Tenderize - v2

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **August 1st, 2023 - August 29th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	08/07/2023
0.2	Document Updates	08/16/2023
0.3	Document Updates	08/28/2023
0.4	Draft Review	08/29/2023
0.5	Draft Review	08/29/2023
1.0	Remediation Plan	09/07/2023
1.1	Remediation Plan Review	09/08/2023
1.2	Remediation Plan Review	09/08/2023

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Piotr Cielas	Halborn	Piotr.Cielas@halborn.com
Francisco González	Halborn	Francisco.Villarejo@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Tenderize v2 is a new kind of liquid staking protocol that delivers liquidity for staked assets without centralizing the underlying validator set.

Tenderize engaged Halborn to conduct a security assessment on their smart contracts beginning on August 1st, 2023 and ending on August 29th, 2023. The security assessment was scoped to the smart contracts provided in the Halborn/Tenderize-Contracts GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

1.2 ASSESSMENT SUMMARY

The team at Halborn was provided 4 weeks for the engagement and assigned a full-time security engineer to verify the security of the smart contracts in scope. The security engineer is a blockchain and smart contract security expert with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessments is to:

- Identify potential security issues within the smart contracts.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some security risks that were mostly addressed by Tenderize. The main ones were the following:

- Deploy and initialize the Registry contract in the same transaction to prevent initialization frontrunning.
- Enforce a uniform parameter naming convention.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions. (solgraph)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs. (MythX)
- Static Analysis of security for scoped contract, and imported functions. (Slither)
- Testnet deployment. (Brownie, Remix IDE, Foundry)

2. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

2.1 EXPLOITABILITY

Attack Origin (AO):

Captures whether the attack requires compromising a specific account.

Attack Cost (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

Attack Complexity (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

Metrics:

Exploitability Metric (m_E)	Metric Value	Numerical Value
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

2.2 IMPACT

Confidentiality (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

Integrity (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

Availability (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

Deposit (D):

Measures the impact to the deposits made to the contract by either users or owners.

Yield (Y):

Measures the impact to the yield generated by the contract for either users or owners.

Metrics:

Impact Metric (m_I)	Metric Value	Numerical Value
Confidentiality (C)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

2.3 SEVERITY COEFFICIENT

Reversibility (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

Scope (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

Severity	Score Value Range
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

2.4 SCOPE

Code repositories:

1. Tenderize-Contracts

- Repository: Tenderize/staking
- Commit ID: d8130ea
- Smart contracts in scope:
  - src/adapters/Adapter.sol
  - src/factory/Factory.sol
  - src/adapters/Adapter.sol
  - src/registry/Registry.sol
  - src/registry/RegistryStorage.sol
  - src/registry/Roles.sol
  - src/tenderizer/ITenderizer.sol
  - src/tenderizer/Tenderizer.sol
  - src/tenderizer/TenderizerBase.sol
  - src/tendertoken/TToken.sol
  - src/tendertoken/TTokenStorage.sol
  - src/unlocks/Unlocks.sol
- Remediations Commit ID 1: dd70e188c3191fab57b908dff33abeffd0e33492
- Remediations Commit ID 2: 634c65411caf53d5a4b3350e3e6358b5c531a97b

Out-of-scope:

- Third-party libraries and dependencies.
- Economic attacks.

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	3	3

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) REGISTRY INITIALIZATION CAN BE FRONTRUN	Low (4.1)	RISK ACCEPTED
(HAL-02) INCONSISTENT PARAMETER NAMING CONVENTION	Low (2.5)	SOLVED - 09/07/2023
(HAL-03) CONTRACT PAUSE FEATURE MISSING	Low (2.5)	RISK ACCEPTED
(HAL-04) INCOMPLETE NATSPEC DOCUMENTATION	Informational (0.0)	ACKNOWLEDGED
(HAL-05) INCORRECT METADATA KEY	Informational (0.0)	SOLVED - 09/07/2023
(HAL-06) THE UNLOCK() FUNCTION DOES NOT RETURN TOKENID	Informational (0.0)	ACKNOWLEDGED

FINDINGS & TECH DETAILS

4.1 (HAL-01) REGISTRY INITIALIZATION CAN BE FRONTRUN - LOW (4.1)

Description:

As outlined in the Tenderize_deploy.s.sol deployment script, the Registry contract implementation and proxy are deployed, but the proxy is not initialized until later. This allows any attacker to frontrun this initialization call (since there is also no access control mechanism in the initialize() function), and initialize the Registry contract with malicious implementations of the Tenderizer and Unlocks contracts, allowing a plausible contract takeover and immediate access to user's funds.

Code Location:

Listing 1: Tenderize_deploy.s.sol

```
37          // 1. Deploy Registry (without initialization)
38          // - Deploy Registry Implementation
39          Registry registry = new Registry{salt: salt}();
40          vm.serializeAddress(json_output, "registry_implementation"
↳ , address(registry));
41          // - Deploy Registry UUPS Proxy
42          address registryProxy = address(new ERC1967Proxy{salt:
↳ salt}(address(registry), ""));
43          vm.serializeAddress(json_output, "registry_proxy",
↳ registryProxy);
44          console2.log("Registry Implementation: ", address(registry
↳ ));
45          console2.log("Registry Proxy: ", registryProxy);
```

BVSS:

AO:A/AC:L/AX:H/C:C/I:C/A:C/D:C/Y:C/R:P/S:C (4.1)

Recommendation:

It is recommended that contracts are deployed and initialized in the same transaction, in order to prevent the initialization call from being frontrun. If the deployment address is needed for further deployment or initialization operations, deployment addresses could be computed deterministically with the CREATE2 library.

Remediation Plan:

RISK ACCEPTED: The Tenderize team accepted the risk of this finding. The CREATE2 library will be used when possible, and special attention to front-runned initialization will be paid, in order to discard them.

4.2 (HAL-02) INCONSISTENT PARAMETER NAMING CONVENTION - LOW (2.5)

Description:

In the Unlocks contract, there are many getter functions that accept id or tokenId as input parameters. id parameter references to unlock ID issued when unlocking any amount of tokens from delegators, while tokenId references the token ID of the Unlocks NFT issued to any user, representing the amount of GRT they will be able to withdraw.

However, it has been detected that id and tokenId are being used inconsistently across different functions in the Unlocks contract, which might introduce inconsistencies or errors both in the front-end or while interacting with the contracts.

Due to this usage, it was detected that tokenURI() function likely reverts, since it is directly using the id input parameter as if it were tokenId (without calling _encodeTokenId()), and it most likely queries for a nonexistent token, causing the call to revert.

Code Location:

Listing 2: Unlocks.sol (Lines 82,86)

```
80     /**
81      * @notice Returns the tokenURI of an unlock token
82      * @param id ID of the unlock token
83      * @return tokenURI of the unlock token
84      */
85     function tokenURI(uint256 id) public view virtual override
↳ returns (string memory) {
86         require(_ownerOf[id] != address(0), "non-existent token");
87         return renderer.json(id);
88     }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:L/D:N/Y:N/R:N/S:U (2.5)

Recommendation:

It is recommended to use a consistent naming convention for function input parameters.

Remediation Plan:

SOLVED: The Tenderize team solved the issue in commit 634c654 by using explicit naming convention, using unlockId and tokenId instead of simply id.

4.3 (HAL-03) CONTRACT PAUSE FEATURE MISSING - LOW (2.5)

Description:

It was identified that no high-privileged user can pause any of the scoped contracts. In the event of a security incident, the owner would not be able to stop any plausible malicious actions. Pausing the contract can also lead to more considered decisions.

BVSS:

AO:A/AC:L/AX:M/C:N/I:L/A:N/D:L/Y:L/R:N/S:U (2.5)

Recommendation:

Consider adding the pausable functionality to the contract.

Remediation Plan:

RISK ACCEPTED: The Tenderize team accepted the risk of this finding in order to avoid introducing any privileged user or upgradability mechanism in the contracts.

4.4 (HAL-04) INCOMPLETE NATSPEC DOCUMENTATION - INFORMATIONAL (0.0)

Description:

Natspec documentation is useful for internal developers that need to work on the project, external developers that need to integrate with the project, auditors that have to review it but also for end users given that many chain explorers have officially integrated the support for it directly on their site.

It was detected that the scoped contracts have an incomplete **natspec** documentation.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

Consider adding the missing **natspec** documentation, adhering to the format guideline included in Solidity documentation.

Remediation Plan:

ACKNOWLEDGED: The Tenderize team acknowledged this finding.

4.5 (HAL-05) INCORRECT METADATA KEY - INFORMATIONAL (0.0)

Description:

When calling the getMetadata() function in the Unlocks contract, the tokenId is supplied to obtain the metadata of the token. Using this tokenId, Tenderizer address and unlockId are extracted.
Then, the extracted unlockId is used to populate the value of the tokenId key, which might be confusing.

Code Location:

Listing 3: Unlocks.sol (Lines 95,96,102)

```
90      /**
91       * @notice Returns the metadata of an unlock token
92       * @param tokenId ID of the unlock token
93       * @return metadata of the unlock token
94       */
95      function getMetadata(uint256 tokenId) external view returns (
↳ Metadata memory metadata) {
96          (address tenderizer, uint256 id) = _decodeTokenId(tokenId)
↳ ;
97          address asset = Tenderizer(tenderizer).asset();
98
99          return Metadata({
100             amount: Tenderizer(tenderizer).previewWithdraw(id),
101             maturity: Tenderizer(tenderizer).unlockMaturity(id),
102             tokenId: id,
103             symbol: ERC20(asset).symbol(),
104             name: ERC20(asset).name(),
105             validator: Tenderizer(tenderizer).validator()
106         });
107     }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

If the tokenId was meant to be stored in the metadata, then tokenId should be used, not id. On the other hand, if the unlockId was meant to be stored, the key should be named id or unlockId, not tokenId, which might be confused with the NFT token ID.

Remediation Plan:

SOLVED: The Tenderize team solved the issue in commit 634c654 by renaming Metadata.tokenId to Metadata.unlockId.

4.6 (HAL-06) THE UNLOCK() FUNCTION DOES NOT RETURN TOKENID - INFORMATIONAL (0.0)

Description:

When calling the unlock() function from the Tenderizer contract, the unlockId is returned from the validator. With that unlockId and the Tenderizer address, the tokenId of the token that represents the NFT containing the unlock can be calculated. Although the createUnlock() function from the Unlocks contract returns the minted tokenID, this is ignored by Tenderizer contract, forcing users to either check the event emitted while minting the token or calculate the address by themselves every time, highly reducing the usability.
In addition, since only unlockId is returned, and due to the naming inconsistencies observed in the contracts, there is a chance that unlockId is confused with tokenId, introducing errors or inconsistencies that might prevent the contracts from working properly.

Code Location:

The following instance of this finding was identified:

Listing 4: Tenderizer.sol (Line 97)

```
92     /**
93      * @notice Unlock tTokens to withdraw assets at maturity
94      * @param assets amount of assets to unlock
95      * @return unlockID of the unlock
96      */
97     function unlock(uint256 assets) external returns (uint256
↳ unlockID) {
98         _rebase();
99
100        // burn tTokens before creating an `unlock`
101        _burn(msg.sender, assets);
102
103        // unlock assets and get unlockID
```

```
104         unlockID = _unstake(validator(), assets);
105
106         // create unlock of unlockID
107         _unlocks().createUnlock(msg.sender, unlockID);
108
109         // emit Unlock event
110         emit Unlock(msg.sender, assets, unlockID);
111     }
```

BVSS:

AO:A/AC:L/AX:H/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation:

Along with the previous recommendations of unifying naming conventions, also returning tokenId might be useful for future contract interactions.

Remediation Plan:

ACKNOWLEDGED: The Tenderize team acknowledged this finding.

AUTOMATED TESTING

5.1 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

Results:

Slither results for Registry.sol	
Finding	**Impact**
Registry.setFee(address,uint96).fee (src/registry/Registry.sol#142) shadows: - Registry.fee(address) (src/registry/Registry.sol#88-90) (function)	Low
Registry.setTreasury(address).treasury (src/registry/Registry.sol#154) shadows: - Registry.treasury() (src/registry/Registry.sol#104-107) (function)	Low
Registry.registerTenderizer(address,address,address).tenderizer (src/registry/Registry.sol#131) shadows: - Registry.tenderizer() (src/registry/Registry.sol#63-66) (function)	Low
Registry.isTenderizer(address).tenderizer (src/registry/Registry.sol#97) shadows: - Registry.tenderizer() (src/registry/Registry.sol#63-66) (function)	Low
Registry.registerAdapter(address,address).adapter (src/registry/Registry.sol#117) shadows: - Registry.adapter(address) (src/registry/Registry.sol#80-82) (function)	Low

Finding	Impact
End of table for Registry.sol	

Slither results for TToken.sol	
Finding	**Impact**
TToken._burn(address,uint256).shares (src/tendertoken/TToken.sol#262) is a local variable never initialized	Medium
TToken.permit(address,address,uint256,uint256,uint8,bytes32,bytes32) (src/tendertoken/TToken.sol#190-227) uses timestamp for comparisons Dangerous comparisons: - deadline < block.timestamp (src/tendertoken/TToken.sol#202)	Low
End of table for TToken.sol	

All the issues flagged by Slither were manually reviewed by Halborn. Reported issues were either considered as false positives or are already included in the report findings.

No issues were found by Slither in Adapter.sol, Factory.sol, RegistryStorage.sol, Roles.sol, Tenderizer.sol, TenderizerBase.sol, TTokenStorage.sol, and Unlocks.sol.

5.2 AUTOMATED SECURITY SCAN

Description:

Halborn used automated security scanners to assist with detection of well-known security issues and to identify low-hanging fruits on the targets for this engagement. Among the tools used was MythX, a security analysis service for Ethereum smart contracts. MythX performed a scan on the smart contracts and sent the compiled results to the analyzers in order to locate any vulnerabilities.

Results:

Adapter.sol

Report for adapters/Adapter.sol
https://dashboard.mythx.io/#/console/analyses/ccde63cd-ecf6-444f-812a-491930d24d91

Line	SWC Title	Severity	Short Description
35	(SWC-000) Unknown	Medium	Builtin symbol "msg" shadowing
35	(SWC-108) State Variable Default Visibility	Low	State variable visibility is not set.

Factory.sol

Report for src/factory/Factory.sol
https://dashboard.mythx.io/#/console/analyses/29a970ce-6a80-4ec6-894c-df0fd4643caa

Line	SWC Title	Severity	Short Description
29	(SWC-108) State Variable Default Visibility	Low	State variable visibility is not set.
30	(SWC-108) State Variable Default Visibility	Low	State variable visibility is not set.

Registry.sol

Report for src/registry/Registry.sol
https://dashboard.mythx.io/#/console/analyses/93ae0a23-7602-4f2f-aac3-2d49b2d6b85a

Line	SWC Title	Severity	Short Description
28	(SWC-108) State Variable Default Visibility	Low	State variable visibility is not set.
29	(SWC-108) State Variable Default Visibility	Low	State variable visibility is not set.

RegistryStorage.sol

No issues were identified by MythX.

Roles.sol

No issues were identified by MythX.

Tenderizer.sol

```
Report for src/tenderizer/Tenderizer.sol
https://dashboard.mythx.io/#/console/analyses/5db63676-019c-4da3-a478-82ae4ab4c29c
https://dashboard.mythx.io/#/console/analyses/8fe315f1-5ea6-43cd-b1a9-1d79f704e92b
https://dashboard.mythx.io/#/console/analyses/50027dc8-08b8-40bd-bfb2-19b3471a04cf
```

Line	SWC Title	Severity	Short Description
225	(SWC-123) Requirement Violation	Low	Requirement violation.

TenderizerBase.sol

No issues were identified by MythX.

TToken.sol

No issues were identified by MythX.

TTokenStorage.sol

No issues were identified by MythX.

Unlocks.sol

No issues were identified by MythX.

All the issues flagged by MythX were manually reviewed by Halborn. Reported issues were either considered as false positives or are already included in the report findings.

THANK YOU FOR CHOOSING

// HALBORN