Securing ERC20 Tokens in Ethereum blockchain

Abstract—**ERC20 is one of the token standards in the Ethereum blockchain that is widely accepted in the industry. They are subset of smart contracts and similarly vulnerable to security flaws. In this paper, we (i) examine ERC20 vulnerabilities and propose a secure implementation, (ii) consider best practices to improve performance of the proposal for real-world scenarios (*i.e.,* ICOs, DApps, *etc.*), (iii) review the proposed Solidity code by 7 auditing tools and compare detected security issues with the top ten ERC20 tokens, (iv) provide list of 89 vulnerabilities and best practices that can be eventually turned into a security analysis tool for ERC20 tokens.**

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

1. Introduction

Ethereum blockchain project was launched in 2014 by announcing Ether (ETH) as its protocol-level cryptocurrency [1], [2]. It allows users to build decentralized applications (DApps) in the form of smart contracts. DApps can use ETH or issue their own custom currency-like tokens. The Ethereum community accepted the most popular token standard called ERC20[1]. It is standardized version of smart contracts which allows other applications (*e.g.,* Wallets, DApps, *etc.*) to interact and use exposed methods. ERC20 does not provide a concrete implementation of methods and only guidelines on how each method should be implemented (such as name of the method, parameters, return types). This gives developers flexibility of coding based on their DApps requirements. In practice however, development of smart contracts has been proven to be error-prone, and as a result, smart contracts are often riddled with security vulnerabilities. Previous research showed that at about 45% of existing smart contracts are vulnerable [3]. From about 2.5M[2] smart contracts on the Ethereum network, 260K[3] are ERC20 tokens which may be vulnerable to security threats. Additionally, tokens are financial assets and some of them have considerable value that exceed the value of ETH itself (*e.g.,* PAX Gold[4], MKR[5] and XIN[6]). They might be audited by trusted parties and existence of security threats may lead to hesitation of auditors.

1. https://eips.ethereum.org/EIPS/eip-20
2. [2020-05-03] https://reports.aleth.io
3. [2020-05-03] https://etherscan.io/tokens
4. [2020-05-02] https://www.paxos.com/paxgold/
5. [2020-05-02] https://makerdao.com/en/
6. [2020-05-02] https://mixin.one/

Contributions. Similar to any new technology, different layers of Ethereum (*e.g.,* Application, Contract, *etc.*) expose security vulnerabilities that caused more than US$100M financial loss by smart contracts[4], [5], [6], [7], [8], [9]. This motivates us to (i) examine ERC20 vulnerabilities and their mitigation techniques, (ii) propose a Solidity[7] code that addresses discussed vulnerabilities and can be used as a template to deploy secure ERC20 tokens, (iii) integrate smart contract best practices to optimize performance of the code for commercial uses (*e.g.,* ICOs), (iv) use auditing tools to compare security of the code with the top ten Ethereum tokens, (v) provide list of potential threats to assist auditors for faster assessment of associated risks to ERC20 tokens and eventually automate the auditing process.

2. ERC20 security vulnerabilities

ERC20 tokens are subset of smart contracts and vulnerable in a similar way. We therefore examine attack vector and broader impact of smart contract vulnerabilities [10], [11], [12], [13], [14] to check their applicability on ERC20 tokens. For each vulnerability, we (i) briefly explain technical details, (ii) ability to affect ERC20 tokens, (iii) discuss mitigation technique. We ultimately put all of the mitigation techniques together and propose a secure ERC20 code that is not vulnerable to any of discussed threats (See Section 4).

Among the layers of Ethereum blockchain, our focus is on the *Contract layer* in which smart contracts are executed (See figure 1). The presence of security vulnerability in supplementary layers affect the entire Ethereum blockchain, not necessarily ERC20 tokens. Therefore, vulnerabilities in other layers are assumed to be out of the scope (*e.g., Indistinguishable chains* at Data layer, *51% hashrate* at Consensus layer, *Unlimited nodes creation* at Network layer and *Web3.js Arbitrary File Write* at Application layer). Moreover, due to the use of the recent version of Solidity compiler, we do not discuss the vulnerabilities identified in the outdated compiler versions, for example:

- *Constructor name ambiguity* in versions before 0.4.22.
- *Uninitialized storage pointer* in versions before 0.5.0.
- *Function default visibility* in versions before 0.5.0
- *Typographical error* in versions before 0.5.8.
- *Deprecated solidity functions* in versions before 0.4.25.
- *Assert Violation* in versions before 0.4.10.
- *Under-priced DoS attack* before EIP-150 & EIP-1884.

7. The most common programming language in Ethereum to develop smart contracts. https://solidity.readthedocs.io

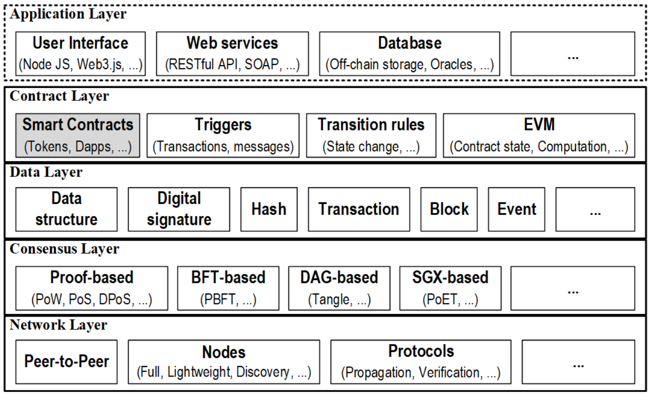


Figure 1: Architecture of Ethereum blockchain, including interactive environment (*i.e.,* Application layer). ERC20 tokens falls under *Smart Contracts* category in *Contract Layer*. The security vulnerabilities of smart contracts can be extended to tokens as well. There might be also security vulnerabilities in other layers that we focus only on ERC20 tokens.

2.1. Arithmetic Over/Under Flows. It is well known issue in many programming languages called *integer overflow*[8]. It was exploited in April, 2018 and some exchanges[9] had suspended deposits and withdrawals of all tokens, especially for Beauty Ecosystem Coin (BEC[10]) that was targeted by this exploit. Although BEC developers had considered most of the security measurements, only line 261[11] was vulnerable[5]. The attacker was able to pass a combination of input values to transfer large amount of tokens[15]. It was even larger than the initial supply of the token, allowing the attacker to take control of token finance and manipulate the price. In Ethereum, integer overflow does not throw exception at runtime. This is by design and can be prevented by using `SafeMath`[12] library where in `a+b` will be replaced by `a.add(b)` and throws an exception in case of arithmetic overflow. This library is offered by OpenZeppelin[13] and has become industry standard. We use it in all arithmetic operations to catch over/under flows.

2.2. Re-entrancy. It is among high severity vulnerabilities that resulted the attack on DAO[14] in 2016. An attacker could manage to drain US$50M off the token funds [4], [16]. ERC20 tokens would also be vulnerable to this attack if exchanging tokens for ETH is supported. An attacker can call the exchange function (e.g., `sell(tokens)`) to sell token and get back equivalent in ETH. However, before reaching to the end of the function and updating balances, the function might transfer control to the caller which allows the same function to be invoked over and over within the same transaction. This can be continued until draining all ETH of the token contract. The attack is known as same-function re-entrancy and could have three variants: Cross-function re-entrancy, Delegated re-entrancy and Create-Based re-entrancy[17]. Mutex[18] or CEI[19] techniques can be used to prevent it. In Mutex, a state variable is used to lock/unlock transferred ETH by the lock owner (*i.e.,* token contract). The lock variable fails subsequent calls until finishing the first call and changing requester balance. CEI updates the requester balance before transferring any fund. All interactions (*i.e.,* external calls) happen at the end of the function and prevents recursive calls. Although CEI does not require a state variable and consumes less Gas, we use Mutex in addition to CEI. This protects token contract against Cross-function re-entrancy when attacker calls a different function than the initial function. In the proposal, `noReentrancy` modifier enforces Mutex and CEI is considered in the implementation of critical functions.

2.3. Unchecked return values. In Solidity, sending ETH to external addresses are commonly performed by: (1) `call.value()`, (2) `transfer()`[15] or (3) `send()`. The `transfer()` method reverts all changes if the external call fails [20]. Other two methods are simply return a boolean value and manual check is required to revert transaction to the initial state. Before *Istanbul* hard fork[21], `transfer()` was the preferred way of sending ETH by forwarding only 2300 Gas. It prevents recursive calls and mitigates re-entrancy attack. EIP-1884[16] has increased Gas cost of some opcodes that fails this method[17]. The best practice is now not to rely on Gas and use `call.value()` method[22], [23]. Since all remaining Gas will be sent by this command, one of re-entrancy mitigations (*i.e.,* Mutex or CEI) must be considered. We use `call.value()` in `sell()` and `withdraw()` functions and check the returned value to revert failed fund transfers.

2.4. Balance manipulation. General assumption to receive ETH by smart contracts is via payable functions[18] (*i.e.,* `receive()`, `fallback()`, *etc.*), however, it is possible to send ETH without triggering payable functions, for example via `selfdestruct(contractAddress)` that is initiated by another contract. This allows forcing ETH and manipulate contract balance[24]. Hence, using checks like `address(this).balance` provides a relative security risk. To prevent exploiting this vulnerability, contract logic should avoid using exact values of the contract balance and keeps track of the known deposited ETH by a new state variable. Although we use `address(this).balance` in our implementation, but we do not check exact value of it (*i.e.,* `address(this).balance == 0.5 ether`). We only check whether the contract has enough ETH to

8. http://bit.ly/3cJDqX6
9. OKEx, Poloniex, HitBTC and Huobi Pro
10. http://bit.ly/2TIartO
11. http://bit.ly/38BwcRI
12. http://bit.ly/2VYuoPU
13. http://bit.ly/2Tx8DVL
14. It was a form of investor-directed venture capital fund to facilitate fundraising on new ideas or new projects through crowdfunding; providing the owners with tokens, which then enable them to vote for their favorite ideas and projects. https://github.com/slockit/DAO.
15. http://bit.ly/39C3x01
16. http://bit.ly/2U2sHi3
17. After *Istanbul* hard-fork, `fallback()` function consumes more than 2300 Gas if called via `transfer()` or `send()` methods.
18. http://bit.ly/38FRRrQ

send out or not. Therefore, there is no need to use a new state variable and consume more Gas to track contract's ETH. However, for developers who are interested to track it manually, there would be `contractBalance` variable to use. Two complementary functions are also considered to get current contract balance and check unexpected received ETH (*i.e.,* `getContractBalance()` and `unexpectedEther()`).

2.5. Public visibility. In Solidity, visibility of functions are `Public` by default and they can be called by any external user/contract. It is recommended to always specify the visibility of all functions to prevent attacks like what happened to Parity MultiSig Wallet [8]. An attacker was able to call public functions and reset the ownership address of the contract. It caused draining of the wallets to the tune of $31M. To prevent such attacks, we explicitly define visibility of each function. Interactive functions (*e.g.,* `Approve()`, `Transfer()`, *etc.*) are publicly accessible per specifications of ERC20 standard.

2.6. Multiple withdrawal. This protocol-level issue was originally raised in 2017 [25], [26] and originating from ERC20 definition. It can be considered as *Transaction-ordering* or [27] or *Front-running* [28] attack. There are two functions (*i.e.,* `Approve()` and `transferFrom()`) that can be used to authorize a third party for transferring tokens on behalf of someone else. Using these functions in an undesirable situation (*i.e.,* Front-running or race-condition[19]) could result in condition that allows attacker to transfer more tokens than the owner ever wanted. There are several suggestions to mitigate this attack, however, securing `transferFrom()` method is the effective one while adhering specifications of the ERC20 standard[29]. We added a new state variable to the `transferFrom()` function to track transferred tokens and mitigate the attack.

2.7. State variable manipulation. `DELEGATECALL` opcode in Ethereum enables to invoke external functions and execute them in the context of calling contract (*i.e.,* Invoked function can modify state variables of the caller). This makes it possible to deploy libraries once and reuse the code in different contracts. However, ability to manipulate internal state variables by external functions can lead to hijacking of the entire contract as it happened in Parity Multisig Wallet [9]. Preventive technique is to use `Library` keyword in Solidity to force the code to be stateless[20] [30]. There are two types of Library: Embedded and Linked. Embedded libraries have only internal functions, in contrast to linked libraries that have public or external functions. Deployment of linked libraries generates a unique address on the blockchain while the code of embedded libraries will be added to the contract's code [31]. In the proposal, there is only one library, `SafeMath`, that is defined as embedded library. We use `Library` keyword to declare it and has only internal functions. Therefor its code will be added to the ERC20 contract's code and EVM uses `JUMP` statement instead of `DELEGATECALL`. As a result, we do not use `DELEGATECALL` and will be safe to this vulnerability.

2.8. Frozen Ether. Smart contracts can receive ETH similar to user accounts. In order to send the received ETH out of the contract, it is necessary to use withdrawal functions, so that the ETH does not get stuck in the contract as it happened in the case of Parity Wallet [32]. We define `withdraw()` function which allows the owner to transfer ETH out of the contract. The `sale()` function also makes it possible to transfer ETH during token exchange.

2.9. Unprotected SELFDESTRUCT. As it happened in Parity wallet [9], Self-destruct method is used to kill the contract and associated storage. It is recommended to get approval by multiple parties before running the method. We do not use this method and the deployed ERC20 tokens will be active on the blockchain forever.

2.10. Unprotected Ether Withdrawal. Improper access control may allow unauthorized persons to withdraw ETH from smart contracts (as it happened in Rubixi[21]). Therefore, withdrawals must be triggered by only authorized accounts. We use `onlyOwner` modifier to enforce authentication on `withdraw()` function before sending out any funds.

3. ERC20 best practices

Best practices are techniques or rules that are accepted to develop the most effective smart contract. They used to maintain quality of the code and a standard way of creating ERC20 tokens. Significant set of best practices have been accepted by the Ethereum community to proactively prevent known vulnerabilities [33]. We examine most of them and integrate in the proposal.

3.1. Compliance with ERC20. According to ERC20 specifications, all 6 methods and 2 events must be implemented and they are not optional. Moreover, ignoring them will cause failed function calls by other applications (*i.e.,* crypto-wallets, crypto-exchanges, web services, etc). They expect to invoke these methods when querying transferred tokens or updating balance of accounts in the UI. Tokens that are not implementing all methods (*e.g.,* `approve()` or `transferFrom()`) will not be fully ERC20-compliant. There might be reasons for this, but it makes those token partially ERC20-compliant. We implement all ERC20 required methods in addition to some complementary functions such as `sell()` and `buy()`. `sell()` allows token holders to exchange tokens for ETH and `buy()` accepts ETH by adjusting buyer's token balance.

19. Performing two or more operations at the same time due to nature of the blockchain.

20. Data are passed as inputs to functions and passed back as outputs. Libraries do not have any storage that makes such attacks unlikely.

21. https://bit.ly/2yrYP7P

3.2. Firing events. In ERC20 standard, there are two defined events: `Approval` and `Transfer`. The first event, logs any successful allowance change by token holders and the latter one, logs successful token transfers by `transfer()` or `transferFrom()` methods. These two events must be fired to notify external application on occurred changes. They might use them to update balances, show UI notifications or check new token approvals. In addition to the above logs, we define 6 extra events that are `Buy`, `Sell`, `Received`, `Withdrawal`, `Change` and `Pause`. They can be used to watch for token events and react accordingly.

3.3. External visibility. There are two types of *function call* in Solidity[34]: (i) Internal (ii) External. Internal function calls expect arguments to be in memory and EVM copies arguments to memory. This is because internal calls use `JUMP` opcodes instead of creating *EVM call*[22]. Conversely, External function calls create *EVM call* and can read arguments directly from `calldata` space. It is cheaper than allocating new memory and designed as a read-only byte-addressable space where the data parameter of a transaction or call is held[35]. As a best practice, using External functions are recommended if we expect that the function to be called externally. We consider this recommendation by replacing `Public` visibility as `External`.

3.4. Fail-Safe Mode. Off-chain computations can be used to performs some self-checks on the ERC20 tokens. In case of detected anomaly/attack, functionality of the token can be put on hold until further investigations. To pause all functionalities, owner of the token can call `pause()` function. It then sets a lock variable and `notPaused` modifier forces it by throwing exception. We apply `notPaused` modifier on all external functions (*e.g.,* `transfer()`, `sell()`, *`etc.`*) to make sure that it would be safe to process external calls and the token is not paused.

3.5. Global or Miner controlled variables. Since malicious miners have the ability to manipulate global Solidity variables (*e.g.,* `tx.origin`, `block.timestamp`, `block.number`, `block.difficulty`, *etc.*), it is recommended not to use these variables. For example, `tx.origin` can be compromised by a phishing attack. It is safer to use `msg.sender` which returns the transaction caller and not the original initiator. We do not use any of these variables for conditional execution, authentication or as the source of randomness.

3.6. Proxy contract. Using proxy is one of the approaches to build upgradable ERC20 tokens. The proxy contract forwards function calls to another contract that can be updated [36], [37]. Since the updated contract code can have vulnerabilities, the use of proxy contracts is not recommended. We do not use any proxy contract and all codes are included in the token contract.

22. Also known as "message call" when a contract calls a function of another contract.

3.7. DoS with unexpected revert. In case of failure when sending fund to a large number of recipients, the entire transaction may fail and no fund will be transferred. This issue may occur due to sending ETH to a contract that does not have `fallback()` function or reverts ETH transfers in the `fallback()` function. To prevent this situation, it is recommended to avoid transferring ETH to multiple addresses in a single transaction. Instead, isolate each external call into its own user-initiated transaction [38]. In the proposal, we use `sell()` function to send ETH back to token sellers. There is no batch transfer in the function and in case of failure, it affects only one seller.

3.8. DoS with block gas limit. The use of loops in contracts is not efficient and requires considerable amount of Gas to execute. It might also cause DoS attack since blocks has a *Gas limit*. If execution of a function exceeds the block gas limit, all transactions in that block will fail. Hence, we do not use loops and rely on `mappings` variables. They store data in collection of key value pairs and are more efficient.

4. Proposal

Considering discussed vulnerabilities in section 2, we propose as secure ERC20 code that is not vulnerable to any of them. It has been deployed on the Mainnet and the Solidity code is available on Etherscan[23]. Developers can refer to each mitigation technique separately to address a specific attack in their customized version. Required comments have been also added to clarify usage of each part. Standard functionalities of the token (*i.e.,* `approve()`, `transfer()`, *etc.*) have been tested by MetaMask[24] and no issue were raised. It could interact with the token successfully[25] and triggers expected events[26] after transferring and receiving tokens. In addition to standard ERC20 methods, we introduce the following complementary features:

1) **Selling tokens:** By using `sell()` function, token holders can send back tokens to the contract and receive ETH in return. Received ETH is based on the current exchange rate which is managed by `exchangeRate` variable. By default, this rate is 100 tokens for 1 ETH. For example, if someone sends 200 tokens to the contract, the contract sends back 2 ETH. After each exchange, `Sell` event tracks exchanged tokens. This feature is a financial advantage for new ERC20 tokens and reduces buyers doubts. They can return purchased token at any time and receive the equivalent in ETH. Another option for them is to wait for the token to be listed by crypto-exchanges (if it ever happens). Otherwise, they would not be able to exchange tokens if this feature is not support by the token contract.

23. https://bit.ly/2xvpnoh
24. 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/
25. http://bit.ly/2IZYzPf
26. http://bit.ly/2Ub3vG9

2) **Buying tokens:** Users can call `buy()` function to purchase autonomously tokens. This function is defined as *payable*[27] and accepts ETH. It calculates the equivalent of tokens based on the current exchange rate. It then increases balance of the buyer and logs `Buy` event for tracking of purchased tokens.
3) **Withdrawing Ether:** This function can be called only by the contract owner. Since the contract accepts ETH, token owner may use `withdraw()` function to transfer ETH out of the contract. Otherwise, received ETH get stuck in the contract and would not be transferable. Transferring ETH out of the contract logs `Withdrawal` event.

Supporting these extra features would be a financial advantage for new tokens and makes them independent of crypto-exchanges. All the required functionalities are directly supported by the token contract and no additional external services are required.

5. Formal verification

Code verification before launching ERC20 tokens could prevent human errors and reveals the presence of vulnerabilities. We use the following publicly available tools [39] to detect security flaws in the proposal[28]:

[1] EY Review Tool [29] by Ernst & Young Global Limited.
[2] SmartCheck[30] by SmartDec.
[3] Securify[31] by ChainSecurity.
[4] ContractGuard[32] by GuardStrike.
[5] MythX[33] by ConsenSys.
[6] Slither Analyzer[34] by Crytic.
[7] Odin[35] by Sooho.

A total of 89 audits have been conducted by these auditing tools, including the best practices in addition to security vulnerabilities. The results are summarized in Table 1 and sorted by Smart Contract Weakness Classification (SWC[36].). Knowledge-base of each tool is used to map audits to the corresponding SWC registry [40], [41], [42], [43], [44]. Since each tool employs different methodology to analyze smart contracts (*e.g.,* comparing with violation patterns, applying set of rules, using static analysis, etc), there would be some false positives. After ignoring them, the average percentage of passed checks for the code reaches to 96%. The following are some examples of ignored false positives:

- *MythX* detects *Re-entrancy attack* in *noReentrancy* modifier. In solidity, modifiers are used to add features or apply some restriction on function[45]. Using *noReentrancy* modifier is a known technique (Mutex) to mitigate *Re-entrancy attack*[46]. Since other tools have not identified such a case, it can be considered as false positive.
- *EY review tool* considers `decreaseAllowance` and `increaseAllowance` as standard ERC20 functions and if not implemented, recognizes the code as vulnerable to *front-running* attack. These two functions are not defined in the ERC20 standard[47] and considered only by this tool as standard ERC20 functions. There are other methods to prevent the attack while adhering ERC20 specifications[29]. The tool also detects the *Overflow* attack, which is already addressed by using the `SafeMath` library. Another identified issue is *Funds can be held only by user-controlled wallets*. It advises to prevent any token transfer to Ethereum addresses that belong to smart contracts. However, interacting with ERC20 token by other smart contracts was one of the main motivations of the ERC20 standard.
- *SmartCheck* does not recommend to use `SafeMath` and advises to explicitly check where it is really needed. Another identified issue is *using private modifier*. They mention in the knowledge-base that "miners have access to all contracts' data and developers must account for the lack of privacy in Ethereum". However, they do not provide an alternative. A workaround could be to use cryptographic techniques for maintaining privacy on the current version of Ethereum. Also, using `approve()` function is not recommended due to front-running attack while there are preventive techniques for it. Despite EIP-1884, the tool still recommends using of `transfer()` method with stipend of 2300 gas.
- The proposal could not pass *Re-entrancy* check of *Securify* while both CEI and Mutex are implemented. It identifies `noReentrancy` modifier as unsafe due to unrestricted writes [48]. Modifier are not accessible by users and are recommended approach to prevent *Re-entrancy*.
- Using `SafeMath` is detected as *Delegatecall to untrusted callee* vulnerability by *SmartCheck*. In solidity, embedded libraries are called by JUMP commands instead of Delegatecall. Therefore, excluding embedded libraries from this check might improve accuracy of the tool.

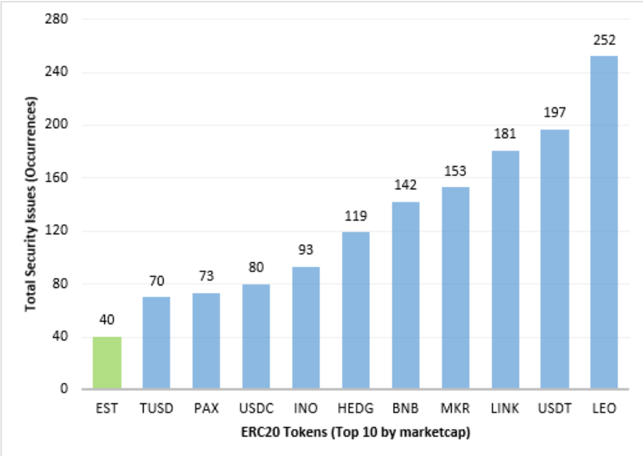
Some tools also need to be updated to meet the latest standards or consider best practices. for example:

- Current version of analyzers[49] in *Slither* detect two *low level call* vulnerabilities in the proposal. This is due to using of `msg.sender.call.value()` that is recommend way of transferring ETH after *Istanbul* hardfork (EIP-1884). Therefore, adapting analyzers to new standards can improve accuracy of the security checks.
- *EY review tool* checks for maximum 50000 gas in `approve()` and 60000 in `transfer()` method. We could not find corresponding SWC registry or standard recommendation on these limitations.
- Locking solidity version to 0.5.11 is detected by *Odin* as *Outdated compiler version*. We have used this version due to its compatibility with all auditing tools. Furthermore, other tools have not identified such an issue.

27. Payable functions provide a mechanism to collect/receive ETH.
28. We could not use some other tools (*e.g.,* Oyente) due to support for lower versions of solidity compiler.
29. https://review-tool.blockchain.ey.com
30. https://tool.smartdec.net
31. https://securify.chainsecurity.com
32. https://contract.guardstrike.com
33. https://mythx.io
34. https://github.com/crytic/slither
35. https://odin.sooho.io/
36. Classifies security issues in smart contracts. https://swcregistry.io/

ID	SWC	Vulnerability / Best practice (Audits)	Mitigation / Recommendation	EY Review	Smart Check	Secu rify	MythX (Mythril)	Contract Guard	Slither	Odin
1	100	Function default visibility	Specifying function visibility		✓		✓	✓		✓
2	101	Integer Overflow and Underflow	Using SafeMath	⊕	✓		✓	✓		✓
3	102	Outdated Compiler Version	Using proper Solidity version	✓	✓	✓	✓	✓	✓	×
4	103	Floating Pragma	Locking the pragma version		✓	✓	✓		✓	✓
5	104	Unchecked Call Return Value	Checking call() return value	✓		✓	✓	✓	✓	✓
6	105	Unprotected Ether Withdrawal	Authorizing trusted parties		×		✓		✓	✓
7	106	Unprotected SELFDESTRUCT Instruction	Approving by multiple parties			✓	✓		✓	✓
8	107	Re-entrancy	Using CEI or Mutex		✓	⊕	⊕	⊕	✓	✓
9	108	State variable default visibility	Specifying variable visibility	✓	×	✓	✓	✓		✓
10	109	Uninitialized Storage Pointer	Initializing upon declaration			✓	✓	✓	✓	✓
11	110	Assert Violation	Using require() statement		✓		✓			✓
12	111	Use of Deprecated Solidity Functions	Using new alternatives		✓		✓	✓	✓	✓
13	112	Delegatecall to untrusted callee	Using for trusted contracts		×	⊕	✓	✓	✓	✓
14	113	DoS with Failed Call	Avoid multiple external calls		✓		✓	✓		✓
15	114	Transaction Order Dependence	Preventing race conditions	⊕		✓	✓			✓
16	115	Authorization through tx.origin	Using msg.sender instead	✓	✓	✓	✓	✓	✓	✓
17	116	Block values as a proxy for time	Using oracles instead of block number	✓	✓	✓	✓	✓		✓
18	117	Signature Malleability	Not using signed message hash				✓			✓
19	118	Incorrect Constructor Name	Using constructor keyword		✓		✓			✓
20	119	Shadowing State Variables	Remove any variable ambiguities			✓	✓	✓	✓	✓
21	120	Weak Sources of Randomness from Chain Attributes	Not using block variables	✓	✓		✓	✓		✓
22	121	Missing Protection against Signature Replay Attacks	Storing every message hash				✓			✓
23	122	Lack of Proper Signature Verification	Using alternate verification schemes				✓			✓
24	123	Requirement Violation	Allowing all valid external inputs		✓	✓	✓			✓
25	124	Write to Arbitrary Storage Location	Controlling write to sensitive storage		✓	✓	✓			✓
26	125	Incorrect Inheritance Order	Inheriting from more general to specific				✓			✓
27	126	Insufficient Gas Griefing	Allowing trusted forwarders		✓					✓
28	127	Arbitrary Jump with Function Type Variable	Minimizing use of assembly		✓	✓	✓		✓	✓
29	128	DoS With Block Gas Limit	Avoiding loops across the entire data	✓	✓	✓	✓	✓	✓	✓
30	129	Typographical Error	Using SafeMath				✓			✓
31	130	Right-To-Left-Override control character (U+202E)	Avoiding U+202E character			✓	✓	✓	✓	✓
32	131	Presence of unused variables	Removing all unused variables		✓	✓		✓	✓	⊕
33	132	Unexpected Ether balance	Avoiding strict Ether balance checks		✓	✓		✓	✓	✓
34	133	Hash Collisions With Variable Length Arguments	Using abi.encode() instead							✓
35	134	Message call with hardcoded gas amount	Using .call.value(...)("")		⊕	×		✓		✓
36	135	Code With No Effects	Writing unit tests to verify correct behavior		✓					✓
37	136	Unencrypted Private Data On-Chain	Storing private data off-chain							✓
38	BP	ERC20 compliance	Implementing 6 functions and 2 events	✓	✓	✓		✓	✓	
39	BP	Number of external functions	Minimizing external functions	✓	✓	✓				
40	BP	Token decimal	Adding a token decimal declaration	✓						
41	BP	Token name	Adding a token name variable	✓						
42	BP	Token symbol	Adding a token symbol variable	✓						
43	◯	Allowance decreases upon transfer	Decreasing allowance in transferFrom()	×						
44	◯	Allowance function returns an accurate value	Returning only value from the mapping	✓						
45	BP	Allowance spending is possible	Ability of token transfer by transferFrom()	✓						
46	BP	The Approval event is correctly logged	Emitting Approval event	✓						
47	◯	It is possible to cancel an existing allowance	Possibility of setting allowance to 0	✓	✓					
48	BP	The decreaseAllowance definition follows the standard	Defining decreaseAllowance function	⊕						
49	BP	The increaseAllowance definition follows the standard	Defining increaseAllowance function	⊕						
50	◯	A transfer with an insufficient amount is reverted	Checking balances in transfer()	✓					✓	
51	BP	Uninitialized state variables	Initializing all the variables	✓	✓			✓	✓	
52	◯	Upon sending funds, the sender's balance is updated	Updating balances in transfer()	✓						
53	◯	The Transfer event correctly logged	Emitting Transfer event	✓						
54	BP	Transfer to the burn address is reverted	Checking transfer to 0x0	✓						
55	◯	Transfer an amount that is greater than the allowance	Checking in transferFrom()	✓						
56	BP	Emitting event when state changes	Emitting Change event	×						
57	BP	Source code is decentralized	Not using hard-coded addresses	✓	✓					
58	◯	Risk of short address attack is minimized	Using recent Solidity version	✓				✓		
59	◯	Function names are unique	No function overloading	⊕					✓	
60	BP	Funds can be held only by user-controlled wallets	Checking for address code	×						
61	BP	Code logic is simple to understand	Avoiding code nesting	✓	✓					
62	BP	All functions are documented	Using NatSpec format	✓						
63	BP	Using only high-level programming language	Not using inline-assembly codes	✓	✓	✓		✓	✓	
64	BP	Acceptable gas cost of the approve() function	Checking for maximum 50000 gas	×						
65	BP	Acceptable gas cost of the transfer() function	Checking for maximum 60000 gas	×						
66	BP	Use of "Pull over Push" efficiency pattern	Allowing user to pull the funds	✓	✓					
67	BP	Use of unindexed arguments	Using indexed events' arguments		✓			✓	✓	
68	◯	Using miner controlled variables	Avoiding now, block.timestamp, etc	✓	✓	✓	✓	✓	✓	
69	◯	Use of return in constructor	Not using return in contract's constructor		✓					
70	◯	Throwing exceptions in transfer() and transferFrom()	Returning true after successful execution		✓				✓	
71	BP	Locked money	Implement a withdraw function or reject payments		✓				✓	
72	BP	Malicious libraries	Not using modifiable third-party libraries		✓					
73	BP	Payable fallback function	Adding fallback() function to receive Ether		✓			✓		
74	BP	Prefer external to public visibility level	Replacing public with external if not used locally		✓				✓	
75	◯	Call with hard-coded gas amount (EIP1884)	Not using transfer() or send() functions		×		✓			
76	BP	Error information in revert condition	Adding error description					✓		
77	BP	Freezing Ether	Adding functions to send Ether out		✓			✓		
78	BP	Complex Fallback	Logging operations in the fallback function					✓		
79	BP	Function Order	Following fallback, external, etc					✓		
80	BP	Visibility Modifier Order	Specifying visibility first and before modifiers					✓		
81	BP	Non-initialized return value	Not specifying return for functions without output		✓			✓		
82	◯	Tautology or contradiction	Fixing comparison that are always true or false						✓	
83	◯	Divide before multiply	Ordering multiplication prior division						✓	
84	◯	Unchecked Send	Ensure that the return value of send() is checked						✓	
85	BP	Builtin Symbol Shadowing	Renaming variables						✓	
86	BP	Low level calls	Checking successful return value from call()						×	
87	BP	Conformance to naming conventions	Following the Solidity naming convention						✓	
88	BP	Too many digits	Using scientific notation						✓	
89	BP	State variables that could be declared constant	Adding constant attribute						✓	
	96% average success rate by considering false positives as passed			**87%**	**91%**	**96%**	**100%**	**100%**	**97%**	**97%**

TABLE 1: Auditing results of 7 smart contract analysis tools on the proposed ERC20 code. 96% average success rate after considering the *false positives* as *passed*. (LEGEND. BP=Best practice, ✓=Passed audit, ⊕=False positive, ×=Failed audit, Empty=Not supported audit by the tool, ◯=Tool specific audit (No SWC registry)



	Auditing tool							
ERC20 Token	**EY Review**	**Smart Check**	**Secu rify**	**MythX (Mythril)**	**Contract Guard**	**Slither**	**Odin**	**Total issues**
EST	9	11	4	2	10	2	2	**40**
TUSD	20	11	2	1	14	16	6	**70**
PAX	16	9	6	4	16	13	9	**73**
USDC	17	9	6	5	18	15	10	**80**
INO	11	10	14	8	14	24	12	**93**
HEDG	10	28	11	1	29	24	16	**119**
BNB	13	21	12	13	41	39	3	**142**
MKR	11	27	38	9	16	34	18	**153**
LINK	12	27	38	9	16	34	18	**181**
USDT	12	29	8	17	46	55	30	**197**
LEO	32	25	8	23	70	75	19	**252**

TABLE 2: Security flaws detected by 7 auditing tools in EST (the proposal) compared to top 10 ERC20 tokens. EST has the lowest reported security issues (occurrences).

6. Comparing audits

In section 5, we checked security of the proposed ERC20 token (EST) by using 7 auditing tools. In this section, we repeat the same process on the top ten tokens based on their market cap[50]. Result of all these evaluation has been summarized in table 2 by considering false positives as failed audits. This provide the same evaluation conditions across all tokens. Since each tool use different analysis methods, number of occurrences are considered for comparisons. For example, MythX detects two re-entrancy attack in EST, therefore, two occurrences are counted instead of one. As it can be seen in the chart, our proposal (EST) has the least security flaws compared to other tokens.

7. Conclusion

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 examine ERC20 security vulnerabilities and thoroughly discuss the technical details, the circumstances of the incidents together with their impacts and mitigations. We also integrate best practices to improve efficiency and productivity of the token. Eventually, we propose a secure ERC20 code that is not vulnerable to any of the attacks. Using auditing tools and comparing with the top ten ERC20 tokens shows the security of the proposal. It can be used as template to deploy new ERC20 tokens, migrate current vulnerable deployments or develop tools to automate auditing of ERC20 tokens.

References

[1] Ethereum. Ethereum project repository. https://github.com/ethereum, May 2014.

[2] Gavin Wood. Ethereum: a secure decentralised generalised transaction ledger. http://gavwood.com/paper.pdf, Mar 2016.

[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, Oct 2016.

[4] 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/, Sep 2016.

[5] PeckShield. Alert: New batchoverflow bug in multiple erc20 smart contracts (cve-2018-10299). https://blog.peckshield.com/2018/04/22/batchOverflow/, Apr 2018.

[6] Santiago Palladino. The parity wallet hack explained. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/, July 2017.

[7] Kai Sedgwick. Myetherwallet servers are hijacked in dns attack. https://news.bitcoin.com/myetherwallet-servers-are-hijacked-in-dns-attack/, Apr 2018.

[8] Haseeb Qureshi. A hacker stole $31m of ether — how it happened, and what it means for ethereum. https://www.freecodecamp.org/news/a-hacker-stole-31m-of-ether-how-it-happened-and-what-it-means-for-ethereum-9e5dc29e33ce/, Jul 2017.

[9] Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gun Sirer. An in-depth look at the parity multisig bug. https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/, Jul 2017.

[10] Age Manning. Comprehensive list of known attack vectors and common anti-patterns. https://github.com/sigp/solidity-security-blog, Nov 2019.

[11] Laurent Njilla Huashan Chen, Marcus Pendleton and Shouhuai Xu. A survey on ethereum systems security: Vulnerabilities, attacks and defenses. https://arxiv.org/pdf/1908.04507.pdf, Aug 2019.

[12] Solidity documentation. Security considerations. https://solidity.readthedocs.io/en/latest/security-considerations.html, Jan 2020.

[13] ConsenSys Diligence. Ethereum smart contract security best practices. https://consensys.github.io/smart-contract-best-practices/, Jan 2020.

[14] Guy Lando. Guy lando's knowledge list. https://github.com/guylando/KnowledgeLists/blob/master/EthereumSmartContracts.md, May 2019.

[15] Reza Rahimian. Overflow attack in ethereum smart contracts. https://blockchain-projects.readthedocs.io/overflow.html, Dec 2018.

[16] Christoph Jentzsch. The history of the dao and lessons learned. https://blog.slock.it/the-history-of-the-dao-and-lessons-learned-d06740f8cfa5, Aug 2016.

[17] Michael Rodler, Wenting Li, Ghassan Karame, and Lucas Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. https://arxiv.org/pdf/1812.05934.pdf, Dec 2018.

[18] wikipedia. Mutual exclusion. https://en.wikipedia.org/wiki/Mutual_exclusion, Jan 2019.

[19] Ethereum documentation. Re-entrancy. https://solidity.readthedocs.io/en/latest/security-considerations.html#re-entrancy, Jan 2020.

[20] Kirill Bulgakov. Three methods to send ether by means of solidity. https://medium.com/daox/three-methods-to-transfer-funds-in-ethereum-by-means-of-solidity-5719944ed6e9, Feb 2018.

[21] Afri Schoedon Alex Beregszaszi. Hardfork meta: Istanbul. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1679.md, Dec 2019.

[22] Steve Marx. Stop using solidity's transfer() now. https://diligence.consensys.net/blog/2019/09/stop-using-soliditys-transfer-now/, Dec 2019.

[23] Hubert Ritzdorf. Ethereum istanbul hardfork, the security perspective. https://docs.google.com/presentation/d/1IiRYSjwle02zQUmWId06Bss8GrxGyw6nQAiZdCRFEPk/, Oct 2019.

[24] Daniel Szego. Solidity security patterns - forcing ether to a contract. http://danielszego.blogspot.com/2018/03/solidity-security-patterns-forcing.html, Mar 2018.

[25] Mikhail Vladimirov. Attack vector on erc20 api (approve/transferfrom methods) and suggested improvements. https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729, Nov 2016.

[26] Tom Hale. Resolution on the eip20 api approve / transferfrom multiple withdrawal attack #738. https://github.com/ethereum/EIPs/issues/738, Oct 2017.

[27] Chris Coverdale. Solidity: Transaction-ordering attacks. https://medium.com/coinmonks/solidity-transaction-ordering-attacks-1193a014884e, Mar 2018.

[28] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. Sok: Transparent dishonesty: front-running attacks on blockchain. *International Conference on Financial Cryptography and Data Security*, 1:380, 2019.

[29] Reza Rahimian, Shayan Eskandari, and Jeremy Clark. Resolving the multiple withdrawal attack on erc20 tokens. https://arxiv.org/abs/1907.00903, Jul 2019.

[30] Ethereum. Solidity — solidity documentation. https://solidity.readthedocs.io/en/latest/contracts.html?highlight=library#libraries, Jan 2020.

[31] Sarvesh Jain. All you should know about libraries in solidity. https://medium.com/coinmonks/all-you-should-know-about-libraries-in-solidity-dd8bc953eae7, Sep 2018.

[32] Parity Technologies. Security alert. https://www.parity.io/security-alert-2/, Nov 2018.

[33] ConsenSys Diligence. Token implementation best practice. https://consensys.github.io/smart-contract-best-practices/tokens/, Mar 2020.

[34] Ethereum. Solidity — solidity documentation. https://solidity.readthedocs.io/en/latest/, Jan 2020.

[35] Facu Spagnuolo. Ethereum in depth, part 2. https://blog.openzeppelin.com/ethereum-in-depth-part-2-6339cf6bddb9/, Jul 2018.

[36] Jack Tanner. Summary of ethereum upgradeable smart contract r&d — part 1–2018. https://blog.indorse.io/ethereum-upgradeable-smart-contract-strategies-456350d0557c, Mar 2018.

[37] OpenZeppelin. Proxy patterns. https://blog.openzeppelin.com/proxy-patterns/, Apr 2018.

[38] ConsenSys Diligence. Secure development recommendations. https://consensys.github.io/smart-contract-best-practices/recommendations/#favor-pull-over-push-for-external-calls, Mar 2020.

[39] Monika di Angelo and Gernot Salzer. A survey of tools for analyzing ethereum smart contracts. https://publik.tuwien.ac.at/files/publik_278277.pdf, Aug 2019.

[40] Petar Tsankov. Securify v2.0. https://github.com/eth-sri/securify2, Jan 2020.

[41] SmartDec. Smartcheck knowledge-base. https://tool.smartdec.net/knowledge, Sep 2018.

[42] ConsenSys. Mythx swc coverage. https://mythx.io/swc-coverage/, Nov 2019.

[43] GuardStrike. Contractguard knowledge-base. https://contract.guardstrike.com/#/knowledge, Mar 2020.

[44] Feist Josselin. Slither – detector documentation. https://github.com/crytic/slither/wiki/Detector-Documentation#name-reused, Mar 2020.

[45] Gary Simon. Solidity modifier tutorial - control functions with modifiers. https://coursetro.com/posts/code/101/Solidity-Modifier-Tutorial---Control-Functions-with-Modifiers, Oct 2017.

[46] Nicolas Venturo and Francisco Giordano. Reentrancy guard. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/ReentrancyGuard.sol, Oct 2017.

[47] Vitalik Buterin Fabian Vogelsteller. Erc-20 token standard. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md, Nov 2015.

[48] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, and Arthur Gervais. Securify: Practical security analysis of smart contracts. https://arxiv.org/pdf/1806.01143.pdf, Aug 2018.

[49] Feist Josselin. Slither – a solidity static analysis framework. https://blog.trailofbits.com/2018/10/19/slither-a-solidity-static-analysis-framework/, Oct 2018.

[50] EtherScan. Token tracker. https://etherscan.io/tokens?sortcmd=remove&sort=marketcap&order=desc, Apr 2020.