On Secure ERC20 Tokens

Reza Rahimian, Seyedehmahsa Moosavi, Jeremy Clark
Concordia University

***Abstract*—Tokens represent digital assets (*e.g.,* company shares, fiat currency, non-fungible assets, *etc.*) on the Ethereum blockchain. ERC20 is one of the major token standards that defines a group of interfaces for standardizing interoperability of tokens. Therefore, web services and other blockchain applications can use and interact with tokens. Similar to any other smart contract that is written in *Solidity*, ERC20 tokens are vulnerable to security flaws. In this report, we (i) examine most significant discovered vulnerabilities on ERC20 tokens and (ii) propose a secure ERC20 code that can be further used as a reference implementation.**

1. Introduction

Ethereum blockchain project [1] was launched in 2014 by announcing Ether as its protocol-level cryptocurrency[1]. According to *CoinMarketCap*, which ranks cryptocurrencies that are actively traded on an exchange service, Ether is ranked second after Bitcoin[2] in 2019. It also has the biggest development community to track enhancement and introduce new ideas[3]. Ethereum allows users to build decentralized applications (DApps) in the form of smart contracts; computer programs that are written in a high-level programming language called Solidity [2]. Migrating applications from centralized to decentralized architecture can solve many issues (*e.g.,* single point of failure, hardware and maintenance costs, data security, distributed trust, *etc.*). Distributed property of the blockchain removes single point of failure (SPOF) and increases resiliency of the deployed applications on top of it. If one node faced hardware or communication failures, other nodes would continue responding to the coming requests since they have full copy of data locally.

DApps can accept and use Ethereum's protocol-level currency (Ether) or issue their own custom currency-like tokens. Tokens are standardized version of smart contracts[4] that define common set of rules (known as API[5]). The Ethereum project accepted a proposed standard called ERC20– the most popular token standard on the Ethereum. It is in fact an interface defines 6 abstract methods and 2 events (see Figure 1). Having a token

```
contract ERC20Interface {
    function totalSupply() external view returns (uint256);
    function balanceOf(address _account) external view returns (uint256);
    function transfer(address _to, uint256 _tokens) external returns (bool);
    function approve(address _spender, uint256 _tokens) external returns (bool);
    function transferFrom(address _from, address _to, uint256 _tokens) external returns (bool);
    function allowance(address _account, address _spender) external view returns (uint256);

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

Figure 1. An interface declaring the required functions and events to meet the ERC20 standard.

standard provides interoperability with other applications (*e.g.,* wallets, decentralized application, web services, *etc.*). Other applications would be aware of functionalities of tokens and available methods to use. ERC20 standard does not provide an actual concrete implementation and only provides guidelines on how each method should be implemented (such as name of the method, parameters, return types). This gives developers flexibility of coding based of requirements of their DApps. The point here is that all ERC20-compliant tokens must implement these 6 methods and 2 events. They are not optional and this constraint makes ERC20 tokens interoperable with other DApps or web services. In other words, other applications would be aware of how to communicate with ERC20 tokens.

A use case of Ethereum tokens could be considered when implementing a decentralized trading system (see Figure 2). Since the application will be running on the blockchain, we need to represent corresponding financial assets as tokens. Leveraging ERC20 tokens facilitate implementation of left side of this trading model (which is share of company X). Correspondingly, the right side needs a financial asset which is equivalent to a fiat currency (like USD or CAD). Stablecoins[6] provide this functionalities and could also be represented as ERC20 token.

Representing both financial assets as ERC20 tokens, gives us two ERC20 tokens with different values to trade. This shows the importance of tokens in Ethereum ecosystem for digitizing tangible assets to digital equivalents on the blockchain. By using ERC20 tokens, we are able to migrate current centralized trading systems to decentralized equivalent and taking advantage of a distributed trading system on the blockchain. Being subset of smart contracts make ERC20 tokens vulnerable to security threats. In this report, we analyze major identified ERC20 vulnerabilities and discuss their mitigations. By considering them in a reference implementation, we would be able to propose a secure version of ERC20 token that addresses these security issues and can be used for deployment of secure tokens on the Ethereum blockchain.

1. A digital currency in which encryption techniques are used to regulate the generation of units of currency and verify the transfer of funds, operating independently of a central bank.

2. CoinMarketCap-Ethereum currency - [2019-07-11] https://coinmarketcap.com/currencies/ethereum/

3. CoinDesk Crypto-Economics Explorer - [2019-07-11] https://www.coindesk.com/data

4. Can be considered as smart transaction: Types of transactions that execute as they are programmed by a scripting language (like Solidity or Viper). Logic of the transaction is dynamic and can represent an application.

5. Advanced Programming Interface.

6. Types of digital assets that value of it will be stable over time and people be able to count on its value since it is pegging to something that has a stable value (like gold or USD).

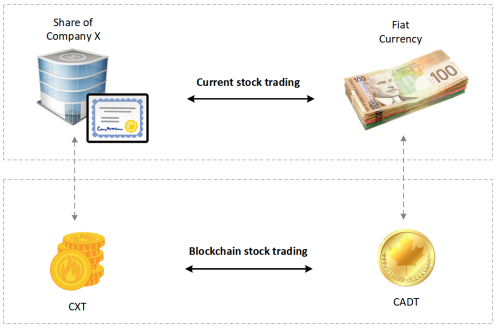

Figure 2. A blockchain trading model using ERC20 tokens to leverage blockchain as a distributed platform. ERC20 standard makes it possible to migrate financial assets to their digital representation on the blockchain.

2. Motivation

The development of smart contracts has proven to be error-prone in practice, and as a result, smart contracts deployed on public platforms are often riddled with security vulnerabilities – previous research [3] showed that at about 45% of existing smart contracts are vulnerable to security threads. Similar to any other smart contract, ERC20 token are also vulnerable to security flaws. There are currently at about 205,787[7] ERC20 tokens on the Ethereum blockchain[8] that might be vulnerable to one of these issues. To address this concern, we focus on identified ERC20 vulnerabilities (such as overflow issue, multiple withdrawal, re-entrancy attack, *etc.*) and research on recommended mitigations. By putting all these solutions together, we will be able to propose a secure implementation of ERC20 token that can be used as a reference model for future ERC20 implementations. Developers can also refer to each mitigation separately to address a specific attack in their migrated ERC20 tokens.

In addition to increasing security of ERC20 tokens, some tokens have considerable value that potentially exceeding the value of Ether itself (*e.g.,* MKR[9] at $580 and XIN[10] at $210). Similar to other financial assets, ERC20 tokens might be audited. Existence of security thread will impact ownership of tokens that may lead to hesitation of auditors. Therefore, providing attack severity helps auditors to better assess associated risk of ERC20 tokens.

3. Contributions

This paper provides an overview of three major vulnerabilities (overflow attack, multiple withdrawal attack, and re-entrancy attack) on Ethereum's most popular token standard; Ethereum ERC20 tokens. We carefully select these vulnerabilities based on their attack vector and broader impact on the Ethereum blockchain. For each of the selected vulnerabilities, we (i) thoroughly **explain** and **visualize** the technical details , (ii) reproduce the attack using *Solidity* programming language (attack demo), (iii) provide the severity and technical details of how to mitigate and/or prevent the attack. Eventually, we **propose** a secure ERC20 standard for the Ethereum blockchain that is not vulnerable to any of the vulnerabilities discussed. We ultimately implement, deploy, and test our ERC20 proposal on Rinkeby[11]. The proof of concept of our proposal can be found in Section 5.

7. https://etherscan.io/tokens, Accessed 12-Aug-2019
8. 64,000 functional ERC20 tokens as of early 2019 [4]
9. https://makerdao.com/en/, [Accessed online: July 27, 2019]
10. https://mixin.one/, [Accessed online: July 27, 2019]

4. ERC20 Vulnerabilities

4.1. Overflow attack

This high severity attack took place in April, 2018 due to a well known and common issue in many programming languages called *integer overflow* [12]. It termed as **batchOverflow** or **proxyOverflow** and some exchanges (such as OKEx[13], Poloniex[14], HitBTC[15] and Huobi Pro[16]) suspended deposits and withdrawals of ALL tokens, especially for Beauty Ecosystem Coin (BEC[17]) that was targeted by this exploit.

4.1.1. Attack analysis. In this attack, malicious person was able to run a transaction[18] and transfer two extremely large amount of BEC tokens to two Ethereum addresses. Although BEC developers had considered most of the security measurements, only line 257 of the code (see Figure 3) was vulnerable against this classic integer overflow attack[5]. The attacker was able to pass a combination of input values that generate large results than the maximum value of `uint256` data type can hold. This caused integer overflow and only the least significant bits have been retained. In other words, the `uint256` variable reached to its maximum value that can be held and it *wraps around*[19] by starting from 0. For example, an `uint8` (8-bit unsigned integer) can represent maximum value of $2^8 - 1 = 255$(`0xff`). Multiplying `0x02` by `0x80` causes integer overflow and produces `0x00` as the result (`0x02` × `0x80` = `0x100` → `0x00`). We can achieve the same result by adding `0x01` to `0xff` (`0x01` + `0xff` = `0x100` → `0x00`). So, In BEC case, the attacker passed two addresses (`cnt = _receivers.lengh = 0x02`) and a large value (`_value = 0x8000000000000000000000000000000000000000000000000000000000000000`) (63 0's)) to `batchTransfer()` function. Because of wrap around, the result of `amount` variable in line 257 was calculated as `0x00` and bypassed sanity checks in line 259 (see Figure 3). Hence, line 264 transferred the specified `_value` to those two addresses. The transferred amount was even larger than the initial supply of the token– that was `7000000000000000000000000000` (27 0's) tokens, hence allowing the attacker to take control of token finance and manipulate its price.

11. Similar to the Ethereum main net but it is designed for testing contracts. Ethers are not real on it and it is just for testing. It can be requested via https://faucet.rinkeby.io
12. https://en.wikipedia.org/wiki/Integer_overflow
13. https://okex.com
14. https://poloniex.com/
15. https://hitbtc.com/
16. https://www.huobi.com/en-us/⟩
17. https://etherscan.io/address/0xc5d105e63711398af9bbff092d4b6769c82f793d
18. https://etherscan.io/tx/0xad89ff16fd1ebe3a0a7cf4ed282302c06626c1af33221ebe0d
19. https://en.wikipedia.org/wiki/Integer_overflow

Contract Source Code </>

```
254
255   function batchTransfer(address[] _receivers, uint256 _value) public whenNotPaused returns (bool) {
256     uint cnt = _receivers.length;
257     uint256 amount = uint256(cnt) * _value;
258     require(cnt > 0 && cnt <= 20);
259     require(_value > 0 && balances[msg.sender] >= amount);
260
261     balances[msg.sender] = balances[msg.sender].sub(amount);
262     for (uint i = 0; i < cnt; i++) {
263         balances[_receivers[i]] = balances[_receivers[i]].add(_value);
264         Transfer(msg.sender, _receivers[i], _value);
265     }
266     return true;
267   }
268 }
```

Figure 3. Vulnerable code in BEC token, `batchTransfer()` function.

Contract Source Code </>

```
202     */
203     function transferProxy(address _from, address _to, uint256 _value, uint256 _feeSmt,
204         uint8 _v,bytes32 _r, bytes32 _s) public transferAllowed(_from) returns (bool){
205
206         if(balances[_from] < _feeSmt + _value) revert();
207
208         uint256 nonce = nonces[_from];
209         bytes32 h = keccak256(_from,_to,_value,_feeSmt,nonce);
210         if(_from != ecrecover(h,_v,_r,_s)) revert();
211
212         if(balances[_to] + _value < balances[_to]
213             || balances[msg.sender] + _feeSmt < balances[msg.sender]) revert();
214         balances[_to] += _value;
215         Transfer(_from, _to, _value);
216
217         balances[msg.sender] += _feeSmt;
218         Transfer(_from, msg.sender, _feeSmt);
219
220         balances[_from] -= _value + _feeSmt;
221         nonces[_from] = nonce + 1;
222         return true;
223     }
```

Figure 4. Vulnerable code in SMT token. Passing two large values to the `transferProxy()` function bypasses sanity check in line 206 and makes the attack possible.

Smart(SMT[20]) and Mesh(MESH[21]) were the next victims of this exploit. The attackers were able to transfer[22] `0x8fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff` (63 f's) tokens to one address [23] and `0x7000000000000000000000000000000000000000000000000000000000000001` (62 0's) as huge fee to the transaction initiator[24].

They executed `transferProxy()` function (see Figure 4) which was designed for transferring tokens on behalf of someone else by taking a fee. Line 206 of this ERC20 token was vulnerable and sum of `_feeSmt` and `_value` produced zero. Consequently, this bypassed the sanity check and allowed execution of the reset of the code that resulted in this attack. In addition to BEC, SMT and MESH tokens, the following ERC20 tokens have been identified as overflow-affected[6]:

1) UGToken[25]
2) SMART[26]
3) MTC[27]
4) First[28]
5) GGoken[29]
6) CNYToken[30]
7) CNYTokenPLus[31]
8) MESH[32]

```
contract multiplyDemo{
    uint256 public a = 0x8000000000000000000000000000000000000000000000000000000000000000;
    uint256 public b = 0x2;
    uint256 public c;

    constructor() public {
        c = 0x3;
    }

    function a_multiply_b() public returns (bool){
        c = a * b;
        return (c == 0) ? true : false;
    }
}
```

Figure 5. Integer overflow demonstration in multiplication of `uint256` variables

Figure 6. Multiplication of `a` and `b` by `a_multiply_b()` function causes `uint256` overflow. It returns `true` since the result is 0. It implies *wraps around* in unchecked context of Ethereum.

4.1.2. Reproducing the attack. To check feasibility of this attack in the current version of Solidity[33] compiler (0.5.10 as July 2019), a demo smart contract (called `multiplyDemo` as shown in Figure 5) is created and used to test overflow issue. As it can be seen, the `a_multiply_b()` function multiplies `a` and `b` to exceed maximum capacity of `uint256` data type and set `c` variable to 0 (due to *wraps around*). We have initially set `c=0x3` to check its result before and after multiplication performed by `a_multiply_b()` function.

Running this code on the RemixIDE[34] returns `true` (since `c=0`). Transaction log also shows that Ethereum has executed `a_multiply_b()` function in unchecked context[35] and logged successful status by returning `true` as output of the function (see figure 6).

The same result can be seen from RemixIDE interface (see Figure 7). On the left, `c` is 3 before execution of `a_multiply_b()` function and on the right, it is 0 after that.

By default, integer overflow does not throw a runtime exception in Ethereum. This is by design and we can get the same overflow issue in the summation of two

20. https://etherscan.io/address/0x55f93985431fc9304077687a35a1ba103dc1e081#code
21. https://etherscan.io/address/0x3ac6cb00f5a44712022a51fbace4c7497f56ee31#code
22. https://etherscan.io/tx/0x1abab4c8db9a30e703114528e31dee129a3a758f7f8ebc3b6494aad3d304e43f
23. https://etherscan.io/token/0x55f93985431fc9304077687a35a1ba103dc1e081?a=0xdf31a499a5a8358b74564f1e2214b31bb34eb46f
24. https://etherscan.io/address/0xd6a09bdb29e1eafa92a30373c44b09e2e2e0651e
25. https://etherscan.io/address/0x43ee79e379e7b78d871100ed696e803e7893b644#code
26. https://etherscan.io/address/0x60be37dacb94748a12208a7ff298f6112365e3bf#code
27. https://etherscan.io/address/0x8febf7551eea6ce499f96537ae0e2075c5a73034#code
28. https://etherscan.io/address/0x9e88770da20ebea0df87ad874c2f5cf8ab92f605#code
29. https://etherscan.io/address/0xf20b76ed9d5467fdcdc1444455e303257d2e27d7#code
30. https://etherscan.io/address/0x041b3eb05560ba2670def3cc5eec2aeef8e5d14b#code
31. https://etherscan.io/address/0xfbb7b2295ab9f987a9f7bd5ba6c9de8ee762deb8#code
32. https://etherscan.io/address/0x3ac6cb00f5a44712022a51fbace4c7497f56ee31#code
33. An object-oriented programming language for writing smart contracts on the Ethereum blockchain
34. Online programming tool for developing smart contracts on the Ethereum blockchain. It supports Solidity and Vyper programming languages. https://remix.ethereum.org/
35. In an unchecked context, arithmetic overflow is ignored and the result is truncated by discarding any high-order bits that don't fit in the destination type.

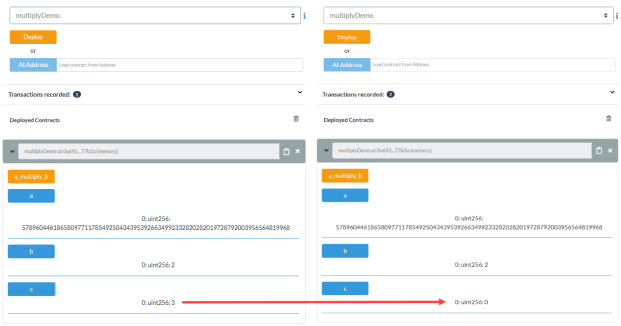

Figure 7. Overflow issue in multiplication. On the left, before execution of `a_multiply_b()` and on the right, after that. `c` variable is set to zero due to *wraps around*.

```
contract overflowDemo {

    uint256 public a = 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff;
    uint256 public b = 0x0000000000000000000000000000000000000000000000000000000000000001;
    uint256 public c;

    constructor() public {
        c = 0x3;
    }

    function a_plus_b() public{
        c = a + b;
    }
}
```

Figure 8. Sample code to demonstrate integer overflow issue in summation of `uint256` variables

`uint256` variables. To reproduce the issue, we created another demo smart contract with `a_plus_b()` function that adds two variables. In this case the result (`c` variable) sets to 0 due to wraps around (see Figures 8 and 9).

4.1.3. Attack mitigation. As analyzed in the previous section, the arithmetic result of numeric values outside of the representable range will lead to wrap around and sets the result to 0. Although this behaviour is expected in Ethereum, it causes security problems as explained in CVE-2018–10299[36] and CVE-2018-10376[37]. To address this issue, it is recommended to use `SafeMath` library[38] when performing any arithmetic calculation. This library is offered by OpenZeppelin[39] and has become industry standard for catching overflows. Moreover, auditing before launching the code could also prevent such human errors and help to be in compliance with the best practices. We used `SafeMath` library and re-implemented vulnerable functions to examine its effectiveness (see Figures 10 and 11).

By adding "`using`"[40] keyword to `a_plus_b()` function, we could give any `uint256` within the `a_plus_b()` function the libraries functions and pass the `uint256` as the first parameter. As highlighted, `a.add(b)` calls `add(uint256 a, uint256 b)` from `SafeMath` library and passes `b` as input variable. The `add` function in `SafeMath` checks the result and raises an error message in case of integer overflow (see Figure 12). Throwing an exception will undo all state

36. https://nvd.nist.gov/vuln/detail/CVE-2018-10299
37. https://nvd.nist.gov/vuln/detail/CVE-2018-10376
38. https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/math/SafeMath.sol
39. https://github.com/OpenZeppelin/openzeppelin-solidity
40. https://solidity.readthedocs.io/en/v0.5.10/contracts.html?highlight=Library#libraries

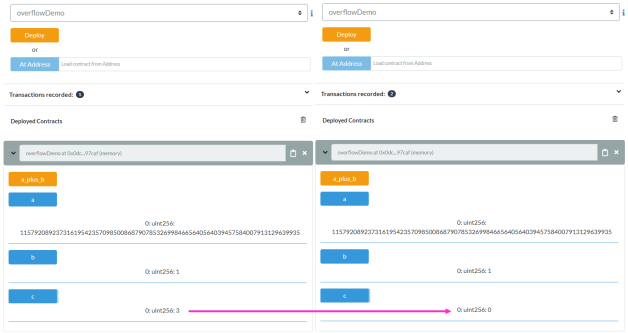

Figure 9. Possibility of integer overflow in summation. On the left, before execution of `a_plus_b()` and on the right, after that. `c` variable is set to zero due to *wraps around*.

```
16  library SafeMath {
17      /**
18       * @dev Returns the addition of two unsigned integers, reverting on
19       * overflow.
20       *
21       * Counterpart to Solidity's `+` operator.
22       *
23       * Requirements:
24       * - Addition cannot overflow.
25       */
26      function add(uint256 a, uint256 b) internal pure returns (uint256) {
27          uint256 c = a + b;
28          require(c >= a, "SafeMath: addition overflow");
29
30          return c;
31      }
```

Figure 10. `Add` function of `SafeMath` library checks for overflow and raises an error in line 28. The `require()` function simply checks for the result to be greater than input (`a`) and throws an exception if the condition is not met. It returns an error message that can be retreived from the exception handler.

changes and stop execution of the transaction. Therefore, it mitigates the attack effectively.

4.2. Multiple withdrawal attack

As explained, ERC20 tokens are technically standardized version of smart contracts that could be vulnerable to security flaws. Some of these vulnerabilities have been already discovered and addressed by Ethereum community. One of these issues that is known as "Multiple Withdrawal Attack" was originally opened on GitHub[7] and raised as separate thread[8] for making it easy to follow. It is security issue in protocol-level and originating from definition of APIs in ERC20 standard for approving and transferring tokens. There are two functions (see figure 13) that can be used to authorize a third party to transfer tokens on behalf of someone else. Using these functions (i.e., `approve` and `transferFrom`) in an undesirable situation could result in conditions that allow attacker to transfer more tokens than the owner ever wanted. Since this security issues is at protocol-level, it is important to consider it when implementing a secure ERC20 code. If the token become valuable (e.g., MKR[41] at $580 and XIN[42] at $210), it may incentivize attackers to use this vulnerability to transfer more token than ever approved.

41. https://makerdao.com/en/, [Accessed online: July 27, 2019]
42. https://mixin.one/, [Accessed online: July 27, 2019]

```
contract overflowDemoSafe {
    using SafeMath for uint256;

    uint256 public a = 0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe;
    uint256 public b = 0x0000000000000000000000000000000000000000000000000000000000000001;
    uint256 public c;

    constructor() public {
        c = 0x3;
    }

    function a_plus_b() public{
        c = a.add(b);
    }
}
```

Figure 11. Using `SafeMath` library to prevent overflow attack.

```
transact to overflowDemoSafe.a_plus_b pending ...

[vm] from:0xca3...a733c to:overflowDemoSafe.a_plus_b() 0x089...659fb value:0 wei

transact to overflowDemoSafe.a_plus_b errored: VM error: revert.
revert  The transaction has been reverted to the initial state.
Reason provided by the contract: "SafeMath: addition overflow". Debug the transaction
```

Figure 12. Reverted transaction by `SafeMath` due to overflow check.

4.2.1. Attack analysis. By using this vulnerability and in existence of race condition[43] or front-running[44], an attacker is able to transfer more tokens than approved. This is possible by executing `transferFrom` function two times, before and after the `approve` method. According to ERC20 API definition[10]:

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

An attacker can take advantage of the gap between execution of `approve` and `transferFrom` functions since the `approve` method overrides current allowance regardless of whether spender already transferred any tokens or not. Transferred tokens are not trackable and only `Transfer` event will be logged (which is not sufficient in case of transferring tokens to a third parity). To make it more clear, a possible attack scenario could happen in the below sequences (see figure 14):

1) Alice allows Bob to transfer N tokens from her token pool by calling `approve(Bob, N)`.
2) After a while, Alice decides to change Bob's approval from N to M by executing `approve(Bob, M)`.
3) Bob notices Alice's second transaction before it was mined and quickly sends another transaction that runs `transferFrom(Alice, Bob, N)`. This will transfer N Alice's tokens to Bob.
4) Bob's transaction is executed before Alice's transaction (because of higher transaction fee, miner's policy or other prioritization techniques) and Bob front-runs Alice's transaction.
5) Alice's transaction is executed after Bob's and allows Bob to transfer more M tokens.

43. Execution of two transactions at the same time with undesirable situation.

44. A course of action where an entity benefits from prior access to privileged market information about upcoming transactions and trades[9].

```
contract ERC20Interface {
    function totalSupply() external view returns (uint256);
    function balanceOf(address _account) external view returns (uint256);
    function transfer(address _to, uint256 _tokens) external returns (bool);
    function approve(address _spender, uint256 _tokens) external returns (bool);
    function transferFrom(address _from, address _to, uint256 _tokens) external returns (bool);
    function allowance(address _account, address _spender) external view returns (uint256);

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

Figure 13. Vulnerable methods in ERC20 API that allow an attacker to withdraw multiple times from approver token pot.

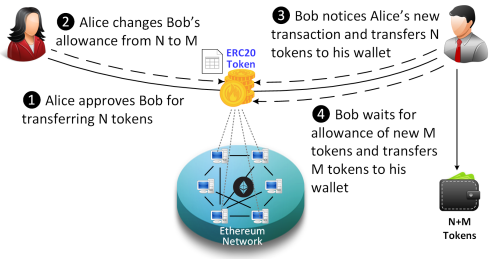

Figure 14. Possible multiple withdrawal attack in ERC20 tokens when Alice changes Bob's allowance from N to M. By front-running, Bob is able to move N+M tokens from Alice's pot.

6) Bob successfully transferred N Alice's tokens earlier and gains ability of transferring another M tokens.
7) Before Alice notices that something went wrong, Bob calls `transferFrom` method for the second time and transfers M Alice's tokens by executing `transferFrom(Alice, Bob, M)`.

In fact, Alice attempted to change Bob's allowance from N to M, but she made it possible for Bob to transfer N+M of her tokens at most, while Alice never wanted to allow so many transfers to be occurred by Bob. It should be noted that the assumption here is to prevent Bob from withdrawing Alice's tokens multiple times when allowance changes from N to M. If Bob could withdraw N tokens after initial approval of Alice, this would be considered as a legitimate transfer (since Alice has already approved it). In other words, it would be responsibility of Alice to make sure before approving anything to Bob. After approval, Bob is allowed to transfer up to N tokens even right before allowance change. In figure 14, transaction #4 would be considered as multiple withdrawal attack since Bob is able to move more M tokens in addition to already transferred N tokens in step #3. An effective solution must prevent transaction #4 to prevent Bob from withdrawing multiple times.

4.2.2. Reproducing the attack. This attack can be categorized as medium severity since executing it requires tracking of Ethereum transactions and ideally being as a miner. However similar attacks happened before [9], [11]. Status.im[45] was targeted by this attack on June 2017 and miners were able to prioritize purchasing of this token during initial coin offering (ICO)[46]. Another similar attack was due to Re-entrancy attack (see section 4.3) which a potential race condition failed DAO to correctly update

45. An open source mobile DApp browser and messenger built for Ethereum

46. When a cryptocurrency startup firm raises money by offering a portion of ownership stake as tokens

the contract state and allowed for funds to be stolen. This shows that feasibility of using similar techniques to execute multiple withdrawal attack.

4.2.3. Attack mitigation. As explained in [11] this attack can be mitigated in three levels:

1) **Prevention by token holder:** This approach is recommended by ERC20 standard [10] and advises token holders to use user interface (UI) that changes 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. Despite recommendation of this approach by ERC20 standard, we will not use it in our proposal since it is not distinguishable which transaction (i.e., owner or attacker transaction) has set allowance to zero. However, it might be considered as last resort for already deployed ERC20 tokens, but for new deployments (focus of this project), prevention in the contract is more efficient (and sustainable) than UI.
2) **Prevention by `approve` method:** In this approach, `approve` method changes spender allowance from N to M atomically by comparing new allowance with transferred tokens 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`" [10]. 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. Overall, securing `approve` method can not prevent the attack while adhering constraints of the ERC20 standard. Therefore, we will not use this approach in our proposal.
3) **Prevention by `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. This approach mitigates the attack by adding a piece of code to the `transferFrom` method (see figure 15).

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

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

Figure 15. Added code to `transferFrom` method to prevent more token transfer than allowed. It uses a new mapping variable to track transferred tokens by each spender.

Overall, if token holders approve someone to transfer N tokens, the spender can transfer exactly N tokens, even if they change it afterwards. This is considered a legitimate transaction and responsibility of the approver before allowing any token transfer. An attack can happen when changing allowance from N to M, that allows spender to transfer $N+M$ tokens. This attack can happen in case of front-running (or existence of race condition) by the approved side. As evaluated in [11], preventing the attack in `transferFrom` function is more compatible approach than `approve` method. Although it consumes more Gas[47], we will use it in our proposal for having more secure ERC20 token.

4.3. Re-entrancy Attack

As mentioned earlier, poorly implemented smart contracts can be exploited in several ways which, in some cases, can result in enormous security attacks. *The re-entrancy attack* is among those high severity security issues that resulted the attack on The DAO system, Decentralized Autonomous Organization in 2016[48]. The DAO was a form of investor-directed venture capital fund with an objective to facilitate fundraising on new ideas or new projects through crowdfunding; providing the owners with tokens, which then enable them to vote for their favourite ideas and projects. However, due to the poor design of The Dao smart contract, attackers managed to drain 86 million USD [12] off its funds. Following sections explain the overview of the DAO attack.

47. Internal pricing for running transactions in Ethereum. It prevent infinite use of computing resources.
48. https://github.com/slockit/DAO.

```
Contract SimpleDAO
{
  mapping ( address => uint256 ) public credit;
  // donate() function is used to send funds to the ↩
       DAO contract.
  function donate (address to) payable{
    credit[msg.sender] += msg.value;
  }
  function assignedCredit (address) returns (unit){
    return credit[msg.sender];
  }
  //withdraw() function is used to retrieve funds ↩
       from the DAO contract.
  function withdraw (unit amount)
  {
    if ( credit[msg.sender] >= amount )
    {
      msg.sender.call.value (amount) () ;
      credit[msg.sender] -= amount;
    }
  }
}
```

Code 1. The simplified version of the DAO smart contract that is vulnerable to the re-entrancy attack.

4.3.1. Attack Analysis. Code 1 is a simplified version of the DAO smart contract, which is vulnerable to the reentrancy attack. In order to interact with the DAO, a user first donates some Ethers to the DAO smart contract using the donate() function and the contract updates the user's balance with the exact amount it has received (Code 1 , line 6). At this point, user can withdraw the funds back by calling the withdraw() function (Code 1 , line 12). Doing so, the DAO contract sends funds back to the user (msg.sender) using the msg.sender.call.value(amount)() (Code 1 , line 16). The issue is that, the DAO smart contract first transfers the funds to the user (msg.sender) and then updates his account balance. This makes the reentrancy attack possible as a malicious user can drain funds from the contract by recursively calling the withdraw() function before the contract's state (user's balance) gets updated. To do so, attacker has to define a payable fallback() function in his smart contract. Followings explain the details of the fallback() function and how it was used in the actual DAO attack.

Fallback() Function: In Solidity programming language, a fallback() function is referred to an unnamed function, without any arguments and return value. This function gets executed automatically every time the smart contract receives Ether without any data.

By analyzing the DAO smart contract, attacker was able to discover the reentrancy bug with the withdraw() function and exploited that using the attacker smart contract (Code 2). As it can be seen in the attacker's smart contract, when the fallback function() gets executed, it calls the withdraw() function from the DAO smart contract and receives the funds back. As mentioned above, every time the attacker's contract receives plain Ether and zero data, it automatically triggers its fallback() function (which itself executes the withdraw() function). Note that in order for a function to be able to receive funds in the Solidity programming language, this function should be marked as **payable**.

As the DAO smart contract does not update the attacker's balance prior to sending funds, the withdraw() function gets executed successfully and funds are sent to the attacker's smart contract. Therefore, the attacker can recursively withdraw funds from the DAO smart contract

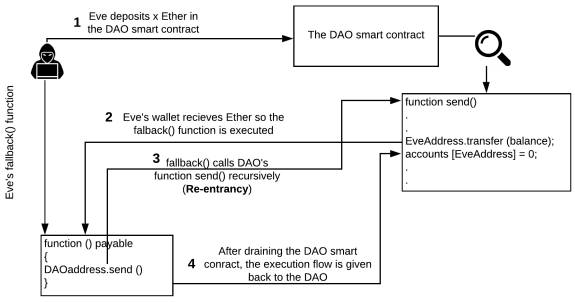

Figure 16. The re-entrancy attack diagram.

```
Import ?browser/SimpleDAO.sol ?;
Contract Attacker
{
  //assign SimpleDAO contract as dao
  SimpleDAO public dao = SimpleDAO (0x2ae...);
  address owner;

  //assign the contract creator as owner
  constructor (Attacker) public {
    owner = msg.sender;
  }

  //a fallback() function that receives no data and ↩
       withdraws funds from SImpleDAO
  function() public payable {
    dao.withdraw (dao.assignedCredit (this)) ;
  }

  //attacker sends all the funds to his personal ↩
       address
  function drainfunds() payable public{
    owner.transfer (address(this).balance);
  }
}
```

Code 2. The attacker's malicious smart contract that drains funds from the DAO smart contract.

without his balance being updated. This attack continues infinitely and it only stops if (i) the contract runs out of gas, or (ii) the victim's balance is completely drained. Once the attack is over, attacker sends all the funds to his personal address (Code 2, line 20). In the actual DAO hack, attackers were able to drain around 3.6M Ether worth approximately $70M from the DAO in a short period of time. Figure 16 visualizes the details of the re-entrancy attack.

4.3.2. Attack Mitigation. Dapp designers and developers can prevent the re-entrancy attack using the following solutions. The Withdraw() function in the victim's smart contract can be locked using the Ethereum's state machine and it can only be executed when it is in *unlocked* state. As it can be seen in (Code 3, line 12) we can lock the withdraw() function in the first line of the body of the function and unlock it before transferring the funds to the user (see Code 3, line 17). Thus, if the fallback() function starts recursive calls, it never reaches to the last line of the body of the withdraw() function where we unlock the state again. This prevents the fallback from calling the withdraw() function repeatedly. Maintaining a lock is necessary but not sufficient to prevent the reentrancy attack, DApps developers have to rely on this approach when designing smart contracts (maintaining a lock in Ethereum is not very expensive). Another preventive design decision

```
1  Contract SimpleDAO
2  {
3    ...
4    //Possible states of withdraw() function.
5    enum <@\textcolor{cyan}{Stages}@> {Opened, Locked}
6
7    Stages stage;
8
9    function withdraw (unit amount)
10   {
11     require(stage == Stages.Opened);
12     stage = Stages.Locked;
13     if ( credit[msg.sender] >= amount )
14     {
15       credit[msg.sender] -= amount;
16       msg.sender.call.value (amount) () ;
17       stage = Stages.Opened;
18     }
19   }
20 }
```

Code 3. The simplified version of the DAO smart contract that uses the state machine to lock the withdraw() function.

	Vulnerability	Mitigation	Line#
1	Overflow attack	Using SafeMath library	30-131
2	Multiple Withdrawal attack	Securing `transferFrom` method	
3	Re-entrancy attack	Preventing the attack in the `fallbac`	

TABLE 1. MAJOR ERC20 VULNERABILITIES WITH THEIR CORRESPONDING MITIGATION PLAN. THE LAST COLUMN REFERS TO THE RESPECTIVE LINE NUMBER IN THE IMPLEMENTED CODE.

the Dapp developers need to be aware of is updating the smart contract's state (user's balance) before transferring any funds to the msg.sender (user) (Code 3, line 16). To fully prevent the the re-entrancy attack, the mentioned techniques are necessary yet not sufficient.

The last method which strongly prevents the re-entrancy attack is to limit the amount of Gas that is given to the fallback() function. *Send()* and *Transfer()* are the two Solidity functions that only have 2300 Gas which is only enough for event logging and not any other action. In this case, if the attacker's fallback() function wants to log event it can, but anything else is not doable as the Gas will not be enough.

5. Proposal

As discussed in section 4, three major vulnerabilities are in the scope of this project. We summarized each security vulnerability and the corresponding mitigation plan in table 1. There are references to line numbers in our secure ERC20 implementation. Required comments have been added to the code to clarify usage of each section.

By using `SafeMath` library each arithmetic operation needs to be replaced by its `SafeMath` equivalent. For example, $a+b$ will be replaced by $a.add(b)$. This prevent integer overflow and mitigate the attack. Adding recommended code by [11] in the `transferFrom` method keeps track of transferred tokens and prevent multiple withdrawal attack. Finally, `fallback` function logs an event to track transferred Ether to the contract. This is inline with minimum recommended Gas to prevent re-entrancy attack. In general, the code can be divided into the following sections:

1) **Lines 1-27:** Declaring ERC20 interface as guideline of requires functions

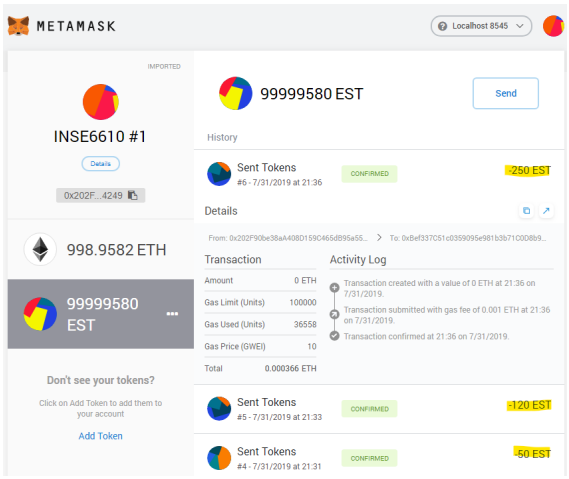

Figure 17. Compatibility of deployed token with existing wallets (MetaMask)

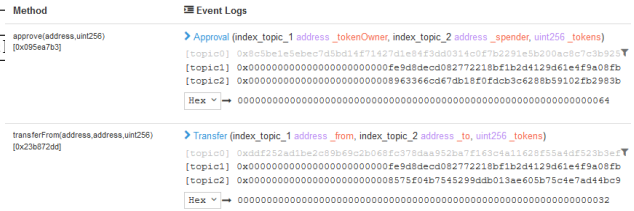

Figure 18. Logged events by the token contract after calling `Approve` or `transferFrom`

2) **Lines 30-131:** Declaring `SafeMath` library to be used in the main code. Using this library mitigates 4.1 attack (Integer overflow attack).
3) **Lines 139-324:** Implementing ERC20 functions by considering security measurements. Line 218-221 mitigates 4.2 (Multiple withdrawal attack) and using locking approach in line 161, 262 and 279 mitigates 4.3 (Re-entrancy attack).

We implemented and tested our ERC20 proposal[49] on Rinkeby test network. It can be similarly implemented on private blockchain since MetaMask supports it[50]. We initially tested our code on a private blockchain based on Ganache[51]

In terms of compatibly, we used MetaMask[52] to test our deployed token. MetaMask did not raise any transfer issue which shows compatibility of the token with existing wallets (see figure 17).

Moreover, transferring and receiving tokens trigger expected events (see figure 18)

In addition to standard ERC20 methods, this implementation extends ERC20 functionalities by introducing

49. https://rinkeby.etherscan.io/address/0xd337819ec1530c69ae9323364d4865d5659057ca#code

50. Connecting MyEtherWallet, Mist, and MetaMask to Your Private Blockchain https://bitfalls.com/2018/03/26/connecting-myetherwallet-mist-metamask-private-blockchain/

51. A private Ethereum blockchain to run tests. It gives the ability to perform all actions similar to the main chain without the cost https://www.trufflesuite.com/docs/ganache/quickstart.

52. An extension for accessing Ethereum enabled distributed applications in the browser. The extension injects the Ethereum web3 API into every website's javascript context, so that DApps can read from the blockchain.https://metamask.io/

new features:

1) **Selling tokens:** Token holders can send back tokens to the contract and receive ETH in return. They will receive ETH based on current exchange rate of the contract (managed by `exchangeRate` variable). Currently, this exchange rate is 100 tokens for 1 ETH. For example, if someone sends 200 tokens to the contract, the contract will send him/her back 2 ETH. This functionality is implemented by `sell(uint256 _tokens)` function and accepts a payable address for returning ETH to it. After each exchange, the function logs `LogSell` event which helps to track exchanged tokens for ETH.
2) **Buying tokens:** Users can call `buy()` function to purchase autonomously tokens. This function is defined as *payable*[53] and accept ETH. It then calculates number of token based on current exchange rate and increases balance of the caller. Like `sell` function, it logs `LogBuy` event for tracking purchased tokens.
3) **Withdrawing ETH:** This function can be called only by the owner of the contract. Since contract accepts ETH, token owner may use this function to transfer ETH out of the contract. Otherwise, received ETH by the contract will stick in the contract and would not be usable. There is a complementary function (i.e., `contractBalance()`) to get current ETH balance of the contract. This function also can be called by the owner to see how many ETH are under the contract. Transferring ETH out of the contract also logs `LogWithdrawal` event which logs moved ETH from the contract.

6. Future Works

As future works, we are interested to see how many ERC20 tokens are impacted by one of these attacks. It requires (i) creating a full Ethereum node that have local copy of the blockchain for faster look up (ii) developing a script to scan the blockchain for ERC20 tokens (iii) using a complementary script to convert byte codes to *Solidity* human readable codes (iv) searching for vulnerable codes and discovering impacted tokens. After finding vulnerable ERC20 tokens, the question would be how to remediate them. Remediation could be in form of migration since after deploying an ERC20 token on the blockchain, it becomes immutable. So, if the token contract being vulnerable to one of these security threads, developers needs to create a new ERC20 token and migrate users to the new one. The new token will have different address and revised code to fix identified vulnerabilities[54]. Nonetheless, migration of ERC20 code to a secure version would not be convenient due to several technical considerations (*i.e.,* transaction cost, data structure, data recovery, number of token holders, *etc.*). This also includes updating any trading platform listing ERC20 tokens. Thus, it would be extremely challenging to estimate associated costs with this migration. Formalization of it could be another topic to discover in further research.

53. Payable functions provide a mechanism to collect/receive funds in ethers. Payable functions are annotated with payable keyword.

54. There would be one exception in case of using upgradable ERC20 tokens. Developers can point the base contract to the new address and make the migration transparent for users. Although having upgradable ERC20 token would have some trade-offs that needs to be considered (*i.e.,* risks of the data separation pattern, risks of delegatecall-based proxies, *etc.*).

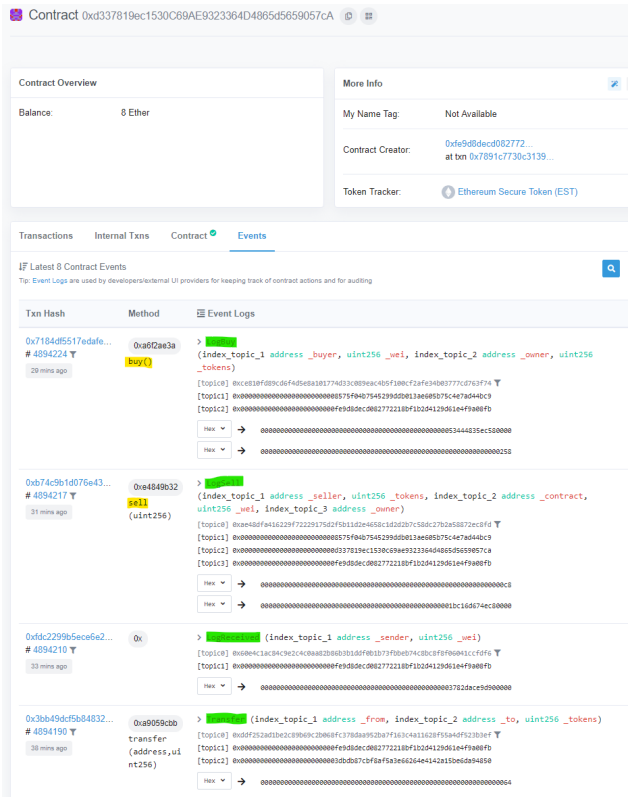

Figure 19. Added functionalities to the ERC20 contract. Each new feature logs corresponding event to track transactions of the token.

7. Conclusion

This paper focuses on the ERC20 tokens; Ethereum's most popular token standard. The development of smart contracts has proven to be error-prone in practice, and as a result, contracts deployed on public platforms are often riddled with security vulnerabilities. Exploited by the attackers, these vulnerabilities can often lead to major security incidents which introduce great cost due to the immutability characteristics of the blockchain technology. In this paper, we carefully select three significant vulnerabilities based on their attack vector and broader impact on the Ethereum blockchain. We thoroughly discuss the technical details of the vulnerabilities, the circumstances of the incidents together with their impacts, mitigations, and the broader lessons we learn from these incidents. Eventually, we propose a secure version of ERC20 token which is not vulnerable to any of the attack discussed.

References

[1] Ethereum. Ethereum project repository. https://github.com/ethereum, May 2014. [Online; accessed 10-Jul-2019].

[2] Solidity — solidity 0.4.25 documentation. https://solidity.readthedocs.io/en/v0.4.25/. (Accessed on 07/23/2019).

[3] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making Smart Contracts Smarter. https://dl.acm.org/ft_gateway.cfm?id=2978309&ftid=1805715&dwn=1&CFID=86372769&CFTOKEN=b697c89273876526-8CBDF39B-A89A-31D2-F565B24919F796C6, October 2016. [Online; accessed 20-Jul-2019].

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

[5] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://blog.peckshield.com/2018/04/22/batchOverflow/, April 2018. [Online; accessed 26-Jun-2019].

[6] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://blog.peckshield.com/2018/04/25/proxyOverflow/, April 2018. [Online; accessed 30-Jun-2019].

[7] 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-Jul-2019].

[8] 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-Jul-2019].

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

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

[11] Reza Rahimian, Shayan Eskandari, and Jeremy Clark. Resolving the multiple withdrawal attack on erc20 tokens. *IEEE SECURITY & PRIVACY ON THE BLOCKCHAIN*, 2019.

[12] Klint Finley. A $50 million hack just showed that the dao was all too human — wired. https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/, September 2016. (Accessed on 07/22/2019).