Tenderize - The Graph Adapter

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **August 30th, 2023 - September 5th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	08/30/2023
0.2	Document Updates	09/01/2023
0.3	Draft Version	09/05/2023
0.4	Draft Review	09/07/2023
0.5	Draft Review	09/07/2023
1.0	Remediation Plan	09/27/2023
1.1	Remediation Plan Review	09/27/2023
1.2	Remediation Plan Review	09/28/2023

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

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 30th, 2023 and ending on September 5th, 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 5 days 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 one was the following:

- Ensure that tokensLockedUntil is considered an epoch and not a block number.

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)	1
	Specific (AO:S)	0.2
Attack Cost (AC)	Low (AC:L)	1
	Medium (AC:M)	0.67
	High (AC:H)	0.33
Attack Complexity (AX)	Low (AX:L)	1
	Medium (AX:M)	0.67
	High (AX:H)	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)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25
Scope (s)	Changed (S:C)	1.25
	Unchanged (S:U)	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/GraphAdapter.sol
- Remediations Commit ID 1: ff8716e2705ce615c7967f912621cbe00e9109bf
- Remediations Commit ID 2: dd70e188c3191fab57b908dff33abeffd0e33492

Out-of-scope:

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

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	0	0	2	2

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) TOKENSLOCKEDUNTIL IS COMPARED WITH BLOCK.NUMBER INSTEAD OF EPOCH	Critical (10)	SOLVED - 09/27/2023
(HAL-02) THE UNLOCKMATURITY() FUNCTION MIGHT RETURN INNACURATE VALUES	Low (2.5)	RISK ACCEPTED
(HAL-03) CONTRACT PAUSE FEATURE MISSING	Low (2.5)	RISK ACCEPTED
(HAL-04) INCOMPLETE NATSPEC DOCUMENTATION	Informational (0.0)	ACKNOWLEDGED
(HAL-05) CONFUSING VARIABLE NAMING	Informational (0.0)	SOLVED - 09/27/2023

FINDINGS & TECH DETAILS

4.1 (HAL-01) TOKENSLOCKEDUNTIL IS COMPARED WITH BLOCK.NUMBER INSTEAD OF EPOCH - CRITICAL(10)

Description:

After delegating GRT to any The Graph indexer, delegated amounts can be unlocked anytime. Still, a cool-down period of ~28 days (in which the deposit does not accrue any reward) applies. Since the Tenderizer contracts act as deposit aggregators, it is expected that multiple stake, unlock, and withdrawal operations take place frequently. However, it was detected that the Tenderizer contracts can handle only a single unlock per cool-down period, preventing other users from unlocking (or even withdrawing already unlocked GRT) until the next cool-down period. This decreases the usability of Tenderizers, and can also allow an attacker to effectively block the user's tokens forever in the contract via front-running the first legitimate unlock in each cool-down period.

Code Location:

Listing 1: Tenderizer.sol (Line 213)

```
212     function _unstake(address validator, uint256 amount) internal
  ↳ returns (uint256 unlockID) {
213         unlockID = abi.decode(_adapter()._delegatecall(abi.
  ↳ encodeCall(_adapter().unstake, (validator, amount))), (uint256));
214     }
```

Every time that a user calls the unlock() or withdraw() functions in a Tenderizer to unlock or withdraw GRT, the unstake() or withdraw() functions from the GraphAdapter contract are called, respectively.

These functions call the _processWithdrawals() function, which withdraws fully unlocked amounts from the delegated contract to the Tenderizer, to be finally withdrawn by the final users. If this function is called while an unstake operation from any user is still in the cool-down period, it always reverts with the !tokens error since there are no tokens to be withdrawn from the contract.

This scenario was supposedly handled in _processWithdraw() function with the next statement: (which tries to prevent GRAPH.withdrawDelegated() function from being called in case no tokens could be withdrawn):

Listing 2: GraphAdapter.sol (Line 236)

```
233     function _processWithdraw(address validator) internal {
234         // withdrawal isn't ready: no-op
235         uint256 tokensLockedUntil = GRAPH.getDelegation(validator, address(this)).tokensLockedUntil;
236         if (tokensLockedUntil == 0 || tokensLockedUntil > block.number) return;
237
238         Storage storage $ = _loadStorage();
239
240         // withdraw undelegated
241         unchecked {
242             // $.currentEpoch - 1 is safe as we only call this function after at least 1 _processUnstake
243             // which increments $.currentEpoch, otherwise del.tokensLockedUntil would still be 0 and we would
244             // not reach this branch
245             $.epochs[$.currentEpoch - 1].amount = GRAPH.withdrawDelegated(validator, address(0));
246         }
247     }
```

However, that check fails since it compares tokensLockedUntil (an epoch around 954 at the time of writing this report) with block.number, which is a much greater number.

Proof of Concept:

Steps to Reproduce:

1. user1, user2, and user3 call deposit() to stake GRT.
2. After that, user1 tries to unlock their GRT by calling unlock().
3. After the cool-down period passes, user2 calls unlock() with a small amount of GRT.

4. Now, no one can call unlock() or withdraw() until the cool-down period passes, effectively locking user1 tokens.

Thereafter, if user2 decides to frontrun the first valid unlock() call after the cool-down period passes, the deposited funds are effectively blocked forever.

```
Running 1 test for test/mainnet.t.sol:ForkingTest
[PASS] testFail_hal01_unlockDoS_2() (gas: 1203962)
Logs:
  user1 tTokenOut : 995000000000000000000000
  user2 tTokenOut : 1990000000000000000000000
  user3 tTokenOut : 2985000000000000000000000

user1 unlockId : 1

user2 unlockId : 2

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 8.53s
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:C/D:M/Y:M/R:N/S:C (10)

Recommendation:

Ensure that GRAPH.withdrawDelegated() is called only when there is no active unlock cool-down period.

Remediation Plan:

SOLVED: The Tenderize team solved this finding in commit dd70e18 by comparing tokensLockedUntil with the appropriate value.

4.2 (HAL-02) THE UNLOCKMATURITY() FUNCTION MIGHT RETURN INNACURATE VALUES - LOW (2.5)

Description:

The unlockMaturity() function is supposed to return the maturity date of any unlock, which is the block number in which the unlock can be withdrawn. For unlocks created in the current Adapter Epoch, a complete THAWING_PERIOD after the current one ends is returned. If the unlock was created on the previous Adapter Epoch, the end of the current THAWING_PERIOD is returned.

However, this is heavily based on the daily usage of the unstake() and withdraw() functions, since they are the only way to call the _processWithdrawals() function, which is the function that actually performs unlocks and withdraws from the Graph Staking contract, and increments the currentEpoch value.

This means that, if a single user calls the unstake() function in any Adapter Epoch but no one else calls unstake() or withdraw() functions after the current THAWING_PERIOD ends, the actual GRAPH.undelegate() call is never performed, causing the GRT tokens to never be actually undelegated. The actual unlock maturity date would be a complete THAWING_PERIOD after the new Adapter Epoch starts.

This behavior could also mean that unstaking cool down periods could be extended indefinitely in some cases.

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 changing the unlockMaturity() logic in order to always provide correct values, or have into account that unlockMaturity() might return **MINIMAL** maturity block numbers, but it will always depend on the timing of _processWithdrawals() calls.

Remediation Plan:

RISK ACCEPTED: The Tenderize team accepted the risk of this finding. In addition, the Tenderize team stated:

> We assume that if a user has called unlock he/she will also call withdraw within quite a respectable timeframe to process the epoch and thus have unlockMaturity() return accurate values. The unlockMaturity function is also purely used as a external getter (by the unlocks metadata for used by front-ends).

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 addition, the Tenderize team stated:

> We prefer not having a high-privilege entity being able to affect liveness. Tenderize v2 has been designed with a governance minimal approach, and tries to reduce upgradeability and admin keys whenever possible.

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. In addition, the Tenderize team stated:

> We feel like all off the necessary external APIs are covered.

4.5 (HAL-05) CONFUSING VARIABLE NAMING - INFORMATIONAL (0.0)

Description:

It was noted that identical variable names are being used in the GraphAdapter contract for different variables. This could easily induce errors when interacting, managing or even developing the protocol, while dramatically decreasing the code readability.

Code Location:

Listing 3: GraphAdapter.sol (Line 92)

```
82     function unlockMaturity(uint256 unlockID) external view
   ↳ override returns (uint256) {
83         Storage storage $ = _loadStorage();
84         Unlock memory unlock = $.unlocks[unlockID];
85         uint256 THAWING_PERIOD = GRAPH.thawingPeriod();
86         // if userEpoch == currentEpoch, it is yet to unlock
87         // => unlockTime + thawingPeriod
88         // if userEpoch == currentEpoch - 1, it is processing
89         // => unlockTime
90         // if userEpoch < currentEpoch - 1, it has been processed
91         // => 0
92         uint256 tokensLockedUntil = $.lastEpochUnlockedAt +
   ↳ THAWING_PERIOD;
93         if (unlock.epoch == $.currentEpoch) {
94             return THAWING_PERIOD + tokensLockedUntil;
95         } else if (unlock.epoch == $.currentEpoch - 1) {
96             return tokensLockedUntil;
97         } else {
98             return 0;
99         }
100    }
```

Here, tokensLockedUntil references the block number in which the introduced unlockID becomes withdrawable.

Listing 4: GraphAdapter.sol (Lines 210,212)

```
209     function _processUnstake(address validator) internal {
210         IGraphStaking.Delegation memory del = GRAPH.getDelegation(validator, address(this));
211         // undelegation already ungoing: no-op
212         if (del.tokensLockedUntil != 0) return;
213
214         Storage storage $ = _loadStorage();
215         uint256 currentEpochAmount = $.epochs[$.currentEpoch].amount;
216
217         // if current epoch amount is non-zero
218         // => progress epoch and undelegate from underlying
219         // if current epoch is zero and previous epoch is non-zero
220         // => only progress epoch, as withdrawal of last epoch is processed
221         // => would return at del.tokensLockedUntil check if withdrawal is yet to process
222         // if current and previous epoch are zero
223         // => no-op - optimization, no need to progress epochs if last two are emtpy
224         // => ie. no pending unlocks, no unlocks processing
225         if (currentEpochAmount != 0) {
226             ++$.currentEpoch;
227             $.lastEpochUnlockedAt = block.number;
228
229             // calculate shares to undelegate from The Graph
230             uint256 undelegationShares = currentEpochAmount * 1 ether / $.tokensPerShare;
231
232             // account for possible rounding error
233             undelegationShares = del.shares < undelegationShares ? del.shares : undelegationShares;
234
235             // undelegate
236             GRAPH.undelegate(validator, undelegationShares);
237         } else if ($.epochs[$.currentEpoch - 1].amount != 0) {
238             ++$.currentEpoch;
239             $.lastEpochUnlockedAt = block.number;
240         }
241     }
```

On the other hand, `tokensLockedUntil` represents in this function the

EPOCH in which the next unlock is withdrawable, even if The Graph's IStakingData Interface describes it as a block number.

In the same fashion, the GraphAdapter's currentEpoch variable could easily be confused with The Graph's epoch.

Also, the usage of Epoch struct could also be confused with epoch key of Unlock struct, and with epochs key of Storage.

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 using a more explicit variable and function naming convention for every contract in the protocol.

Remediation Plan:

SOLVED: The Tenderize team solved this issue in commit ff8716e by renaming the variable to unlockBlock in both code and comments in the unlockMaturity () function.

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 graph.sol	
Finding	**Impact**
GraphAdapter.rebase(address,uint256) (src/adapters/GraphAdapter.sol#136-187) performs a multiplication on the result of a division: - _tokensPerShare = delPool.tokens * 1000000000000000000 / delPool.shares (src/adapters/GraphAdapter.sol#141) - staked = delegation.shares * _tokensPerShare / 1000000000000000000 (src/adapters/GraphAdapter.sol#153)	Medium
GraphAdapter._processUnstake(address) (src/adapters/GraphAdapter.sol#199-231) ignores return value by GRAPH.undelegate(validator,undelegationShares) (src/adapters/GraphAdapter.sol#226)	Medium
GraphAdapter.stake(address,uint256) (src/adapters/GraphAdapter.sol#94-97) ignores return value by GRAPH.delegate(validator,amount) (src/adapters/GraphAdapter.sol#96)	Medium
End of table for graph.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.

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:

GraphAdapter.sol

Report for src/adapters/GraphAdapter.sol
https://dashboard.mythx.io/#/console/analyses/a840c5f1-1f70-4e92-81d0-3b5c2d45cb74

Line	SWC Title	Severity	Short Description
96	(SWC-113) DoS with Failed Call	Medium	Multiple calls are executed in the same transaction.
200	(SWC-113) DoS with Failed Call	Low	Multiple calls are executed in the same transaction.
217	(SWC-120) Weak Sources of Randomness from Chain Attributes	Low	Potential use of "block.number" as source of randonmness.
229	(SWC-120) Weak Sources of Randomness from Chain Attributes	Low	Potential use of "block.number" as source of randonmness.
236	(SWC-120) Weak Sources of Randomness from Chain Attributes	Low	Potential use of "block.number" as source of randonmness.
236	(SWC-120) Weak Sources of Randomness from Chain Attributes	Low	A control flow decision is made based on The block.number environment variable.
245	(SWC-113) DoS with Failed Call	Low	Multiple calls are executed in the same transaction.
245	(SWC-107) Reentrancy	Low	Write to persistent state following external call
245	(SWC-107) Reentrancy	Low	Read of persistent state following external call

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