering interoperability.

3.9. New token standards

Minimum viable tokens could alternatively be considered a new non-ERC20 token (*cf.* ERC223). In fact, there are many ERC20 alternatives that extend or modify

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

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

ERC20 for a variety of purposes, mostly around functionality but some address multiple withdrawals. We summarize the main proposals in Table 1.

Despite the enhancements of these new token standards for future deployments, ERC20 is ingrained in the community and industry with168,092 deployed tokens[8], many interoperable developer tools and libraries, and web platforms built on trading these tokens. Ideally, and the goal of this paper, a backward compatible solution could be found that does not change the ERC20 API or require token migration to a new standard (which is not necessarily possible to do at the contract level). Like minimum viable tokens, we categorize these approaches potentially mitigating the attack (depending on which standard — see Table 1) but not offering interoperability.

3.10. Approving trusted parties

A final solution is to limit token transfer approvals to trusted entities. Such a solution is discretionary—it cannot be automated within a contract—so it adds additional burden for the user. At first glance, it seem that Alice would never authorize Bob to spend her tokens if she does not trust Bob. However approvals are constrained to specific amounts specifically to enable some less trustworthy interactions. The multiple withdrawal attack is damaging because Bob can circumvent the constraints. If Bob is another smart contract, instead of a user, then Alice could confirm it does not have the logic to conduct a multiple withdrawal attack, cannot be updated (*e.g.,* does not delegate function calls to code at other addresses), and thus it can be trusted with insecure ERC20 tokens. This is a sensible approach but it is quite limited to specific approval scenarios.

4. New mitigations

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

4.1. Proposal 1: Securing `approve`

By implementing the Compare-and-Set (CAS) pattern [8] in the `approve` method, we set up a small state machine so that new allowances can be set

6. `safeApprove(address _spender, uint256 _currentValue, uint256 _value`

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

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

Token Standard	A description of non-compliance with ERC20
ERC 223 [18]	It does not implement ERC20 `approve` and `transferFrom` methods by assuming they are potentially insecure and inefficient.
ERC 667 [19]	It solves the problem of the transfer function in ERC223 (*i.e.,* the need to implement `onTokenTransfer` routing in the receiving contract). It mitigates the attack using the same code as ERC223 with a supplementary function.
ERC 721 [20]	Unlike ERC20 tokens that share the same characteristics, ERC721 tokens are non-fungible tokens (NFT) where each token is unique and not interchangeable. In addition to this functional difference, ERC721does not implement `transferFrom` method of ERC20 standard and introduces a safe transfer function called `safeTransferFrom`.
ERC 777 [21]	It does not implement `transfer` or `transferFrom` methods and replaces them with safe `send` and `operatorSend` methods. Moreover, the costly `approve`/`transferFrom` paradigm is replaced by `tokensReceived` function. Therefore, ERC777 is not backward compatible. A token might implement both ERC20 and ERC777 but the ERC20 methods would require attack mitigation.
ERC 827 [22]	It uses OpenZeppelin's [6] ERC20 implementation and defines three new functions to allow users for transferring data in addition to value in ERC20 transactions. This feature enables ERC20 tokens to have the same functionality as Ether (transferring data and value). In fact, it extends functionality of ERC20 tokens and not addressing the attack.
ERC 1155 [23]	It is improved version of ERC721 by allowing each Token ID to represent a new configurable token type, which may have its own metadata, supply and other attributes. ERC1155 aimed to remove the need to "approve" individual token contracts separately. Therefore, it does not implement any code to address the vulnerability
ERC 1377 [24]	It implements `approve` method with three parameters in addition to the ERC20 default `approve` with two inputs. Additionally, it uses OpenZeppelin [6] approach for increasing and decreasing approvals. We would consider it as mix of MiniToken and OpenZeppelin approaches that we discussed before.

TABLE 1. EVALUATION OF STANDARD'S ADHERENCE TO ERC20 AND MITIGATION OF THE MULTIPLE WITHDRAWAL ATTACK.

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

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

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

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

atomically after a comparison with transferred tokens. This tracking requires adding a new variable to the `transferFrom` method (see Figure 10). Since this is an internal variable, it is not visible to already deployed smart contracts and keeps the `transferFrom` function ERC20-compatible. Similarly, a block of code is added to the `approve` function (see Figure 11) to work in both cases with zero and non-zero allowances. This new logic in the `approve` function compares a new allowance—passed as `_tokens` argument to the function—with the current allowance of the spender and the already transferred tokens. Allowance are in `allowed[msg.sender][_spender]` as in typical EDRC20 implementation, and `transferred[msg.sender][_spender]` is the new state. The method decides to increase or decrease the current allowance based on this comparison. If the new allowance is less than initial allowance—sum of `allowance` and `transferred` variables—it denotes decreasing of allowance, otherwise increasing of allowance is intended. Such a modified `approve` function prevents the attack by either increasing or decreasing the allowance instead of setting it to an explicit value.

Unlike other solutions, there is no need to set allowance from N to 0 and then to M. The token holder can directly change the allowance from N to M which saves time waiting for the confirmation of a transaction and any monitoring of the contract. Consider the following transaction sequences to illustrate how the state changes:

Scenario A. 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, the new allowance (120) will be subtracted from the transferred tokens (100).
7) 20 tokens will be added to Bob's allowance.
8) Bob would be able to transfer 20 more tokens (120 in total as Alice wanted).

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

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

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

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

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

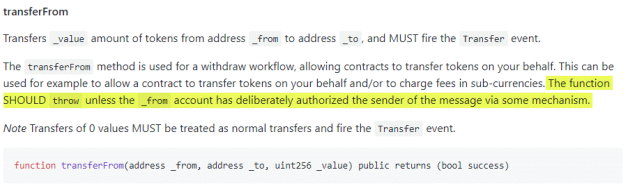
Discussion. In summary, we can use the CAS pattern to implement a secure `approve` method that can mitigate the attack effectively. However, it violates one of the ERC20 specifications that says: "If [the allowance] function is called again, it overwrites the current allowance with `_value`" (item 2 in Section 2.4). Our solution does not comply with this as the resulting allowance can be different than what is passed by the approver (as shown in the scenarios above).

Furthermore we argue that is in fact impossible to secure the `approve` method without adjusting the allowance. Considering the following transaction sequence:

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

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

9. https://rinkeby.etherscan.io/address/0x8825bac68a3f6939c296a40fc8078d18c2f66ac7
10. https://rinkeby.etherscan.io/address/0xf2b34125223ee54dff48f71567d4b2a4a0c9858b


transferFrom

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

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

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

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

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

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

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

4.2. Proposal 2: Securing `transferFrom`

As an alternative to Proposal 1, we can also consider securing the `transferFrom` method instead of the `approve` method. As specified by the ERC20 standard (see figure 13), the goal here is to prevent the spender from transferring more tokens than allowed. Based on this assumption, we should not rely solely on the allowance value in deciding whether to allow or prevent an approve and should also consider the number of transferred tokens, which requires new state as in Proposal 1.

Our solution, which is compliant with a careful reading of ERC20, is to interpret allowance as a 'global' or 'lifetime' allowance value, instead of the amount allowed at the specific time of invocation. For example, say Alice approves Bob for 50 tokens, Bob transfers 50 tokens, Alice approves Bob for 30 (more) tokens, and Bob transfers 30 tokens. In our implementation, Alice would approve Bob for 50 tokens and he transfers 50 tokens. To approve Bob for 30 more tokens, she approves Bob for 80 tokens. He has already spent 50 of these 80 tokens so he will only be allowed to transfer an addition 30. Thus 80 is his lifetime allowance and 50 (kept internally) is the amount he has transferred. In a bit more detail, consider the following, which prevents multiple withdrawals by modifying the implementation of `transferFrom` but keeping `approve` untouched:

1) Alice approvals Bob to transferring 100 tokens
2) Alice broadcasts an approval of 70, decreasing Bob's allowance.
3) Bob front-runs Alice's transaction and transfers 100 tokens (remark: a legitimate transfer).
4) Alice's transaction is confirmed and sets Bob allowance to 70 by the default `approve` method.
5) Bob's noticed the new allowance and tries to move 70 additional tokens by broadcasting `transferFrom(_Bob,70)`.
6) Since Bob has already transferred more than 70 tokens, his transaction fails and prevents multiple withdrawal.

7) In the end, Bob's allowance is set at 70 and his transferred tokens are set at 100.

Performance & Discussion. Interpreting `allowance` as a lifetime allowance is completely in accordance with the ERC20 standard (see figure 13). In our solution, there is no relation between allowance (`allowed[_from][msg.sender]`) and transferred tokens (`transferred[_from][msg.sender]`). The first variable shows lifetime transferable tokens by a spender and can be changed independently of the transferred tokens (*i.e.,* `approve` method does not check transferred tokens). If Bob has not already transferred that many tokens, he would be able to transfer the difference of it—`allowed[_from][msg.sender].sub(transferred[_from][msg.sender])`.

Our token is implemented as TKNv3[11] on Rinkeby test network and it passes compatibility checks by transferring tokens between standard wallets. In terms of gas consumption, its `transferFrom` function needs at about 37% more gas than standard `transferFrom` implementation. We believe this is acceptable for having a secure ERC20 token.

5. Conclusion

While this paper is a deep dive into a specific issue with ERC20, it also illustrates a number of higher level lessons for blockchain developers. When ERC20 standard was first implemented, it changed how people used Ethereum, giving rise to an ICO craze with its ease of use [25]. This led to the deployment of thousands of early implementation ERC20 tokens which has resulted in numerous attacks on different implementations. Now we see decentralize exchanges relying on existing ERC20 tokens and the multiple withdrawal attack seems too important to ignore. Fixing existing ERC20 code will help future deployments but cannot fix the already deployed tokens. In addition to deploying secure contracts, we suggest blockchain developers conduct external audits and consider security-by-design practices when dealing with other smart contract implementations.

References

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

[2] Fabian Vogelsteller and Vitalik Buterin. ERC-20 Token Standard. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md, November 2015. [Online; accessed 2-Dec-2018].

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

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

[5] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. Sok: Transparent dishonesty: front-running attacks on blockchain. *International Conference on Financial Cryptography and Data Security*, 2019.

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

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

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

[9] Friedhelm Victor and Bianca Katharina Lüders. Measuring ethereum-based erc20 token networks. In *International Conference on Financial Cryptography and Data Security*, 2019.

[10] Michael Fröwis, Andreas Fuchs, and Rainer Böhme. Detecting token systems on ethereum. *arXiv preprint arXiv:1811.11645*, 2018.

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

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

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

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

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

[16] 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, November 2016. [Online; accessed 25-Nov-2018].

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

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

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

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

[21] Jacques Dafflon, Jordi Baylina, and Thomas Shababi. EIP 777: A New Advanced Token Standard. https://eips.ethereum.org/EIPS/eip-777, November 2017. [Online; accessed 12-Jan-2019].

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

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

[24] Atkins Chang, Noel Bao, Jack Chu, Leo Chou, and Darren Goh. Service-Friendly Token Standard. https://github.com/fstnetwork/EIPs/blob/master/EIPS/eip-1376.md, September 2018. [Online; accessed 12-Jan-2019].

[25] Gianni Fenu, Lodovica Marchesi, Michele Marchesi, and Roberto Tonelli. The ico phenomenon and its relationships with ethereum smart contract environment. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 26–32. IEEE, 2018.

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