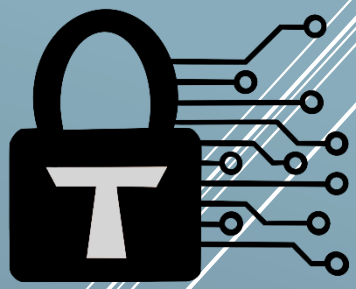
Trust Security

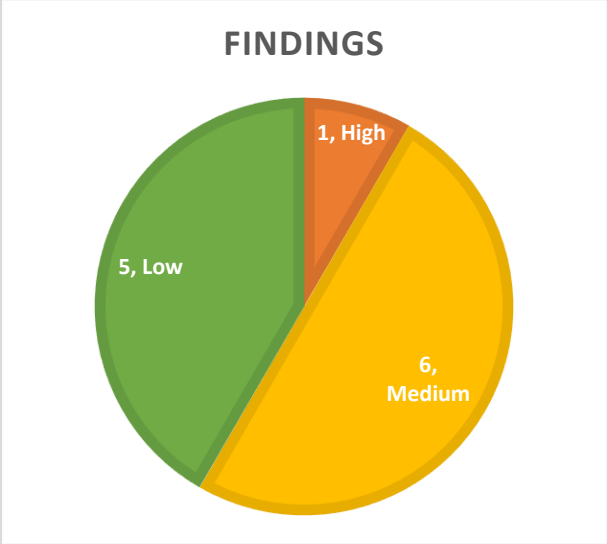


Smart Contract Audit

Tenderize

15/02/2024

Executive summary

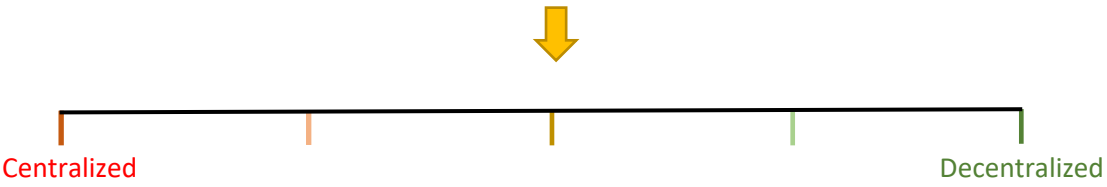


Category	Liquid staking
Audited file count	14
Lines of Code	858
Auditor	Trust
Time period	25/10/23-15/11/23

Findings

Severity	Total	Open	Fixed	Acknowledged
High	1	-	1	-
Medium	6	-	3	3
Low	6	-	1	5

Centralization score



Signature

Digitally signed by Trust
Date: 2024.02.15 11:49:44 +02'00'

Document properties

Versioning

Version	Date	Description
0.1	15/11/23	Client report
0.2	30/11/23	Mitigation review
0.3	10/02/24	Mitigation review #2
0.4	15/02/24	Client response

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

- adapters/GraphAdapter.sol
- adapters/Adapter.sol
- registry/Roles.sol
- registry/RegistryStorage.sol
- register/Registry.sol
- unlocks/Renderer.sol
- unlocks/Base64.sol
- unlocks/Unlocks.sol
- tenderizer/TenderizerBase.sol
- tenderizer/ITenderizer.sol
- tenderizer/Tenderizer.sol
- tendertoken/TTokenStorage.sol
- tendertoken/TToken.sol
- factory/Factory.sol

Repository details

- **Repository URL:** https://github.com/Tenderize/staking
- **Audit hash:** ed8d90e0c2cec81fd5acae209225fb8d1128c3a0
- **Mitigation review commit hashes:**
  - M-1: cb8a17b64f692e976cc7394e5b72611ad7321435
  - M-2: 9b5dd4594281d70091c53b70a0308dc06ad62c2b
  - M-4: 48f36138bf2b6dc762fa9ae9b340149680b1ac52
  - M-5: bb454af9aae897bc09a7338f7df5c3c1ffb625af
  - L-2: 7f95435e1c860be889313cdaa621516f242c9985
  - R-4: b1e40ced83a70dbe19c66bc5388bf4e900279646
  - R-6: 0ed851e8416847df5c98a6d8739db52873a9c341
- **Mitigation review #2 commit hash:** c09b98867599fd0f44a0f3f7b88dd7e60159b85f

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Since its inception it has safeguarded over 30 clients through private services and over 30 additional projects through bug bounty submissions.

About the Auditors

Trust has established a dominating presence in the smart contract security ecosystem since 2022. He is a resident on the Immunefi, Sherlock and C4 leaderboards and is now focused in auditing and managing audit teams under Trust Security. When taking time off auditing and bug hunting, he enjoys assessing bounty contests in C4 as a Supreme Court judge.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Excellent**	The project is modularized very well to minimize complexity.
Documentation	**Good**	Project is mostly well documented.
Best practices	**Excellent**	Project consistently adheres to latest industry standards.
Centralization risks	**Moderate**	Privileged actors can perform dangerous operations.

Findings

High severity findings

TRST-H-1 Attacker can steal deposits by abusing a stale Graph tokensPerShare

- **Category:** Accounting issues
- **Source:** GraphAdapter.sol
- **Status:** Fixed

Description

This issue was introduced when combining the fixes for M-2 and R-6. When the pool has no delegators and **shares** is zero, **_tokensPerShare** gets the default value that will be used on the Graph side.

```
uint256 _tokensPerShare = delPool.shares != 0 ? delPool.tokens * 1
ether / delPool.shares : 1 ether;
```

Also, in case there were no profits since last *rebase()*, there is an early exit, with no rebase changes.

```
if (staked > oldStake) {
    // handle rewards
    // To reduce long waiting periods we want to still reward users
    // for which their stake is still to be unlocked
    // because technically it is not unlocked from the Graph either
    // We do this by adding the rewards to the current epoch
    uint256 currentEpochAmount = (staked - oldStake) *
currentEpoch.amount / oldStake;
    currentEpoch.amount += currentEpochAmount;
} else {
    return newStake;
}
$.epochs[$.currentEpoch] = currentEpoch;
$.tokensPerShare = _tokensPerShare;
// slash/rewards is already accounted for in
$.epochs[$.currentEpoch].amount
newStake = staked - currentEpoch.amount;
```

The two behaviors combined introduce an opportunity for **tokensPerShare** to be stale. Suppose a delegator has received rewards and is at **1.2e18** share price. At this point, all delegators, from Tenderizer and externally, undelegate. All rebases from that moment and until the next rewarding will reach the early exit, and not update the **tokensPerShare**.

Now suppose an attacker deposits through the tenderizer, followed by victims. Attacker can withdraw the victim's tokens through the adapter.

```
if (currentEpochAmount != 0) {
    ++$.currentEpoch;
    $.lastEpochUnlockedAt = block.number;
    // calculate shares to undelegate from The Graph
    uint256 undelegationShares = currentEpochAmount * 1 ether /
```

```
$.tokensPerShare;
    // account for possible rounding error
    undelegationShares = del.shares < undelegationShares ? del.shares
: undelegationShares;
    // undelegate
    GRAPH.undelegate(validator, undelegationShares);
```

The amount of shares requested for undelegation will use the stale **tokensPerShare** value, overestimating the share count. This means when the victims eventually withdraw their deposit, there will be less Graph delegator shares to redeem.

Recommended mitigation

Make sure **tokensPerShare** is always updated in *rebase()*.

Team response

Fixed.

Mitigation review

The code now always calculates the tokens / share ratio from the Graph staking contract. It is always up to date.

Medium severity findings

TRST-M-1 The Base64 implementation used by the Unlock NFT is broken on Arbitrum

- **Category:** Interoperability issues
- **Source:** Base64.sol
- **Status:** Fixed

Description

The Base64 library used is an almost exact copy of the Brechtpd base64 library. It is a known issue of the library that a commit from 2021 has broken the code, as it now uses the **mstore8** instruction directly which is not well supported on Arbitrum. The *encode()* function is used by the *tokenURI()* function in the Unlocks NFT. Since the function is mandatory for ERC721 implementations, this could break integrations in unforeseen ways.

Recommended mitigation

Consider replacing the **mstore8** opcode with **mstore** and bit-shift arithmetic. Refer to the highlighted commit for the working version of the code.

Team response

This is only used in the “Renderer” contract and does not affect user value of the NFT. Moved to using mstore with bitshifts instead of mstore8.

Mitigation review

Issue has been fixed as suggested.

TRST-M-2 Rebasing does not work when validator has no delegators, deposit() functionality is impacted

- **Category:** Divide-by-zero errors
- **Source:** GraphAdapter.sol
- **Status:** Fixed

Description

The GraphAdapter's *rebase()* function calculates any changes in value between the last recorded delegator share value and the current value. It calculates the **_tokensPerShare** below:

```
function rebase(address validator, uint256 currentStake) external
override returns (uint256 newStake) {
    Storage storage $ = _loadStorage();
    Epoch memory currentEpoch = $.epochs[$.currentEpoch];
    IGraphStaking.DelegationPool memory delPool =
GRAPH.delegationPools(validator);
    uint256 _tokensPerShare = delPool.tokens * 1 ether /
delPool.shares;
```

Note that when the validator is not currently receiving any delegations, the **delPool.shares** value will be zero and the function would revert. The main impact is that the Tenderizer's *deposit()* cannot be called unless the validator is already receiving delegations.

```
function deposit(address receiver, uint256 assets) external returns
(uint256) {
    _rebase();
    // transfer tokens before minting (or ERC777's could re-enter)
    ERC20(asset()).safeTransferFrom(msg.sender, address(this),
assets);
```

Recommended mitigation

Add an early-exit statement in the *rebase()* function, in case the shares amounts is zero.

Team response

This is only used in the “Renderer” contract and does not affect user value of the NFT. Moved to using mstore with bitshifts instead of mstore8

Mitigation review

The function will now complete successfully.

TRST-M-3 Attacker can perform profitable sandwich attacks on token rebases

- **Category:** MEV attacks
- **Source:** Tenderizer.sol
- **Status:** Acknowledged

Description

Tenderizer tokens have rebasing functionality. Once rewards are sent to delegators, the token amount of the user increases through *rebase()* which is triggered in almost all code paths. Since there is expected to be a market for Tenderize tokens, this creates a significant arbitrage opportunity. An attacker can spot incoming rewards and frontrun them with a *stake()* call. Then after their balance has increased, sell on external markets for profit. This would dilute the profits of honest stakers who have deposits the money for a longer period of time. A similar attack would be selling Ttokens before a slashing event occurs which reduces the user's token amount.

Recommended mitigation

Consider changing the rewarding algorithm to make it stream over time, or implement a wait period before being eligible for rewards.

Team response

This is an underlying problem of Graph Protocol which manifests in Tenderize equally. It is mitigated by the delegation tax which makes this type of attack not profitable in case of The Graph.

TRST-M-4 Mishandling of the burn amount could trigger serious issues in Adapters

- **Category:** Rounding issues
- **Source:** TToken.sol
- **Status:** Acknowledged

Description

The *_burn()* function in TToken receives an asset amount and reduces the equivalenting shares from the source balance.

```
function _burn(address from, uint256 assets) internal virtual {
    uint256 shares;
    if (assets == 0) revert ZeroAmount();
    if ((shares = convertToShares(assets)) == 0) return;
    Storage storage $ = _loadStorage();
    $._totalSupply -= assets;
    $.shares[from] -= shares;
    // Cannot underflow because a user's balance
    // will never be larger than the total supply.
    unchecked {
        $._totalShares -= shares;
    }
}
```

The function uses *convertToShares()*, which rounds down. It also has an early return statement. This means a user can call *unlock()* and trigger a withdrawal of a tiny amount without even holding any tokens. There is also danger that the state machine of the adapters would get broken since more than the currently staked assets can be unstaked. In the GraphAdapter a potential state lock is prevented using the code below:

```
// calculate shares to undelegate from The Graph
uint256 undelegationShares = currentEpochAmount * 1 ether /
$.tokensPerShare;
// account for possible rounding error
undelegationShares = del.shares < undelegationShares ? del.shares :
undelegationShares;
// undelegate
GRAPH.undelegate(validator, undelegationShares);
```

The epoch amount, which is larger than the amount staked, is taken and then adjusted to be at most the registered delegator shares count. It isn't known how other adapters would handle a situation where more is unstaked than is staked.

Recommended mitigation

Do not perform an early return from *_burn()*. Additionally, round the number of shares up to prevent any malicious behavior by attackers.

Team Response

Now reverts if not at least one share worth of assets is burnt**.**

Mitigation review

The first issue has been addressed, but the function still rounds down the number of shares burnt. As seen from historical attacks, it is important in general to always round in favor of the protocol, to reduce attack opportunities.

Team Response

The team is aware of the need to round upwards theoretically, but it is not an actual concern for the implemented Tenderizer adapters as the token/share ratio is not controlled by attacker.

TRST-M-5 TToken asset conversions could revert, impacting functionality of Tenderizer

- **Category:** Division by zero issues
- **Source:** TToken.sol
- **Status:** Fixed

Description

The TToken has asset-to-shares conversion as implemented below:

```
function convertToShares(uint256 assets) public view returns
(uint256) {
    Storage storage $ = _loadStorage();
```

```
    uint256 _totalShares = $._totalShares; // Saves an extra SLOAD if
slot is non-zero
    return _totalShares == 0 ? assets :
assets.mulDivDown(_totalShares, $._totalSupply);
}
```

An issue will arise when **_totalShares != 0** and **$._totalSupply = 0**, at that point the calculation will revert due to division by zero. This situation would occur after a slashing action, for example. It's important to handle this case properly to not lose functionality like *deposit()*.

Recommended mitigation

Consider validating **$._totalSupply** is not zero before the division operation. If it is, it is reasonable to return **assets**.

Team response

Now checking if **$._totalSupply** is non-zero, else return assets. We no longer have to check for **$._totalShares** being non-zero here due to shares always being 0 when supply is zero, but vice-versa is not necessarily true as pointed out (e.g. full slash).

Mitigation review

The issue has been correctly fixed.

TRST-M-6 TTokens output stale balanceOf() and totalSupply() values, leading to integration risks

- **Category:** Logical flaws
- **Source:** TToken.sol
- **Status:** Acknowledged

Description

The TToken is a rebasing token, therefore on any transfer, *rebase()* is called. However, the *balanceOf()* and *totalSupply()* functions of the ERC20 interface can potentially return stale values.

```
function balanceOf(address account) public view virtual returns
(uint256) {
    return convertToAssets(_loadStorage().shares[account]);
}
/**
 * @notice Returns the total supply of the tToken
 * @return Total supply of the tToken
 */
function totalSupply() public view virtual returns (uint256) {
    Storage storage $ = _loadStorage();
    return $._totalSupply;
}
```

This represents severe integration risks. For example, a lending protocol could use the *balanceOf()* result to determine collateral amount, and in this case it could be possible to take uncollateralized loans.

Recommended mitigation

In the functions above, account for a potential *rebase()*.

Team response

There is some developer risk and patterns that should be respected. E.g. our own TenderSwap will always transfer tokens before making calculations which ensures the rebase takes place. However when just querying on-chain data these values could indeed be stale. However the same issue exists with non-rebasing tokens, whereby then the exchange rate isn't updated.

Low severity findings

TRST-L-1 A single compromised adapter would enable complete shutdown of all tenderizers

- **Category:** Controlled-delegatecall flaws
- **Source:** Tenderizer.sol
- **Status:** Acknowledged

Description

The Tenderizer operates by performing delegate calls to the appropriate adapter resolved at runtime. The following functions are publicly accessible:

```
function _previewDeposit(uint256 assets) public returns (uint256) {
    return
abi.decode(_adapter()._delegatecall(abi.encodeCall(_adapter().preview
Deposit, (assets))), (uint256));
}
function _previewWithdraw(uint256 unlockID) public returns (uint256)
{
    return
abi.decode(_adapter()._delegatecall(abi.encodeCall(_adapter().preview
Withdraw, (unlockID))), (uint256));
}
function _unlockMaturity(uint256 unlockID) public returns (uint256) {
    return
abi.decode(_adapter()._delegatecall(abi.encodeCall(_adapter().unlockM
aturity, (unlockID))), (uint256));
}
```

The adapter is resolved by the immutable registry entry using the clone-level immutable **asset**.

```
function _adapter() internal view returns (Adapter) {
    return Adapter(_registry().adapter(asset()));
}
```

Users should always use the Tenderizer through the factory-generated clone, but an attacker can call the Tenderizer directly and can supply any **asset** in the immutable args. In the immutable clone pattern, all the relevant state lives in the clone, so typically it doesn't matter if the implementation contract's state is arbitrary. But the one exception is when the

implementation contract is destroyed through a *selfdestruct()* call. If that occurs, the clones would be completely dysfunctional.

In order to exploit it, attacker would have to make governance register an adapter that contains a selfdestruct() opcode, or contain upgradeable code that the attacker can change. Since the different staking ecosystems exist in parallel, it is possible that one of them would request to change the logic without the Tenderize governance. Without knowledge that the ecosystem isolation can be broken through this attack vector, it may occur. Also, if the governance approves an adapter for testing purposes which has the mentioned logic, it could also be abused.

Recommended mitigation

The Tenderizer contract should never be called directly. Consider adding a check in every function, that the current *address(this)* is not the base Tenderizer address. The base address can be stored in an immutable variable during the Tenderizer construction.

Team response

There is no possible way to hijack execution of adapters since callers cannot inject an address that is being delegatecalled.

Hence the proposed solution while valid, is an additional but unnecessary safety measure as long as the pattern of not delegatecalling to injected addresses is respected.

If the pattern is respected the implementations can never be selfdestructed.

TRST-L-2 Some Tenderizer ERC20 view functions will be unreadable

- **Category:** Encoding issues
- **Source:** Tenderizer.sol
- **Status:** Fixed

Description

The Tenderizer overrides the *name()* and *symbol()* ERC20 functions as seen below.

```
function name() external view override returns (string memory) {
    return string(abi.encodePacked("tender", ERC20(asset()).symbol(),
" ", validator()));
}
// @inheritdoc TToken
function symbol() external view override returns (string memory) {
    return string(abi.encodePacked("t", ERC20(asset()).symbol(), "_",
validator()));
}
```

Note that *encodePacked()* concatenates what are presumably string elements to produce a readable output. However, the *validator()* function returns an **address** type, which is raw-encoded. Therefore, the functions will not produce readable output.

Recommended mitigation

Consider using a raw-to-hex encoder like the OpenZeppelin Strings library.

Team response

Fixed in a commit at a later state than the audit occurred.

TRST-L-3 The unlockMaturity() function could return wrong results

- **Category:** Validation flaws
- **Source:** GraphAdapter.sol
- **Status:** Acknowledged

Description

In Tenderizers, the *unlockMaturity()* should return the block number in which the given **unlockID** is unlocked.

```
function unlockMaturity(uint256 unlockID) external view override
returns (uint256) {
    Storage storage $ = _loadStorage();
    Unlock memory unlock = $.unlocks[unlockID];
    uint256 THAWING_PERIOD = GRAPH.thawingPeriod();
    // if userEpoch == currentEpoch, it is yet to unlock
    // => unlockBlock + thawingPeriod
    // if userEpoch == currentEpoch - 1, it is processing
    // => unlockBlock
    // if userEpoch < currentEpoch - 1, it has been processed
    // => 0
    uint256 unlockBlock = $.lastEpochUnlockedAt + THAWING_PERIOD;
    if (unlock.epoch == $.currentEpoch) {
        return THAWING_PERIOD + unlockBlock;
    } else if (unlock.epoch == $.currentEpoch - 1) {
        return unlockBlock;
    } else {
        return 0;
    }
}
```

The issue is that the function does not validate that **unlockID** exists. If it doesn't, the **unlock.epoch** value would be zero as mapping values are zero initialized. Then, the value would be wrong depending on the state of **$.currentEpoch**. Integration with external projects could introduce security risks as well as front-end issues.

Recommended mitigation

Require that **unlock.shares** is not zero in *unlockMaturity()*.

Team response

We decided to not have revert statements in functions that are external view functions.

TRST-L-4 An attacker could make the withdrawal period longer than necessary

- **Category:** DOS attacks
- **Source:** GraphAdapter.sol
- **Status:** Acknowledged

Description

When a user unstakes from The Graph Tenderizer, *unstake()* will either:

- Immediately begin unstaking on The Graph staking contract
- If unstaking is currently ongoing, schedule it for the next round

The issue is that an attacker could unstake a near-zero amount infinitely at almost zero cost. This means that honest unstakes would have to wait until the current "fake" withdrawal completes. This is possibly a significant delay.

Recommended mitigation

Consider requiring a minimum unstaking amount to disincentivize clogging of the queue.

Team response

This is true if there isn't any unstake in The Graph Protocol ongoing. We don't anticipate this to occur very often (mostly only the first time there is being unstaked). We further decrease the griefing risk here by still allowing users to earn rewards if they have unstaked but have to wait longer than the Graph's default unbonding period.

TRST-L-5 GraphAdapter rebase() does not take into account delegation transfers from L1

- **Category:** Donation attacks
- **Source:** GraphAdapter.sol
- **Status:** Fixed

Description

It seems to be a hard assumption that all delegations and undelegations in the GraphAdapter must be initiated by the Tenderizer. However, it has been observed that anyone could bridge Graph delegations from L1 using the *transferDelegationToL2()* function of the staking contract, and provide an arbitrary L2 beneficiary address.

```
function transferDelegationToL2(
    address _indexer,
    address _l2Beneficiary,
    uint256 _maxGas,
    uint256 _gasPriceBid,
    uint256 _maxSubmissionCost
) external payable override notPartialPaused {
```

The early code in *rebase()* assumes the delegation shares has not changed since the last calculation, so if the **_tokensPerShare** hasn't changed, it returns early.

```
uint256 _tokensPerShare = delPool.tokens * 1 ether / delPool.shares;
// Account for rounding error of -1 or +1
// This occurs due to a slight change in ratio because of new
delegations or withdrawals,
// rather than an effective reward or loss
if (
    (_tokensPerShare >= $.tokensPerShare && _tokensPerShare -
$.tokensPerShare <= 1)
```

```
        || (_tokensPerShare < $.tokensPerShare && $.tokensPerShare -
_tokensPerShare <= 1)
) {
    return currentStake;
}
```

However, it would be correct to still perform the rebasing logic, as there is more value registered for the TToken. In this scenario, it seems benign and attacker wouldn't gain from the donation. However, it should be deeply examined if the assumption can lead to a severe impact in an unforeseen way.

Recommended mitigation

Consider not returning early in *rebase()*. Also, consider the effect of donation on the unstaking logic.

Team response

Fixed.

Mitigation review

The code has been refactored to always fetch the updated token and shares count. Therefore there is no longer an early return possible in *rebase()*.

TRST-L-6 Fetching the lock metadata could revert with Graph tenderizers

- **Category:** Integer overflow issues
- **Source:** GraphAdapter.sol
- **Status:** Acknowledged

Description

Users can query the state of their locks with *getMetadata()*:

```
function getMetadata(uint256 tokenId) external view returns
(Metadata memory metadata) {
    (address payable tenderizer, uint96 unlockId) =
_decodeTokenId(tokenId);
    address asset = Tenderizer(tenderizer).asset();
    Adapter adapter = Tenderizer(tenderizer).adapter();
    uint256 maturity =
Tenderizer(tenderizer).unlockMaturity(unlockId);
    uint256 currentTime = adapter.currentTime();
    return Metadata({
        amount: Tenderizer(tenderizer).previewWithdraw(unlockId),
        maturity: maturity,
        progress: maturity > currentTime
            ? 100 - FixedPointMathLib.mulDivUp((maturity -
currentTime), 100, adapter.unlockTime())
            : 100,
        unlockId: unlockId,
        symbol: ERC20(asset).symbol(),
        name: ERC20(asset).name(),
        validator: Tenderizer(tenderizer).validator()
```

```
    });
}
```

the progress field tracks how close **currentTime** is to **maturity** as a percentage of the *unlockTime()*.

Note that if **maturity – currentTime** is greater than *unlockTime()*, the function would calculate **100 – x**, where x > 100, which would revert in uint256 data types as used here.

In fact this is possible for Graph unlocks. The *unlockTime()* is defined below:

```
function unlockTime() external view override returns (uint256) {
    return GRAPH_STAKING.thawingPeriod();
}
```

The Graph *thawingPeriod()* is defined as the amount of blocks to wait before scheduling undelegation and its execution.

The **maturity** value is calculated below:

```
function unlockMaturity(uint256 unlockID) external view override
returns (uint256) {
    Storage storage $ = _loadStorage();
    Unlock memory unlock = $.unlocks[unlockID];
    uint256 THAWING_PERIOD = GRAPH_STAKING.thawingPeriod();
    // if userEpoch == currentEpoch, it is yet to unlock
    // => unlockBlock + thawingPeriod
    // if userEpoch == currentEpoch - 1, it is processing
    // => unlockBlock
    // if userEpoch < currentEpoch - 1, it has been processed
    // => 0
    uint256 unlockBlock = $.lastEpochUnlockedAt + THAWING_PERIOD;
    if (unlock.epoch == $.currentEpoch) {
        return THAWING_PERIOD + unlockBlock;
    } else if (unlock.epoch == $.currentEpoch - 1) {
        return unlockBlock;
    } else {
        return 0;
    }
}
```

The GraphAdapter has a queueing mechanism – if an undelegation already began, a new undelegation will only begin after the current one is fulfilled. Therefore, **maturity** can be up to **TWO** thawing periods into the future.

Therefore, a queued **unlockID** will cause *getMetadata()* to revert.

Recommended mitigation

The *unlockTime()* should take the pessimistic view and be equal to two thawing periods.

Team response

This contract isn't upgradeable, nothing we can do but handle it in the renderer if we ever decide to use that in the future for displaying these unlock NFTs.

Additional recommendations

TRST-R-1 Call all parent initializers in an upgradable contract

The Registry inherits from UUPSUpgradable and AccessControlUpgradeable, however it only calls the latter's initialization function.

```
function initialize(address _tenderizer, address _unlocks) public
initializer {
    __AccessControl_init();
    _grantRole(UPGRADE_ROLE, msg.sender);
    _grantRole(GOVERNANCE_ROLE, msg.sender);
```

Both initializers are no-ops, but as a best practice consider initializing all parents.

TRST-R-2 Improve visibility for the AccessControl module

The Registry makes use of several roles. For transparency, there is much to benefit from having the privileged addresses accessible through on-chain queries. Consider using the AccessControlEnumerable module instead.

TRST-R-3 unlockMaturity() is not always accurate

The *unlockMaturity()* function may not return an accurate maturity block time. In case the unlock hasn't started processing yet, the value returned is the last processed block time + **2 * THAWING_PERIOD**. This would only be true if at the block when the previous withdrawals finish processing, the next unstaking begins. Realistically, this would be a best-effort lower bound. It's important to document this behavior, so that potential integrations with the contract will not run into issues.

TRST-R-4 The EIP712Domain is non-standard

The TToken **DOMAIN_SEPARATOR** is defined as:

```
function DOMAIN_SEPARATOR() public view virtual returns (bytes32) {
    return keccak256(
        abi.encode(
            keccak256("EIP712Domain(string version,uint256
chainId,address verifyingContract)"),
            keccak256("1"),
            block.chainid,
            address(this)
        )
    );
}
```

A classic EIP-712 domain separator contains one more field, **string name**. Consider adding it for better standardization.

TRST-R-5 Handle deployment addresses with caution

The **GRAPH**, **GRAPH_EPOCHS**, **GRT** addresses in the GraphAdapter are defined using the L1 addresses. Note that it is planned for the adapter to be deployed on L2. It is recommended to define the addresses in a way that is not error-prone, for example deliver those using a deployment script which is separated per chain.

TRST-R-6 Redundant code path in GraphAdapter

The *rebase()* function in GraphAdapter handles the case where the new staked amount is lower, i.e. slashing has occurred. However, in The Graph delegators cannot be slashed (only indexers can), so there is no use in this part of the code.

TRST-R-7 Transfer behavior is not ERC20-compliant

In TToken transfers, the input amount is converted to shares, which are transferred from the **from** address to the **to** address. Due to rounding errors, in practice **amount** represents the upper bound for the transfer. If rounding occurs, the ERC20 behavior is not followed, as it states that exactly **amount** must be transferred (and **amount** is emitted in the event anyhow). It is recommended to clearly document this behavior. There may be cases where integrations would fail as many dApps assume that when a transfer is executed, the **to** address will have at least **amount** transferred in.

Centralization risks

TRST-CR-1 Privileged roles may cause harmful effects on the protocol and users

The project defines the following roles:

- UPGRADE_ROLE – The role may upgrade the Registry. The Registry is hard coded to all Tenderizers, and change of the Registry logic could lead to theft of all funds stored in the contract, by replacing the **adapter** for the Tenderizer.
- FEE_GAUGE_ROLE – May set protocol fee up to 0.5% defined in the Tenderizer contract
- GOVERNANCE_ROLE – Able to register or overwrite any adapter. The adapter determines critical logic of the Tenderize, specifically handling of staked assets. It can also register factories, which generate trusted Tenderizer contracts.

TRST-CR-2 Token URI could be replaced by malicious Renderer owner

The Renderer is implemented as an upgradeable contract. Therefore, when resolving the *tokenURI()* of the Unlocks NFT, a malicious owner could make it display any information. This is not actually harmful because the important properties of the Unlock are maintained.

Systemic risks

TRST-SR-1 Fee uptake might be excessive due to lack of high watermark

In many investment and staking schemes, fees are not charged on fluctuating positions, but rather on any price increase over the all-time-high. If for slashing or any other reason, the total supply of the Tenderizer is often reduced, this could lead to higher fees than expected.

TRST-SR-2 External integration risks

Tenderizer is a liquid staking layer which depends on external integrations to drive the staking logic. Naturally, any bugs or malicious behavior on the external side could lead to lose of up to the staked amount. Specifically for the Graph, the intended behavior of the Graph staking contract is that delegators would not be slashed, so in theory the worst case would be loss of opportunity cost of the use of capital.