A deep dive on ERC-20 contract vulnerabilities

No Author Given

No Institute Given

Abstract. ERC-20 is the most prominent Ethereum standard for transferable tokens. Tokens implementing the ERC-20 interface can interoperate with a large number of already deployed internet-based services and Ethereum-based smart contracts. In recent years, security vulnerabilities in ERC-20 implementations have been uncovered. We (i) systemize these across 7 auditing tools into a set of 82 distinct vulnerabilities and best practices, and (ii) use our experience to provide a new secure implementation of the ERC-20 interface, TokenHook, that is freely[1] available and diversifies known implementation from OpenZeppelin and ConsenSys. We also (iii) analyze the top ten ERC-20 tokens by market capitalization for comparison.

1 Introduction

The Ethereum blockchain project was launched in 2014 by announcing Ether (ETH) as its protocol-level cryptocurrency [17,71]. Ethereum allows users to build and deploy decentralized applications (DApps), or smart contracts, that can accept and use ETH. Many DApps also issue their own custom tokens with a variety of intents, including tokens as: financial products, in-house currencies, voting rights for DApp governance, valuable assets, crypto-collectibles, *etc.* To encourage interoperability with other DApps and web apps (exchanges, wallets, *etc.*), the Ethereum community accepted a popular token standard (for non-fungible tokens) called ERC-20 [22]. While numerous ERC-20 extensions or replacements have been proposed, ERC-20 remains prominent. Of the 2.5M[50] smart contracts on the Ethereum network, 260K are tokens [63]. 98% of these tokens are ERC-20 [20], demonstrating their widespread acceptance by the industry and Ethereum community.

The development of smart contracts has been proven to be error-prone, and as a result, smart contracts are often riddled with security vulnerabilities. An early study in 2016 found that 45% of smart contracts at that time had vulnerabilities[35]. ERC-20 token are subset of smart contracts and security is particularly important given that many tokens have considerable market capitalization (*e.g.,* USDT, LINK, CRO, LEO, DAI, *etc.*). As tokens can be held by commercial firms, in addition to individuals, and firms need audited financial statements in certain circumstances, the correctness of the contract issuing the tokens is now in the purview of professional auditors. One tool we examine, EY Smart Contract and Token Review[21], is from a 'big-four' auditing firm.

[1] Implementation on Etherscan with source code and deployed on Mainnet and Rinkeby: `https://bit.ly/35FMbAf`, `https://bit.ly/33wDENx`

Contributions. Similar to any new technology, Ethereum has undergone numerous security attacks that have collectively caused more than US$100M in financial losses[24,43,41,55,46,3]. Although research has been done on smart contract vulnerabilities in the past[29], our focus is on ERC-20 tokens only. Some vulnerabilities (such as multiple withdrawals) will be more serious in token contracts. This motivates us to (i) comprehensively study all known vulnerabilities in ERC-20 token contracts, systematizing them into a set of 82 distinct vulnerabilities and best practices, and review the completeness and precision of auditing tools in detecting these vulnerabilities to establish the reliability of an audit based on these tools. We (ii) use this research to provide a new secure implementation of the ERC-20 interface, TokenHook, that is freely available and open source. Compared to other implementations from OpenZeppelin[39] and ConsenSys[7], it is more secure and fully compatible with ERC-20 specifications (see section3). Finally, (iii) we examine the practicality of our work in the context of the top ten ERC-20 tokens by market capitalization.

2 TokenHook

TokenHook, our ERC20-compliant implementation written in Solidity. It covers all functionalities supported by existing ERC-20 contracts while mitigates more security vulnerabilities. TokenHook is open source and available on Etherscan, where it has been tested with MetaMask and deployed on Mainnet[2]. It can be customized by developers, who can refer to each mitigation technique separately and address a specific attack. Required comments have been also added to clarify usage of each part. Standard functionalities of the token (*i.e.,* `approve()`, `transfer()`, *etc.*) have been unit tested. A demonstration of token interactions and event triggering can also be seen on Etherscan[3].

Among the layers of the Ethereum blockchain, ERC-20 tokens fall under the *Contract layer* in which DApps are executed. The presence of security vulnerability in supplementary layers affect the entire Ethereum blockchain, not necessarily ERC-20 tokens. Therefore, vulnerabilities in other layers are assumed to be out of the scope (*e.g., Indistinguishable chains* at the data layer, the *51% attack* at the consensus layer, *Unlimited nodes creation* at network layer, and *Web3.js Arbitrary File Write* at application layer). Moreover, we exclude vulnerabilities identified in now 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.

[2] Etherscan: `https://bit.ly/35FMbAf`
[3] Etherscan: `https://bit.ly/33xHfL2`, `https://bit.ly/35TimMW`

2.1 Security features

We examine general attack vectors and cross-check their applicability to ERC-20 tokens (See section A). We then consider mitigation techniques in our implementation as summarized below:

1. Securing `transferFrom()` function by tracking transferred tokens. It mitigates *multiple withdrawal* attack wherein an attacker may transfer more token than the approved amount by the token holder. (*cf.* Section A.1)
2. Using `SafeMath` library in all arithmetic operations to catch over/under flows. (*cf.* Section A.2)
3. Implementing `noReentrancy` modifier for external functions. It enforces Mutex and mitigate *same-function re-entrancy* and *cross-function re-entrancy* attacks. (*cf.* Section A.3)
4. Checking the returned value of `call.value()` to revert failed fund transfers in `sell()` and `withdraw()` functions. It mitigates *Unchecked return values* attack while making the token contract compatible with EIP-1884[28]. (*cf.* Section A.4)
5. Mitigating *Frozen Ether* issue by defining `withdraw()` function. It allows the owner to transfer ETH out of the token contract. Otherwise, sent ETH to the token will be stuck forever (*cf.* Section A.5)
6. Applying `onlyOwner` modifier to `withdraw()` function. It Mitigates *Unprotected Ether Withdrawal* issue by enforcing authentication before transferring any funds out of the contract. (*cf.* Section A.6)
7. Adding `Library` code to the ERC-20 contract's code (*i.e.,* Using embedded libraries). It mitigates *State variable manipulation* attack and avoids updating internal variables by external contracts. Also, calling functions in embedded libraries requires less gas compared to invoking them via external calls (*cf.* Section A.7)
8. Avoiding default `Public` visibility by explicitly defining visibility of each function. Most of the functions are declared as `External` (*e.g.,* `Approve()`, `Transfer()`, *etc.*) per specifications of ERC-20 standard. (*cf.* Section A.8)

2.2 Complementary features

In addition to reviewing known vulnerabilities, we also took into account a number of best practices (See section B). They improve the performance of TokenHook and complement its features:

1. Implementing all functions to make it fully compatible with the ERC-20 standard. Therefore, there would not be any failed call for other DApps (*i.e.,* crypto-wallets, crypto-exchanges, web services, *etc.*) which are expecting them (Section B.1)
2. Enforcing `External` visibility for interactive functions (*e.g.,* `approve()` and `transfer()`, *etc.*) to improve performance. Functions declared as `External` can read arguments directly from non-persistent `calldata` area, instead of allocating persistent memory by EVM. (Section B.2)

3. Ability to stop the token in case of new security threats or legal requirements (*e.g.,* Liberty Reserve [70] or TON cryptocurrency[15]). To freeze all functionality of TokenHook, the owner (or multiple parties) can call `pause()` function. It then sets a lock variable and methods are marked with `notPaused` modifier, throw exceptions until functionality is restored using `unpause()`. (Section B.3)
4. Defining nine extra events: `Buy`, `Sell`, `Received`, `Withdrawal`, `Pause`, `Change`, `ChangeOwner`, `Mint` and `Burn`. `Change` event logs any state variable updates that can be used to watch for token inconsistent behavior (*e.g.,* via TokenScope[5]) and react accordingly. (Section B.4)
5. Implementing `sell()` and `buy()` for exchanging tokens and ETH. `sell()` allows token holders to exchange tokens for ETH and `buy()` accepts ETH by adjusting buyer's token balance. This can be considered as a financial incentive in which it is possible to buy and sell tokens at a fixed price by the token contract (*e.g.,* Launching an Initial Coin Offering (ICO), Prediction market sell). Otherwise, buyers will have to wait for the token to be listed on crypto-exchanges (if it ever happens) or look for a buyer/seller themselves. In addition, it reduces the cost of token exchange by eliminating crypto-exchange's fees.
6. Making TokenHook as non-upgradable token for auditing purpose. Initial token audit might show it as secure while the upgraded versions contains new vulnerabilities that did not exist at the time of initial audit.
7. Considering other best practices such as not using batch processing in `sell()` function to avoid *DoS with Unexpected revert* issue, not using miner controlled variable in conditional statements and not using `SELFDESTRUCT`.

2.3 Increased diversity

The authors of the ERC-20 standard reference two sample implementations from OpenZeppelin[39] and ConsenSys[7]. ConsenSys implementation is deprecated (according to the GitHub page) and OpenZeppelin template is the most referenced implementation by the community[52,72,45]. Having different ERC-20 implementation provides more variety in implementations and minimizes the possibility of bugs that existed in the past. Between 17 March 2017 and 13 July 2017, OpenZeppelin implemented the wrong interface in their framework that affected 130 tokens[9]. Diversity in ERC-20 implementation can reduce the impact of such errors and increase the robustness of ERC-20 tokens. Additionally, TokenHook has the following advantages:

- Unlike OpenZeppelin which introduces two new functions to mitigate *front-running attack* (*i.e.,* `increaseAllowance()` and `decreaseAllowance()`), TokenHook secures standard ERC-20 function (*i.e.,* `transferFrom()`). DApps can interact with standard `approve()` and `transferFrom()` methods without adapting their code to these new functions. TokenHook is therefore fully compliant with the ERC-20 specification and can interact with already developed and legacy DApps.

- TokenHook mitigates *Frozen Ether* issue by introducing `withdraw()` function while sent ETH to OpenZeppelin contract are unrecoverable.
- *Fail-Safe Mode* is a built-in feature of TokenHook while OpenZeppelin requires incorporation of `Pausable.sol` contract.
- OpenZeppelin requires other optimizations such as *Locking the pragma*, Emitting *Change* event (*cf.* TokenScope[5]) when updating state variables (*e.g.,* `_decimals=decimals_` in `_setupDecimals()`), *Initializing totalSupply in constructor*, using `External` visibility instead of Public to increase readability (*i.e.,* no internal call) and consume less gas, Avoiding similar variable names (*e.g.,* `_name=name_` in `constructor()`), Using *mixedCase* format when declaring variable and functions (*e.g.,* `_symbol`, `_decimals`), *etc.*
- Using reusable codes has made the OpenZeppelin code complex and challenging for security tools. Developers need to manually check the code for vulnerabilities instead of using vulnerability assessment tools. Additionally, most of the security tools are not able to import libraries/interfaces from external files (*e.g.,* SafeMath.sol, IERC20.sol). TokenHook has a flat layout and all codes are in one file. It is easier for developers to understand and upload it to audit tools for vulnerability assessment.

3 Code audit

We used a variety of code audit tools on TokenHook to validate the code and also to illuminate the completeness and error-rate of such tools on one specific use-case (similar work studies in less depth a variety of use-cases[2]). We also did not adapt older tools that support significantly lower versions of the Solidity compiler (*e.g.,* Oyente). Moreover, if the version of tool is not mentioned, online versions are used which do not have a specific version:

1. EY Review Tool by Ernst & Young Global Limited[21].
2. SmartCheck by SmartDec[59].
3. Securify v2.0 by ChainSecurity[65].
4. ContractGuard by GuardStrike[26].
5. MythX by ConsenSys[6].
6. Slither Analyzer v0.6.12 by Crytic[33].
7. Odin by Sooho[60].

3.1 Analysis of audit results

A total of 82 audits have been conducted by these auditing tools. Audits include best practices and security vulnerabilities. The results are summarized in Tables1–3. To compile the list, we referenced the knowledge-base of each tool[64,59,6,26,33], understood each threat, manually mapped the audit to the corresponding SWC registry[58], and manually determined when different tools were testing for the same vulnerability or best practice (which was not always clear from the tools' own descriptions). Space will not permit us to discuss each

ID	SWC	**Vulnerability or best practice** Mitigation or recommendation	EY Token Review	Smart Check	Securify	MythX (Mythril)	Contract Guard	Slither	Odin
			Security tools						
1	100	**Function default visibility** Specifying function visibility, external, public, internal or private		✓		✓	✓		✓
2	101	**Integer Overflow and Underflow** Utilizing the SafeMath library to mitigate over/under value assignments	⊕	!		✓	✓		✓
3	102	**Outdated Compiler Version** Using proper Solidity version to protect against compiler attacks	✓	✓	✓	✓	✓	✓	×
4	103	**Floating Pragma** Locking the pragma to avoid deployments using outdated compiler version		✓	✓	✓		✓	✓
5	104	**Unchecked Call Return Value** Checking call() return value to prevent unexpected behavior in DApps	⊕		✓	✓	✓	⊕	✓
6	105	**Unprotected Ether Withdrawal** Authorizing only trusted parties to trigger ETH withdrawals		!		✓		✓	✓
7	106	**Unprotected SELFDESTRUCT Instruction** Removing self-destruct functionality or approving it by multiple parties			✓	✓		✓	✓
8	107	**Re-entrancy** Using CEI and Mutex to mitigate self-function and cross-function attacks		✓	⊕	⊕	⊕	✓	✓
9	108	**State variable default visibility** Specifying visibility of all variables, public, private or internal	✓	✓	✓	✓	✓		✓
10	109	**Uninitialized Storage Pointer** Initializing variables upon declaration to prevent unexpected storage access	✓	✓	✓	✓	✓	✓	✓
11	110	**Assert Violation** Using require() statement to validate inputs, checking efficiency of the code		✓		✓			✓
12	111	**Use of Deprecated Solidity Functions** Using new alternatives functions such as keccak256() instead of sha3()		✓		✓	✓	✓	✓
13	112	**Delegatecall to untrusted callee** Calling into trusted contracts to avoid storage access by malicious contracts		⊕	⊕	✓	✓	✓	✓
14	113	**DoS with Failed Call** Avoid multiple external calls where one error may fail other transactions	✓	✓		✓	✓		✓
15	114	**Transaction Order Dependence** Preventing race conditions by securing approve() or transferFrom()	⊕		✓	✓			✓
16	115	**Authorization through tx.origin** Using msg.sender to authorize transaction initiator instead of originator	✓	✓	✓	✓	✓	✓	✓
17	116	**Block values as a proxy for time** Not using block.timestamp or block.number to perform functionalities	✓	✓	✓	✓	✓		✓
18	117	**Signature Malleability** Not using signed message hash to avoid signatures alteration				✓			✓
19	118	**Incorrect Constructor Name** Using constructor keyword which does not match with contract name		✓		✓			✓
20	119	**Shadowing State Variables** Removing any variable ambiguities when inheriting other contracts			✓	✓	✓	✓	✓
21	120	**Weak Sources of Randomness from Chain Attributes** Using oracles as source of randomness instead of block.timestamp	✓	✓		✓	✓		✓
22	121	**Missing Protection against Signature Replay Attacks** Storing every message hash to perform signature verification				✓			✓
23	122	**Lack of Proper Signature Verification** Using alternate verification schemes if allowing off-chain signing				✓			✓
24	123	**Requirement Violation** Checking the code for allowing only valid external inputs		✓	✓	✓			✓
25	124	**Write to Arbitrary Storage Location** Controlling write to storage to prevent storage corruption by attackers		✓	✓	✓			✓
26	125	**Incorrect Inheritance Order** Inheriting from more general to specific when there are identical functions				✓			✓
27	126	**Insufficient Gas Griefing** Allowing trusted forwarders to relay transactions		✓					✓

Table 1. Auditing results of 7 smart contract analysis tools on TokenHook. ✓=Passed audit, ⊕=False positive, ×=Failed audit, Empty=Not supported audit by the tool, !=Informational, ◯=Tool specific audit (No SWC registry), BP=Best practice

.

ID	SWC	Vulnerability or best practice Mitigation or recommendation	EY Token Review	Smart Check	Securify	MythX (Mythril)	Contract Guard	Slither	Odin
			Security tools						
28	127	**Arbitrary Jump with Function Type Variable** Minimizing use of assembly in the code		✓	✓	✓		✓	✓
29	128	**DoS With Block Gas Limit** Avoiding loops across the code that may consume considerable resources	✓	✓	✓	✓	✓	✓	✓
30	129	**Typographical Error** Using SafeMath library or performing checks on any math operation				✓			✓
31	130	**Right-To-Left-Override control character (U+202E)** Avoiding U+202E character which forces RTL text rendering			✓	✓	✓	✓	✓
32	131	**Presence of unused variables** Removing all unused variables to decrease gas consumption		✓	✓		✓	✓	⊕
33	132	**Unexpected Ether balance** Avoiding Ether balance check in the code (*e.g.,* this.balance == 0.24 Ether)		✓	✓		✓	✓	✓
34	133	**Hash Collisions With Variable Length Arguments** Using abi.encode() instead of abi.encodePacked() to prevent hash collision							✓
35	134	**Message call with hardcoded gas amount** Using .call.value()("") which is compatible with EIP1884		⊕	⊕	✓	✓		✓
36	135	**Code With No Effects** Writing unit tests to ensure producing the intended effects by DApps		✓					✓
37	136	**Unencrypted Private Data On-Chain** Storing un-encrypted private data off-chain		!					✓
38	○	**Allowance decreases upon transfer** Decreasing allowance in transferFrom() method	✓						
39	○	**Allowance function returns an accurate value** Returning only value from the mapping instead of internal function logic	✓						
40	○	**It is possible to cancel an existing allowance** Possibility of setting allowance to 0 to revoke previous allowances	✓	✓					
41	○	**A transfer with an insufficient amount is reverted** Checking balances in transfer() method before updating balances	✓					✓	
42	○	**Upon sending funds, the sender's balance is updated** Updating balances in transfer() or transferFrom() methods	✓						
43	○	**The Transfer event correctly logged** Emitting Transfer event in transfer() or transferFrom() functions	✓						
44	○	**Transfer an amount that is greater than the allowance** Checking balances in transferFrom() method before updating balances	✓						
45	○	**Risk of short address attack is minimized** Using recent Solidity version to mitigate the attack	✓				✓		
46	○	**Function names are unique** No function overloading to avoid unexpected behavior	✓					✓	
47	○	**Using miner controlled variables** Avoiding block.number, block.timestamp, block.difficulty, now, etc	✓	✓	✓	✓	✓	✓	
48	○	**Use of return in constructor** Not using return in contract's constructor		✓					
49	○	**Throwing exceptions in transfer() and transferFrom()** Returning true after successful execution or raising exception in failures		✓				✓	
50	○	**State variables that could be declared constant** Adding constant attribute to variables like name, symbol, decimals, etc						✓	
51	○	**Tautology or contradiction** Fixing comparison in the code that are always true or false						✓	
52	○	**Divide before multiply** Ordering multiplication prior division to avoid integer truncation						✓	
53	○	**Unchecked Send** Ensuring that the return value of send() is always checked						✓	
54	BP	**Too many digits** Using scientific notation to make the code readable and simpler to debug						✓	

Table 2. Continuation of Table1.

ID	SWC	**Vulnerability or best practice** Mitigation or recommendation	EY Token Review	Smart Check	Securify	MythX (Mythril)	Contract Guard	Slither	Odin
			Security tools						
55	BP	**The decreaseAllowance definition follows the standard** Defining decreaseAllowance input and output variables as standard	✓						
56	BP	**The increaseAllowance definition follows the standard** Defining increaseAllowance input and output variables as standard	✓						
57	BP	**Minimize attack surface** Checking whether all the external functions are necessary or not	✓	✓	✓				
58	BP	**Transfer to the burn address is reverted** Reverting transfer to 0x0 due to risk of total supply reduction	✓						
59	BP	**Source code is decentralized** Not using hard-coded addresses in the code	✓	✓					
60	BP	**Funds can be held only by user-controlled wallets** Transferring tokens to users to avoid creating a secondary market	!						
61	BP	**Code logic is simple to understand** Avoiding code nesting which makes the code less intuitive	✓	✓					
62	BP	**All functions are documented** Using NatSpec format to explain expected behavior of functions	✓						
63	BP	**The Approval event is correctly logged** Emitting Approval event in the approve() method	✓						
64	BP	**Acceptable gas cost of the approve() function** Checking for maximum 50000 gas cost when executing the approve()	!						
65	BP	**Acceptable gas cost of the transfer() function** Checking for maximum 60000 gas cost when executing the transfer()	!						
66	BP	**Emitting event when state changes** Emitting Change event when changing state variable values	✓						
67	BP	**Use of unindexed arguments** Using indexed arguments to facilitate external tools log searching		✓			✓	✓	
68	BP	**ERC-20 compliance** Implementing all 6 functions and 2 events as specified in EIP-20	✓	✓	✓		✓	✓	
69	BP	**Conformance to naming conventions** Following the Solidity naming convention to avoid confusion						✓	
70	BP	**Token decimal** Declaring token decimal for external apps when displaying balances	✓						
71	BP	**Locked money (Freezing ETH)** Implementing withdraw/reject functions to avoid ETH lost		✓			✓	✓	
72	BP	**Malicious libraries** Not using modifiable third-party libraries		✓					
73	BP	**Payable fallback function** Adding either fallback() or receive() function to receive ETH		✓			✓		
74	BP	**Prefer external to public visibility level** Improving the performance by replacing public with external		✓				✓	
75	BP	**Token name** Adding a token name variable for external apps	✓						
76	BP	**Error information in revert condition** Adding error description in require()/revert() to clarify the reason					✓		
77	BP	**Complex Fallback** Logging operations in the fallback() to avoid complex operations					✓		
78	BP	**Function Order** Following fallback, external, public, internal and private order					✓		
79	BP	**Visibility Modifier Order** Specifying visibility first and before modifiers in functions						✓	
80	BP	**Non-initialized return value** Not specifying return for functions without output		✓			✓[6]		
81	BP	**Token symbol** Adding token symbol variable for usage of external apps	✓						
82	BP	**Allowance spending is possible** Ability of token transfer by transferFrom() to transfer tokens on behalf of another usercalc	✓						
		99.5% success rate in performed audits by considering 'False Positives' and 'Informational' checks as 'Passed' (More details in section3)	100%	100%	100%	100%	100%	100%	97%

Table 3. Continuation of Table2.

one at the same level of detail as the ones we highlight in sections A and B, however we will include a simple statement describing the issue and the mitigation.

Since each tool employs different methodology to analyze smart contracts (*e.g.,* comparing with violation patterns, applying a set of rules, using static analysis, *etc.*), there are false positives to manually check. Some false positive (*e.g., noReentrancy* modifier in *MythX* and *ContractGuard*) are not due to old/unmaintained rules. There is a need to optimize the tools to consider `Modifiers` in code analysis. The following are some examples of false positives (which we do not count in calculating our success rate):

1. *MythX* detects *Re-entrancy attack* in the *noReentrancy* modifier. In Solidity, modifiers are not like functions. They are used to add features or apply some restriction on functions[57]. Using modifiers is a known technique to implement Mutex and mitigate the attack[66]. This is a false positive and note that other tools have not identified the attack in modifiers.
2. *ContractGuard* flags *Re-entrancy attack* in `transfer()` function while both CEI and Mutex are implemented.
3. *Slither* detects two *low level call* vulnerabilities[32]. This is due to use of `call.value()` that is recommend way of transferring ETH after *Istanbul* hard-fork (EIP-1884). Therefore, adapting analyzers to new standards can improve accuracy of the security checks.
4. *SmartCheck* recommends not using `SafeMath` and check explicitly where overflows might be occurred. We consider this failed audit as false possible whereas utilizing `SafeMath` is a known technique to mitigate over/under flows. It also flags *using a private modifier* as a vulnerability by mentioning, "miners have access to all contracts' data and developers must account for the lack of privacy in Ethereum". However private visibility in Solidity concerns object oriented inheritance not confidentiality. For actual confidentiality, the best practice is to encrypt private data or store them off-chain. The tool also warns against `approve()` in ERC-20 due to *front-running attacks*. Despite EIP-1884, it still recommends using of `transfer()` method with stipend of 2300 gas. There are other false positives such as SWC-105 and SWC-112 that are passed by other tools.
5. *Securify* detects the *Re-entrancy* attack due to unrestricted writes in the `noReentrancy` modifier[65]. Modifiers are the recommended approach and are not accessible by users. It also flags *Delegatecall to Untrusted Callee (SWC-112)* while there is no usage of `delegatecall()` in the code. It might be due to use of `SafeMath` library which is an embedded library. 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. Similar to *SmartCheck*, it still recommends to use the `transfer()` method instead of `call.value()`.
6. *EY token review* considers `decreaseAllowance` and `increaseAllowance` as standard ERC-20 functions and if not implemented, recognizes the code as vulnerable to a *front-running*. These two functions are not defined in the ERC-20 standard[22] and considered only by this tool as mandatory func-

tions. There are other methods to prevent the attack while adhering ERC-20 specifications (see Rahimian *et al.* for a full paper on this attack and the basis of the mitigation in TokenHook [48]). The tool also falsely detects the *Overflow*, mitigated through `SafeMath`. Another identified issue is *Funds can be held only by user-controlled wallets.* The tool warns against any token transfer to Ethereum addresses that belong to smart contracts. However, interacting with ERC-20 token by other smart contracts was one of the main motivations of the standard. It also 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 and therefore consider them as informational.

7. *Odin* raises *Outdated compiler version* issue due to locking solidity version to 0.5.11. We have used this version due to its compatibility with other auditing tools. Furthermore, other tools have not identified such an issue and we therefore consider it as informational.

3.2 Comparing audits

After manually overriding the false positives, the average percentage of passed checks for TokenHook reaches to 99.5%. To pass the one missing check and reach a 100% success rate across all tools, we prepared the same code in Solidity version 0.8.0, however it cannot be audited anymore with most of the tools.

We repeated the same auditing process on the top ten tokens based on their market cap[20]. The result of all these evaluation have been summarized in Table4 by considering false positives as failed audits. This provide the same evaluation conditions across all tokens. Since each tool uses different analysis methods, number of occurrences are considered for comparisons. For example, MythX detects two *re-entrancy* in TokenHook; therefore, two occurrences are counted instead of one.

As it can be seen in Table 4, TokenHook has the least number of security flaws (occurrences) compared to other tokens. We stress that detected security issues for TokenHook are all false positives. Considering number of vulnerability might not be a perfect measurement, but it provides at least an approximation of the token security factor compared to other tokens. It should also be noted that the existence of vulnerability does not mean that it can be exploited [44]. It could have been the subject of a separate study that we leave it for future works

4 Conclusion

98% of tokens on Ethereum today implement ERC-20. While attention has been paid to the security of Ethereum DApps, threats to tokens can be specific to ERC-20 functionality. Further, there is no vulnerability reference site (*cf.* the SWC Registry) specifically for ERC-20 tokens. In this paper, we provide a detailed study of ERC-20 security, collecting and deduplicating 82 vulnerabilities

ERC-20 Token	Auditing Tool							Total issues
	EY Token Review	Smart Check	Securify	MythX (Mythril)	Contract Guard	Slither	Odin	
TokenHook	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 4. Security flaws detected by seven auditing tools in TokenHook (the proposal) compared to top 10 ERC-20 tokens by market capitalization in May 2020. TokenHook has the lowest reported security issues (occurrences).

and best practices, examining the ability of seven audit tools, and auditing 10 ERC-20 deployments. Most importantly, we provide a concrete implementation of ERC-20 called TokenHook. It is designed to be secure against known vulnerabilities and diversifies other implementations. We test it at Solidity version 0.5.11 (due to the limitation of the audit tools) and also provide it at 0.8.0. TokenHook can be used as template to deploy new ERC-20 tokens, migrate current vulnerable deployments, and to benchmark the precision of Ethereum audit tools.

References

1. Alex Beregszaszi, A.S.: Hardfork meta: Istanbul. `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1679.md` (Dec 2019)
2. di Angelo, M., Salzer, G.: A survey of tools for analyzing ethereum smart contracts. `https://publik.tuwien.ac.at/files/publik_278277.pdf` (Aug 2019)
3. Breidenbach, L., Daian, P., Juels, A., Gun Sirer, E.: An in-depth look at the parity multisig bug. `https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/` (Jul 2017)
4. Bulgakov, K.: 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)
5. Chen, T., Zhang, Z., Li, Z.: Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum. `http://www4.comp.polyu.edu.hk/~csxluo/TokenScope.pdf` (Nov 2019)
6. ConsenSys: Mythx swc coverage. `https://mythx.io/swc-coverage/` (Nov 2019)
7. Consensys: Tokens. `https://github.com/ConsenSys/Tokens` (Jun 2020)
8. Coverdale, C.: Transaction-ordering attacks. `https://medium.com/coinmonks/solidity-transaction-ordering-attacks-1193a014884e` (Mar 2018)
9. Cremer, L.: Missing return value bug — at least 130 tokens affected. `https://medium.com/coinmonks/missing-return-value-bug-at-least-130-tokens-affected-d67bf08521ca` (Jun 2018)

10. Deployer, B.: Beautychain (bec). https://etherscan.io/address/0xc5d105e63711398af9bbff092d4b6769c82f793d (Jun 2020)
11. Diligence, C.: Ethereum smart contract security best practices. https://consensys.github.io/smart-contract-best-practices/ (Jan 2020)
12. Diligence, C.: Token implementation best practice. https://consensys.github.io/smart-contract-best-practices/tokens/ (Mar 2020)
13. documentation, E.: Re-entrancy. https://solidity.readthedocs.io/en/latest/security-considerations.html#re-entrancy (Jan 2020)
14. documentation, S.: Security considerations. https://solidity.readthedocs.io/en/latest/security-considerations.html (Jan 2020)
15. Durov, P.: What was ton and why it is over. https://telegra.ph/What-Was-TON-And-Why-It-Is-Over-05-12 (May 2020)
16. Eskandari, S., Moosavi, S., Clark, J.: Sok: Transparent dishonesty: front-running attacks on blockchain. International Conference on Financial Cryptography and Data Security **1**, 380 (2019)
17. Ethereum: Project repository. https://github.com/ethereum (May 2014)
18. Ethereum: Solidity — solidity documentation. https://solidity.readthedocs.io/en/latest/contracts.html?highlight=library#libraries (Jan 2020)
19. Ethereum: Solidity — solidity documentation. https://solidity.readthedocs.io/en/latest/ (Jan 2020)
20. EtherScan: Token tracker. https://etherscan.io/tokens?sortcmd=remove&sort=marketcap&order=desc (Apr 2020)
21. EY: Token review. https://review-tool.blockchain.ey.com (Sep 2019)
22. Fabian Vogelsteller, V.B.: Erc-20 token standard. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md (Nov 2015)
23. Ferreira Torres, C., Schutte, J., State, R.: Osiris: Hunting for integer bugs in ethereum smart contracts. https://dl.acm.org/doi/10.1145/3274694.3274737 (Dec 2018)
24. Finley, K.: 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)
25. GmbH, G.F.: Golem network token. https://etherscan.io/address/0xa74476443119A942dE498590Fe1f2454d7D4aC0d#code (Nov 2016)
26. GuardStrike: Contractguard knowledge-base. https://contract.guardstrike.com/#/knowledge (Mar 2020)
27. Hale, T.: Resolution on the eip20 api approve / transferfrom multiple withdrawal attack #738. https://github.com/ethereum/EIPs/issues/738 (Oct 2017)
28. Holst Swende, M.: Repricing for trie-size-dependent opcodes. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1884.md (Mar 2019)
29. Huashan Chen, Marcus Pendleton, L.N., Xu, S.: A survey on ethereum systems security: Vulnerabilities, attacks and defenses. https://arxiv.org/pdf/1908.04507.pdf (Aug 2019)
30. Jain, S.: All you should know about libraries in solidity. https://medium.com/coinmonks/all-you-should-know-about-libraries-in-solidity-dd8bc953eae7 (Sep 2018)
31. Ji, R., He, N., Wu, L., Wang, H.: Deposafe: Demystifying the fake deposit vulnerability. https://arxiv.org/pdf/2006.06419.pdf (Jun 2020)
32. Josselin, F.: Slither – a solidity static analysis framework. https://blog.trailofbits.com/2018/10/19/slither-a-solidity-static-analysis-framework/ (Oct 2018)

33. Josselin, F.: Slither – detector documentation. https://github.com/crytic/slither/wiki/Detector-Documentation#name-reused (Mar 2020)
34. Lando, G.: Guy lando's knowledge list. https://github.com/guylando/KnowledgeLists/blob/master/EthereumSmartContracts.md (May 2019)
35. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: 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)
36. Manning, A.: Comprehensive list of known attack vectors and common anti-patterns. https://github.com/sigp/solidity-security-blog (Nov 2019)
37. Marx, S.: Stop using solidity's transfer() now. https://diligence.consensys.net/blog/2019/09/stop-using-soliditys-transfer-now/ (Dec 2019)
38. OpenZeppelin: Proxy patterns. https://blog.openzeppelin.com/proxy-patterns/ (Apr 2018)
39. OpenZeppelin: Contracts. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol (Jun 2020)
40. OpenZeppelin: Contracts. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol (Jun 2020)
41. Palladino, S.: The parity wallet hack explained. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/ (July 2017)
42. Parity: Security alert. https://www.parity.io/security-alert-2/ (Nov 2018)
43. PeckShield: Alert: New batchoverflow bug in multiple erc20 smart contracts. https://blog.peckshield.com/2018/04/22/batchOverflow/ (Apr 2018)
44. Perez, D., Livshits, B.: Smart contract vulnerabilities: Vulnerable does not imply exploited. https://www.usenix.org/system/files/sec21summer_perez.pdf (Aug 2018)
45. Quintal, J.: Building robust smart contracts with openzeppelin. https://www.trufflesuite.com/tutorials/robust-smart-contracts-with-openzeppelin (Aug 2017)
46. Qureshi, H.: A hacker stole $31m of ether — how it happened, and what it means. https://www.freecodecamp.org/news/a-hacker-stole-31m-of-ether-how-it-happened-and-what-it-means-for-ethereum-9e5dc29e33ce/ (Jul 2017)
47. Rahimian, R.: Overflow attack in ethereum smart contracts. https://blockchain-projects.readthedocs.io/overflow.html (Dec 2018)
48. Rahimian, R., Eskandari, S., Clark, J.: Resolving the multiple withdrawal attack on erc20 tokens. https://arxiv.org/abs/1907.00903 (Jul 2019)
49. Reinno: A new way to own and invest in real estate. https://reinno.io/tokenization.html (Jun 2020)
50. Reporting, A.: Generate meaningful knowledge from ethereum. https://reports.aleth.io/ (Jul 2020)
51. Ritzdorf, H.: Ethereum istanbul hardfork, the security perspective. https://docs.google.com/presentation/d/1IiRYSjwle02zQUmWId06Bss8GrxGyw6nQAiZdCRFEPk/ (Oct 2019)
52. Roan, A.: 7 openzeppelin contracts you should always use. https://medium.com/better-programming/7-openzeppelin-contracts-you-should-always-use-5ba2e7953cc4 (Apr 2020)
53. Rodler, M., Li, W., Karame, G., Davi, L.: Sereum: Protecting existing smart contracts against re-entrancy. https://arxiv.org/pdf/1812.05934.pdf (Dec 2018)
54. Rubixi: Rubixi contract. https://etherscan.io/address/0xe82719202e5965Cf5D9B6673B7503a3b92DE20be#code (Mar 2016)

55. Sedgwick, K.: Myetherwallet servers are hijacked. https://news.bitcoin.com/myetherwallet-servers-are-hijacked-in-dns-attack/ (Apr 2018)
56. Shirshova, N.: The benefits of "buy" and "sell". https://medium.com/orbise/the-benefits-of-buy-and-sell-token-functions-dcea536aaf7c (Dec 2018)
57. Simon, G.: Solidity modifier tutorial - control functions with modifiers. https://coursetro.com/posts/code/101/Solidity-Modifier-Tutorial---Control-Functions-with-Modifiers (Oct 2017)
58. SmartContractSecurity: Smart contract weakness classification and test cases. https://swcregistry.io/ (Jun 2020)
59. SmartDec: Knowledge-base. https://tool.smartdec.net/knowledge (Sep 2018)
60. Sooho: Verify a smart contract. https://odin.sooho.io/ (Mar 2020)
61. Spagnuolo, F.: Ethereum in depth, part 2. https://blog.openzeppelin.com/ethereum-in-depth-part-2-6339cf6bddb9/ (Jul 2018)
62. Tanner, J.: Summary of ethereum upgradeable smart contract r&d — part 1–2018. https://blog.indorse.io/ethereum-upgradeable-smart-contract-strategies-456350d0557c (Mar 2018)
63. Tracker, T.: Erc-20 tokens. https://etherscan.io/tokens (Jul 2020)
64. Tsankov, P.: Securify v2.0. https://github.com/eth-sri/securify2 (Jan 2020)
65. Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A.: Securify: Practical security analysis of smart contracts. https://arxiv.org/pdf/1806.01143.pdf (Aug 2018)
66. Venturo, N., Giordano, F.: Reentrancy guard. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/ReentrancyGuard.sol (Oct 2017)
67. Vessenes, P.: Monolithdao. https://github.com/MonolithDAO/token/blob/master/src/Token.sol (Apr 2017)
68. Vladimirov, M.: Attack vector on erc20 api (approve/transferfrom methods) and suggested improvements. https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729 (Nov 2016)
69. wikipedia: Mutex. en.wikipedia.org/wiki/Mutual_exclusion (Jan 2019)
70. Wikipedia: Liberty reserve. https://en.wikipedia.org/wiki/Liberty_Reserve (Jun 2020)
71. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger. http://gavwood.com/paper.pdf (Mar 2016)
72. Zalecki, M.: Create and distribute your erc20 token with openzeppelin. https://www.tooploox.com/blog/create-and-distribute-your-erc20-token-with-openzeppelin (Aug 2018)

A Sample of high profile vulnerabilities

In this section, we sample some high profile vulnerabilities, typically ones that have been exploited in real world ERC-20 tokens[36,29,14,11,34]. For each, we (i) briefly explain technical details, (ii) the ability to affect ERC-20 tokens, and (iii) discuss mitigation techniques.

A.1 Multiple withdrawal

This ERC-20-specific issue was originally raised in 2017[68,27]. It can be considered as a *transaction-ordering*[8] or *front-running*[16] attack. There are two

ERC-20 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) can result in allowing a malicious authorized entity to transfer more tokens than the owner wanted. There are several suggestions to extend ERC-20 standard (*e.g.,* MonolithDAO[67] and its extension in OpenZeppelin[39]) by adding new functions (*i.e.,* `decreaseApproval()` and `increaseApproval()`), however, securing `transferFrom()` method is the effective one while adhering specifications of the ERC-20 standard[48].

A.2 Arithmetic Over/Under Flows.

An *integer overflow* is a well known issue in many programming languages. For ERC-20, one notable exploit was in April 2018 that targeted the BEC Token[10] and resulted in some exchanges (*e.g.,* OKEx, Poloniex and HitBTC) suspending deposits and withdrawals of all tokens. Although BEC developers had considered most of the security measurements, only line 261 was vulnerable[23][43]. The attacker was able to pass a combination of input values to transfer large amount of tokens[47]. It was even larger than the initial supply of the token, allowing the attacker to take control of token financing and manipulate the price. In Ethereum, integer overflows do not throw an exception at runtime. This is by design and can be prevented by using the `SafeMath` library[40] wherein `a+b` will be replaced by `a.add(b)` and throws an exception in the case of arithmetic overflow.

A.3 Re-entrancy

One of the most studied vulnerabilities is re-entrancy, which resulted in a US$50M attack on a DApp (called the DAO) in 2016 and triggered an Ethereum hard-fork to revert[24]. At first glance, re-entrancy might seem inapplicable to ERC-20 however any function that changes internal state, such as balances, need to be checked. Further, some ERC-20 extensions could also be problematic. One example is ORBT tokens[49] which support token exchange with ETH without going through a crypto-exchange[56]: an attacker can call the exchange function to sell the token and get back equivalent in ETH. However, if the ETH is transferred in a vulnerable way before reaching the end of the function and updating the balances, control is transferred to the attacker receiving the funds and the same function could be invoked over and over again within the limits of a single transaction, draining excessive ETH from the token contract. This variant of the attack is known as *same-function re-entrancy*, but it has three other variants: *cross-function*, *delegated* and *create-based* [53]. Mutex[69] and CEI[13] 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, developers must be careful enough to update balances before external calls. Implementation of Mutex is more efficient and blocks *cross-function* attacks at the beginning of the function regardless of internal update sequences. CEI can also be considered as a best practice and basic mitigation for the *same-function re-entrancy*.

A.4 Unchecked return values

In Solidity, sending ETH to external addresses is supported by three options: `call.value()`, `transfer()`, or `send()`. The `transfer()` method reverts all changes if the external call fails, while the other two return a boolean value and manual check is required to revert transaction to the initial state[4]. Before the *Istanbul* hard-fork[1], `transfer()` was the preferred way of sending ETH. It mitigates reentry by ensuring ETH recipients would not have enough gas (*i.e.,* a 2300 limit) to do anything meaningful beyond logging the transfer when execution control was passed to them. EIP-1884[28] has increased the gas cost of some opcodes that causes issues with `transfer()`[4]. This has led to community advice to use `call.value()` and rely on one of the above re-entrancy mitigations (*i.e.,* Mutex or CEI)[37,51].

A.5 Frozen Ether

As ERC-20 tokens can receive and hold ETH, just like a user accounts, functions need to be defined to withdraw deposited ETH (including unexpected ETH). If these functions are not defined correctly, an ERC-20 token might hold ETH with no way of recovering it (*cf.* Parity Wallet[42]). If necessary, developers can require multiple signatures to withdraw ETH.

A.6 Unprotected Ether Withdrawal

Improper access control may allow unauthorized persons to withdraw ETH from smart contracts (*cf.* Rubixi[54]). Therefore, withdrawals must be triggered by only authorized accounts and ideally multiple parties.

A.7 State variable manipulation

The `DELEGATECALL` opcode in Ethereum enables a DApp to invoke external functions of other DApps and execute them in the context of calling contract (*i.e.,* the invoked function can modify the state variables of the caller). This makes it possible to deploy libraries once and reuse the code in different contracts. However, the ability to manipulate internal state variables by external functions has lead to incidents where the entire contract was hijacked (*cf.* the second hack

[4] After *Istanbul*, the `fallback()` function consumes more than 2300 Gas if called via `transfer()` or `send()` methods.

of Parity MultiSig Wallet[3]). Preventive techniques is to use `Library` keyword in Solidity to force the code to be stateless, where data is passed as inputs to functions and passed back as outputs and no internal storage is permitted[18]. There are two types of Library: *Embedded* and *Linked*. Embedded libraries have only internal functions (EVM uses `JUMP` opcode instead of `DELEGATECALL`), in contrast to linked libraries that have public or external functions (EVM initiate a "message call"). 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 [30]. It is recommended to use Embedded libraries to mitigate this attack.

A.8 Public visibility

In Solidity, visibility of functions are `Public` by default and they can be called by any external user/contract. In the Parity MultiSig Wallet hack[46], an attacker was able to call public functions and reset the ownership address of the contract, triggering a $31M USD theft. It is recommended to explicitly specify visibility of functions instead of default `Public` visibility.

B A sample of best practices

Due to space, we highlight a few best practices that have been accepted by the Ethereum community to proactively prevent known vulnerabilities[12]. Some best practices are specific to ERC-20, while others are generic for all DApps—in which case, we discuss their relevance to ERC-20.

B.1 Compliance with ERC-20.

According to the ERC-20 specifications, all six methods and two events must be implemented and are not optional. Tokens that do not implement all methods (*e.g.,* GNT which does not implement the `approve()`, `allowance()` and `transferFrom()` functions due to *front-running*[25]) can cause failed function calls from other applications. They might also be vulnerable to complex attacks (*e.g.,* Fake deposit vulnerability[31], Missing return value bug[9]).

B.2 External visibility.

Solidity supports two types of *function calls*: internal and external[19]. Functions calls are different than functions visibility (*i.e.,* Public, Private, Internal and External). Internal function calls expect arguments to be in memory and the EVM copies the arguments to memory. Internal calls use `JUMP` opcodes instead of creating an *EVM call*.[5] Conversely, External function calls create an *EVM call* and can read arguments directly from the `calldata` space. This is cheaper

[5] Also known as "message call" when a contract calls a function of another contract.

than allocating new memory and designed as a read-only byte-addressable space where the data parameter of a transaction or call is held[61]. A best practice is to use external visibility when we expect that functions will be called externally.

B.3 Fail-Safe Mode.

In the case of a detected anomaly or attack on a deployed ERC-20 token, the functionality of the token can be frozen pending further investigation. Similar to Liberty Reserve digital currency service[70], governments may compel to stop the token's ability to operate.

B.4 Firing events.

In ERC-20 standard, there are two defined events: `Approval` and `Transfer`. The first event logs successful allowance changes by token holders and the second logs successful token transfers by the `transfer()` and `transferFrom()` methods. These two events must be fired to notify external application on occurred changes. The external application (*e.g.,* TokenScope[5]) might use them to detect inconsistent behaviors, update balances, show UI notifications, or to check new token approvals. It is a best practice to fire an event for every state variable change.

B.5 Global or Miner controlled variables.

Since malicious miners have the ability to manipulate global Solidity variables (*e.g.,* `block.timestamp`, `block.number`, `block.difficulty`, *etc.*), it is recommended to avoid these variables in ERC-20 tokens.

B.6 Proxy contracts.

An ERC-20 token can be deployed with a pair of contracts: a proxy contract that passes through all the function calls to a second functioning ERC-20 contract[62,38]. One use of proxy contract is when upgrades are required—a new functional contract can be deployed and the proxy is modified to point at the update. Form audit point of view, it is recommended to have non-upgradable ERC-20 tokens.

B.7 DoS with Unexpected revert.

A function that attempts to complete many operations that individually may revert could deadlock if one operation always fails. For example, `transfer()` can throw an exception—if one transfer in a sequence fails, the whole sequence fails. One standard practice is to account for ETH owed and require withdrawals through a dedicated function. In TokenHook, ETH is only transferred to a single party in a single function `sell()`. It seems overkill to implement a whole accounting system for this. As a consequence, a seller that is incapable of receiving ETH (*e.g.,* operating from a contract that is not payable) will be unable to sell their tokens for ETH. However they can recover by transferring the tokens to a new address to sell from.