Treehouse Labs

tETH Updates #3

Security Assessment Report

Version: 2.1

March, 2025

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Treehouse Labs smart contract updates in scope. The review focused solely on the security aspects of the Solidity implementation of the contracts, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the components in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Treehouse Labs smart contract updates contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Treehouse Labs smart contracts in scope.

Overview

Treehouse Protocol is a restaking service with the liquid restaking token `tETH` as its premier product.

Built on top of other Liquid Staking Tokens, such as Lido's `stETH`, it aims to leverage opportunities, such as Aave lending markets and SparkLend, to outperform other LSTs and maximise staking yields delivered to end users.

This review focused on recent changes made to the Treehouse `tETH` contracts, focusing primarily on strategy actions for AaveV3 and SparkLend as well as the accounting system behind these strategies for `tETH`.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the Treehouse tETH repository.

The scope of this time-boxed review was strictly limited to changes made between commits 2ba527b8 and a148603.

Retesting was performed on:

1. Commit 10ef7db
2. Commit 220cdda
3. Pull 19

Note: third party libraries and dependencies were excluded from the scope of this assessment.

Approach

The security assessment covered components written in Solidity.

The manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

To support the Solidity components of the review, the testing team also utilised the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`
- Aderyn: `https://github.com/Cyfrin/aderyn`

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 9 issues during this assessment. Categorised by their severity:

- Critical: 1 issue.
- Low: 6 issues.
- Informational: 2 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Treehouse Labs smart contract changes in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
TREE3-01	Incorrect `nextWindow` Calculation Prevents Timely Accounting Calls	Critical	Resolved
TREE3-02	Modules Can Be Attached Before Being Registered	Low	Closed
TREE3-03	Net Asset Value Varies Based On Strategy Used	Low	Resolved
TREE3-04	`doAccounting` May Revert Due To Gas Usage	Low	Closed
TREE3-05	Module IDs Can Collide	Low	Closed
TREE3-06	Incorrect Variable Checked	Low	Resolved
TREE3-07	Redundant Boundary Checks	Low	Closed
TREE3-08	Asset IDs Differ Between SparkLend And Aave	Informational	Closed
TREE3-09	Miscellaneous General Comments	Informational	Closed

TREE3-01	Incorrect `nextWindow` Calculation Prevents Timely Accounting Calls		
Asset	`PnlAccounting.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The `doAccounting()` function in the `PnlAccounting` contract incorrectly updates the `nextWindow` variable using `+=`, causing the cooldown period to accumulate with the current `block.timestamp`. This leads to an unintended extension of the waiting period after the first call, making subsequent calls impossible within a reasonable timeframe, contrary to the intended 1-hour cooldown.

The `PnlAccounting` contract is responsible for marking the protocol's profit and loss (PnL) by calculating the Net Asset Value (NAV) and applying it to the accounting system. The `doAccounting()` function is designed to be called periodically by an authorised executor or owner to update the protocol's NAV, with a cooldown period enforced via the `cooldown` variable (defaulting to 3600 seconds, or 1 hour).

The following logic is used to enforce the cooldown:

PnlAccounting.sol

```
if (block.timestamp < nextWindow) revert StillInWaitingPeriod();
nextWindow += (uint64(block.timestamp) + cooldown);
```

The current implementation uses the line `nextWindow += (uint64(block.timestamp) + cooldown);` to set the next allowed timestamp. On the first call, this works as expected: if `block.timestamp` is, for example, 1000, and `cooldown` is 3600, `nextWindow` becomes 4600, enforcing a 1-hour wait from that point. However, on the second call (e.g., at `block.timestamp = 5000`), instead of setting `nextWindow` to 8600 (5000 + 3600), it adds to the previous `nextWindow` (4600 + 5000 + 3600 = 14200). This accumulation continues with each subsequent call, pushing `nextWindow` further into the future and effectively preventing timely calls after the initial one.

Recommendations

Replace the `+=` operator with `=` to reset nextWindow based on the current `block.timestamp` plus the cooldown. This ensures a consistent 1-hour cooldown between calls.

Resolution

The issue was resolved in commit 10ef7db.

TREE3-02	Modules Can Be Attached Before Being Registered		
Asset	`NavRegistry.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

It is possible to call NAV calculations on non-existent modules, which can lead to recorded assets being lower than anticipated.

When attaching a module, there is no check that a module has already been registered. This can enable calls to modules that do not exist, which can result in Net Asset Value calculations returning zero.

When attaching a module to a strategy using `attachTo()`, the only check carried out is if the strategy and module are already attached, as seen below:

```
function attachTo(address strategy, ModuleParams calldata params) external onlyOwner {
  if (_strategyModuleIds[strategy].moduleIds.add(params.moduleId) == false) revert AlreadyAttached(params.moduleId);
  strategyModuleCd[strategy][params.moduleId] = params.cd;
  }
```

This means that it is possible to attach a module that does not exist in the `modules` mapping to a strategy. Later, when attempting to use `getStrategyNav()`, the `strategy` address from `modules[bytes4(moduleIds[i])].addr)` will return the zero address and the static call will be sent to it, but importantly, it will succeed without reverting the call. This can cause calculations of the Net Asset Value to return zero and will lead to the system reporting a large loss.

While this issue can have a high impact to end users, it is unlikely to happen in practice as it requires a sequence of irregular actions to occur first. Assets would have to be added with one module, have their value recorded in the NAV and then have the project shift to a new module that had not been set up correctly using `registerModule()`, before any incorrect valuation would occur with the NAV.

Recommendations

When attaching a module to a strategy, the function `attachTo()` verify that the module being attached has been already registered. This can be done by checking if the module exists using the `isModuleRegistered()` function.

Resolution

The issue was acknowledged by the project team with the following comment:

> *"Whilst attaching a new NAV module without registering it is valid, in practice the NAV won't be changed because the address called using staticCall is retrieved from the modules mapping in getNav . Calling staticCall on the zero address will not revert, but the result will be 0."*

TREE3-03	Net Asset Value Varies Based On Strategy Used		
Asset	`PnlAccounting.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

SparkLend and AaveV3 rely on different oracle systems, meaning they can report a different Net Asset Value, which can lead to the system deducting a performance fee from depositors despite no material change having occurred.

This can happen as SparkLend, while forked from the AaveV3 codebase, makes use of their own oracle system that leverages Chainlink, Redstone and Chronicle oracles to calculate a median price. AaveV3 on the other hand, uses Chainlink with a fallback oracle in the event the Chainlink oracle fails. This means that the two systems can report different prices for assets and liabilities held by their market.

When the oracles return different prices, it is possible for the system to record a profit or loss simply by shifting assets from AaveV3 to SparkLend, or vice versa. Note that as the owner controls system accounting, the update related to this should only occur once every hour at most, with the default `cooldown` time set.

In the event a profit is realised, then the protocol shall also deduct a management fee from the recorded profit. Note that this issue is unlikely to occur in practice due to the cooldown period mentioned before. This means that `doAccounting()` cannot be called twice in the same block. While the effect is technically still present over larger timescales, its scale of impact is likely to be less than other external events, such as collateral price change and staking profits accrual.

Recommendations

Adjust respective strategy totals by their own oracles to normalise assets.

Specifically, within `NavAaveV3.nav()`, the line [36]:

```
_nav = (navInBase * 1e10 * PRECISION) / RATE_PROVIDER_REGISTRY.getEthInUsd();
```

should make use of the same oracle as the protocol being integrated.

Resolution

The issue was resolved in pull 19 and the following comment was provided by the project team:

> *"Calculate NAV of spark strategy using individual assets, which we will price using fundamental oracles (i.e wrapping and unwrapping exchange rates), instead of just using the NAV number that we can query from them."*

TREE3-04	`doAccounting` May Revert Due To Gas Usage		
Asset	`PnlAccounting.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The main accounting function relies on `NavRegistry.getStrategyNav()`, which contains code copying storage values to memory in bulk, which may cause the parent function call to revert due to excessive gas usage.

`getStrategyNav()` makes use of OpenZeppelin's `EnumerableSet` library and, in particular, uses the method `values()` to retrieve an entire set from storage.

Using `values()` in this manner is intended only for `view` functions, as this can be a very gas intensive operation and can lead to the calling function reverting.

While `getStrategyNav()` is a view function, `PnlAccounting.doAccounting()` that calls it is not, and will incur the full gas cost associated with retrieving these sets.

This effect is compounded by the intermediate function `getProtocolNav()` which needs to call `getStrategyNav()` for all registered strategies.

Currently, the number of planned strategies and modules is low, and so is the likelihood of this becoming an issue. However, as the system expands, the risk will grow. If `doAccounting()` becomes uncallable, then the system becomes stuck, unable to realise a profit or loss and no management fees would be collected from the protocol.

Recommendations

This issue can be solved by multiple approaches:

- Ensure the worst-case gas cost of calling `doAccounting()` is carefully considered when adding new strategies and modules to avoid exceeding the gas limit.
- Enable multiple calls for processing `doAccounting()` to mitigate the risk of running out of gas. However, this approach would require a significant rewrite of `doAccounting()` and the storage of intermediate totals between calls, making it less desirable.
- Make use of a system similar to the strategy pausing system in `StrategyStorage.sol` to reduce the number of calls to `getStrategyNav()` made in `getProtocolNav()`. This way, if a strategy is no longer in use, or only contains dust amounts of value, it can be safely ignored in accounting functions.
- Given `doAccounting()` can only be called by the `Owner` or `Executor` roles, they could be trusted to provide the list of strategies and modules to be called, negating the need to call `values()` in `getStrategyNav()`.

Resolution

The issue was acknowledged by the project team.

TREE3-05	Module IDs Can Collide		
Asset	`NavRegistry.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Module IDs are 4-byte values used to identify different actions for each strategy, and because these are short hashes, it is possible for these values to be identical for different module names. This would prevent the addition of new, valid modules.

Each module ID is generated from the keccak256 hash of the module name, shortened to `bytes4` as seen below in the code snippet from `ignition/modules/utils.ts` :

utils.ts

```
export const getActionIdFromName = (name: string) => ethers.solidityPackedKeccak256([`string`], [name]).slice(0, 10)
```

While `NavRegister.registerModule()` features protection against registering multiple modules with the same ID, this issue would force the development team to alter their naming strategy, which could make the process to determine a module ID less clear.

Recommendations

Make use of the full `bytes32` keccak256 hash for a module ID. With the full 32 bytes, a collision is functionally impossible.

Alternatively, the development team could add checks to the module registration offchain process to prompt modifying clashing IDs if they are generated. This approach may lead to less ideal action names, but has the benefit of not needing changes to the onchain codebase.

Resolution

The issue was acknowledged by the project team with the following comment:

> *"Module names will be checked for collisions, and this is improbable in practice."*

TREE3-06	Incorrect Variable Checked		
Asset	`periphery/PnlAccounting.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

When setting `_newDeviation` via `setDeviation()`, there is a check on line [**132**] to ensure the set value does not exceed `PRECISION`. However, the value checked is actually the existing storage value `deviation`. The impact of this is low as both `deviation` and `_newDeviation` are `uint16` values and so cannot currently exceed the `PRECISION` value.

If `PRECISION` was reduced, or the values had a larger `uint` type, this issue could cause further problems as the function `setDeviation()` would allow an oversized `_newDeviation` to be set. Then once set, any attempt to reduce this value by calling `setDeviation()` again would revert due to the current `deviation` value triggering this check's revert.

Recommendations

Correct line [**132**] to check `_newDeviation`, not `deviation`.

Resolution

The issue was resolved in commit 220cdda.

TREE3-07	Redundant Boundary Checks		
Asset	`periphery/PnlAccounting.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Certain parameter boundary checks exceed the maximum value allowed by the parameter's type, meaning they will never be triggered. This also creates a misleading impression of the acceptable value range, as the actual maximum is often much smaller than the intended constraint.

PnlAccounting.sol

```
/**
* @notice Set a cooldown for accounting
* @param _newCooldownInSeconds new cooldown in seconds
*/
function setCooldownSeconds(uint16 _newCooldownInSeconds) external onlyOwner {
 if (_newCooldownInSeconds < 60 || _newCooldownInSeconds > 2 days) revert InvalidCooldown();
 emit CooldownUpdated(_newCooldownInSeconds, cooldown);
 cooldown = _newCooldownInSeconds;
}
```

Noted examples are:

- `_newCooldownInSeconds`, which is a `uint16` and can only have a maximum value of `2^16 - 1` (65,535). On line [**113**], there is a check if this value is less than 2 days, which is 172,800 seconds. Currently, the maximum accepted value (65,535), equates to approximately 18 hours.
- `_newDeviation`, which is also a `uint16`, on line [**132**] it appears (see TREE3-06) that the check is intended to ensure that this value is less than 1,000,000, but again, the maximum value it can be set to is 65,535, meaning that the actual value range is only about 6% of the intended `[0, 1,000,000]` range size.

Recommendations

Either reduce the parameter constraints mentioned so they are validated meaningfully, or alter the type of the parameters to allow larger inputs.

`uint24` would natively be able to store values as large as 16,000,000, which should be sufficient for the parameters mentioned.

Resolution

The issue was acknowledged by the project team.

TREE3-08	Asset IDs Differ Between SparkLend And Aave
Asset	`strategy/actions/spark/*`
Status	**Closed:** See Resolution
Rating	Informational

Description

Strategy actions for AaveV3 and SparkLend make use of an `_assetId` which is then used to fetch the token address using `IPoolV3(_lendingPool).getReserveAddressById(_assetId)`. However, `_assetIds` resolve to different tokens for the SparkLend `_lendingPool` contract, for example:

- `_assetId` 1 on AaveV3 lendingPool is `0x7f39C581F595B53c5cb19bD0b3f8dA6c935E2Ca0`, which is the address of `wstETH`.
- `_assetId` 1 on SparkLend lendingPool is `0x83F20F44975D03b1b09e64809B757c47f942BEeA` which is the address of `sDAI`.

None of the supported asset IDs for SparkLend overlap with the accepted protocol tokens. As a result, the impact of this issue is purely informational - if an incorrect asset ID for a different platform is provided, the call will revert, since, for example, the Treehouse protocol should not have any sDAI under its management.

Recommendations

Be aware of the need for different asset IDs when working with different strategy platforms.

Resolution

The issue was acknowledged by the project team.

TREE3-09	Miscellaneous General Comments
Asset	All contracts
Status	**Closed:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Different Lending Parameters On Different Markets**
   Related Asset(s): N/A
   While SparkLend is forked from the AaveV3 codebase, they differ on the Loan-to-Value and liquidation percentage parameters used in normal and efficiency mode markets. Likewise, AaveV3 Prime Instance is also listed in some test scripts and has different safety parameters.
   Ensure the development team is aware of these different values and avoids using the same parameters and leverage for both markets, as this may lead to riskier positions than intended.
2. **Missing Zero Address Checks**
   Related Asset(s): PnlAccounting.sol, NavRegistry.sol
   The `updateExecutor()` function makes no check that `_newExecutor` is not set to the zero address.
   Unless setting `_newExecutor` as the zero address is expected, a zero address check should be included in `updateExecutor()`.
   The same applies to `registerModule()` and `updateModule()` in `NavRegistry.sol`
3. **Equality To Booleans**
   Related Asset(s): NavRegistry.sol
   On line [**165**], line [**175**], line [**185**] and line [**209**], there is a check if a condition is equal to a `boolean`. These checks can be done directly using the boolean result itself.
   Check the `boolean` values directly to avoid redundant equality checks.
4. **Event Emission**
   Related Asset(s): NavRegistry.sol
   Consider emitting events in critical setters: `attachTo()`, `detachFrom()` and `updateParams()`.
5. **Magic Numbers**
   Related Asset(s): periphery/NavLens.sol, modules/nav/NavAaveV3.sol
   On line [**55**] of `NavLens.sol`, the module ID is stored directly as the hardcoded hash value. This makes it difficult to determine if the supplied value is correct and reduces code readability. Similarly, in `NavAaveV3.sol` on line [**36**], there is use of the constant `1e10` to scale the calculation.
   The ID should be replaced with a named constant where the derivation of this hash is clear. The magic number should also become a named constant to improve readability.
6. **Length Validation**
   Related Asset(s): periphery/NavLens.sol
   Consider validating that length of `_stratLen` is equal to `dynamicModuleParams`.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The above issues were acknowledged by the project team.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are given along with this document. Due to the time constraints tests illustrating issues raised in the report were prioritised.

The `forge` framework was used to perform these tests and the output is given below.

```
Ran 1 test for test/tests-fork/NavRegistry.t.sol:test_NavRegistry
[FAIL. Reason: next call did not revert as expected] test_attachModule_vuln() (gas: 115162)
Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 428.32ms (149.01µs CPU time)

Ran 3 tests for test/tests-fork/PnlAccounting.t.sol:PnlAccountingTest
[PASS] test_bytes32_bytes() (gas: 10448)
[FAIL. Reason: treasury profit should not change: 0 != 37893] test_different_oracle_prices_vuln() (gas: 1807722)
[PASS] test_initialBalance() (gas: 8372)
Suite result: FAILED. 2 passed; 1 failed; 0 skipped; finished in 439.01ms (11.97ms CPU time)

Ran 2 test suites in 442.47ms (1.30s CPU time): 2 tests passed, 2 failed, 0 skipped (4 total tests)
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

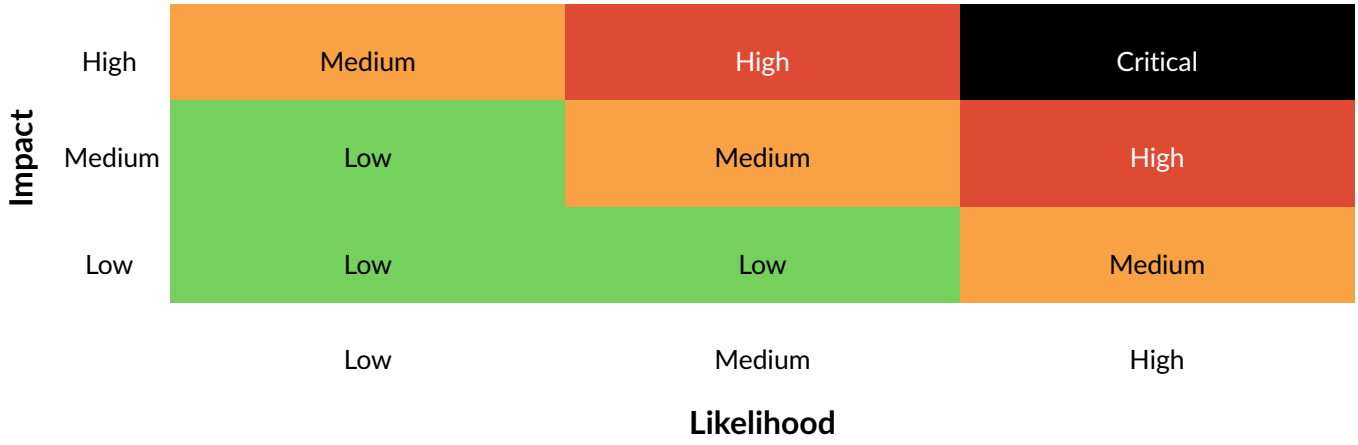

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].