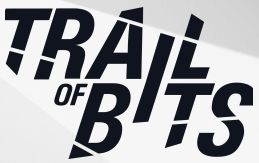

Treehouse tETH

Security Assessment

October 3, 2024

Prepared for:
Ben Loh
Treehouse Finance

Prepared by: **Alexander Remie**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineering directors were associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Alexander Remie, Consultant
alexander.remie@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
September 17, 2024	Report readout meeting
October 3, 2024	Delivery of comprehensive report

Executive Summary

Engagement Overview

Treehouse engaged Trail of Bits to review the security of its tETH contracts. tETH is a liquid restaking token that serves to converge the fragmented on-chain ETH interest rates market. Holders of tETH earn yield through interest rate arbitrage while still being able to use tETH for DeFi activities.

This assessment is a continuation of a previous assessment conducted by Trail of Bits in July 2024. A single consultant from the blockchain team conducted a review focusing on the smart contracts from September 3 to September 13, 2024, for a total of two engineer-weeks of effort. Our testing efforts focused the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target systems. We conducted this audit with full knowledge of the system. With full access to source code and documentation, we performed static and dynamic testing of the smart contracts, using automated and manual processes.

Observations and Impact

The tETH smart contracts heavily rely on privileged actors to manually perform necessary operations; these include operations related to PnL distribution, funding the redemption contract to enable user withdrawals, updates to state variables that directly impact users' solvency and funds, and investments into and divestments from strategies.

Additionally, we identified one issue that allows anyone to steal `wstEth` that was accidentally transferred to the `TreehouseRouter` contract. This is because a function in the contract returns the wrong amount to indicate the amount of `wstEth` that belongs to the caller when making a deposit (TOB-TETH-1). We also highlighted one dangerous use of a storage variable in a contract that uses delegatecalls; this issue could lead to loss of funds when new actions are implemented and enabled in the future (TOB-TETH-5).

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that the Treehouse team take the following steps prior to achieving deployment:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Identify all system properties that are expected to hold, and use dynamic end-to-end fuzz testing to validate those system properties.**

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	2
Low	0
Informational	4
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Configuration	1
Data Validation	2
Undefined Behavior	3

Project Goals

The engagement was scoped to provide a security assessment of the tETH protocol. Specifically, we sought to answer the following non-exhaustive list of questions:

- Does the use of delegatecall for executing actions pose a (future) risk to the system?
- Is the conversion between the different currencies implemented correctly?
- Is the vault implemented as a single-asset vault?
- Can IAU tokens be transferred freely?
- Are all functions protected by adequate access controls?
- Are there missing events?
- Are all function inputs validated?
- Can someone steal tokens from the protocol?
- Is the parameter-replace mechanism implemented correctly?
- Can the deposit cap be circumvented?
- Are sufficient protections present to prevent tokens in the protocol's accounting from being rescued?

Project Targets

The engagement involved a review and testing of the targets listed below.

tETH protocol

Repository	https://github.com/0xhypn/tETH-protocol
Version	`60c4c39c800d280057fefe2ce1945f61c0dc795d`
Type	Solidity
Platform	EVM

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following contracts:

- **`TreehouseRouter`**: We used manual review to look for flaws in the deposit mechanism; flaws in the functions that convert between the ETH, stETH, and wstETH currencies; missing pausing-related modifiers; missing or incorrect access controls; unsafe ERC20 transfer functions; incorrect setup of the contract during deployment (i.e., the constructor); ways to circumvent the "allowable assets" mechanism; ways to circumvent the deposit cap; and ways to steal the infinite `InternalAccountingUnit` and `stETH` token approvals.

- **`TAsset`**: We manually reviewed the implementation to look for missing events, missing or incorrect input validation, donation attack vectors, flaws in the overriding of ERC20 functions, incorrect access controls, and incorrect contract construction during deployment.

- **`TreehouseRedemption`**: We manually reviewed the contract to look for missing or incorrect access controls, flaws in the token transfers, incorrect state updates, unexpected overflows leading to a revert, missing events, missing or insufficient validation of input data, missing pausing-related modifiers, reentrancy vulnerabilities, and ways to be able to extract more tokens than should be possible.

- **`TreeHouseAccounting`**: We used manual review to look for missing events, missing or incorrect access controls, missing or insufficient validation of input data, incorrect interactions with the `InternalAccountingUnit` contract and `TAsset` contracts, and ways to steal the infinite `InternalAccountingUnit` token approval.

- **`InternalAccountingUnit`**. We used manual review to look for incorrect access controls and flaws in the overridden ERC20 functions `_update` and `_checkOwner`.

- **`strategy/` contracts:** We manually reviewed the implementation of all of the various action contracts to look for flaws related to input validation, incorrect argument/return value passing, flaws in the logic, and flaws in the integration with external protocols (Lido and Aave). Regarding contracts that are not a specific action, we assessed whether the use of a (double) delegatecall could be used to steal tokens or otherwise cause problems for the tETH protocol, missing or incorrect access controls, missing events, missing pausing-related modifiers, flaws in the inline assembly that is used to perform delegatecalls, missing validation in the action registry system used for adding/replacing action contracts, flaws in the replaceable-parameter mechanism, and flaws caused by the action-contract-id mechanism.

- **`rate-providers/` contracts:** We manually reviewed these contracts to look for missing events, missing or incorrect access controls, flaws in the conversion of different rates, and whether it is possible for users to register their own rate provider.
- **`periphery/` contracts:** We used manual review to look for missing events, missing or incorrect access controls, reentrancy vulnerabilities, unsafe ERC20 transfer functions, flawed in the internal accounting, ways to make the system allow unsupported tokens, and flaws in the handling of prices with different numbers of decimals.
- **`libs/` contracts:** We used manual review to assess whether the assigned contract owner can rescue tokens that should not be rescuable. We looked for missing or incorrect access controls and missing events. We looked for ways that a denylisted address can circumvent this restriction.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- Issues found in previous audits were considered out of scope.
- The offchain component(s) of this system were considered out of scope.
- We did not review the high-level economic incentives and disincentives imposed by the system.
- We did not look for front-running vulnerabilities.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The protocol uses Solidity 0.8.24, which has overflow protection by default for arithmetic operations. Most of the operations are documented with inline documentation. Asset calculations rely on the assumption that the dollar value of stETH will always be equal to ETH; this could introduce accounting issues if stETH depegs (TOB-TETH-3). Additionally, we found an issue where the wrong numerical value is returned, allowing any accidentally transferred wstETH to be stolen by any account through the router contract (TOB-TETH-1).	**Satisfactory**
Auditing	All functions involved in critical state-changing operations emit events. The codebase uses a wide variety of informative events and error messages, which are emitted at appropriate locations. We did identify one issue where an event may contain duplicate values (TOB-TETH-6).	**Satisfactory**
Authentication / Access Controls	Most functions within the contracts are restricted by access controls that permit only privileged actors to execute them. Users have limited control, primarily restricted to deposits, creating and canceling redemption requests, and finalizing redemptions. Given the presence of multiple privileged actors performing different roles, it would be beneficial to document these roles and the actions they are authorized to perform.	**Moderate**
Complexity Management	The smart contract codebase contains a significant number of contracts; however, they are easy enough to reason through, and most of the complexity is left to be handled by the strategy manager via the off-chain	**Satisfactory**

	components. Each contract in the protocol has a clear purpose, and there are no signs of excessive inheritance or high cyclomatic complexity. All functions are concise and well documented, have a clear purpose, and are appropriately tested. Unlike what the Treehouse team expressed, the Vault is not a single-asset vault. This does not pose an immediate security risk but does increase the complexity of the Vault (TOB-TETH-4).	
Decentralization	The system's operations depend on certain privileged actors manually executing essential tasks (via off-chain executions). These tasks include operations related to profit and loss distribution, user withdrawals, updating state variables affecting user solvency and funds, and managing investments and divestments. Due to the extensible nature of the portfolio management system, privileged actors can perform arbitrary actions. Additionally, the `Rescuable` contract includes a provision that allows the retrieval of any uninvested funds from the Vault (TOB-TETH-2).	**Weak**
Documentation	The code is generally well commented using NatSpec style. The supplied documentation regarding the system design, architecture, and descriptions of the onchain and offchain components are generally sufficient.	**Strong**
Low-Level Manipulation	The use of double delegatecall in the strategy flow raises concerns, as it could potentially lead to unintended consequences, such as inadvertently corrupting storage when more advanced strategies are developed (TOB-TETH-5). It is advisable to establish guidelines for writing delegatecalls to prevent such issues. Additionally, the use of assembly is currently limited to executing delegatecalls.	**Moderate**
Testing and Verification	The smart contract codebase contains several unit and integration tests. These tests appear to cover the protocol's most common use cases and test a fair number of potential reverts or other scenarios outside of the "happy path." However, there is no targeted fuzz testing of arithmetic	**Moderate**

	operations, invariants, or function properties. Furthermore, there is no mutation testing. These methodologies can expose unforeseen edge cases or anomalies that regular testing might miss. Fuzzing involves testing with random data inputs to trigger unhandled exceptions or crashes, while mutation testing, a method of code quality validation, alters the software code in small ways to assess whether the test cases can distinguish the original code from the mutated one. Implementing these methodologies can help ensure the resistance of the application against potential unusual inputs or behaviors.	
Transaction Ordering	Yield aggregator protocols in general are vulnerable to front-running issues, especially during profit harvesting; the codebase should undergo a more in-depth review to find these vulnerabilities.	**Further Investigation Required**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Anyone can steal wstETH tokens accidentally transferred to the TreehouseRouter contract	Undefined Behavior	**Medium**
2	Underlying tokens can be “rescued”	Data Validation	**Medium**
3	WstEth.wrap expects stETH amount instead of ETH amount	Undefined Behavior	**Informational**
4	Single-asset vault is not a single-asset vault	Configuration	**Informational**
5	Dangerous storage variable in Strategy contract due to use of delegatecall	Undefined Behavior	**Informational**
6	Missing return value check can lead to incorrect event emission	Data Validation	**Informational**

Detailed Findings

1. Anyone can steal wstETH tokens accidentally transferred to the TreehouseRouter contract

Severity: **Medium**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-TETH-1
Target: `contracts/TreehouseRouter.sol`	

Description

The `_stethToWsteth` function returns the contract's balance of `wstETH` tokens instead of the value returned from the `WstEth.wrap` function. As a result, the returned value includes any `wstETH` that was accidentally transferred to the `TreehouseRouter` contract.

```
169    function _stethToWsteth(uint amount) private returns (uint) {
170      IwstETH(payable(wstETH)).wrap(amount);
171      return IERC20(wstETH).balanceOf(address(this));
172    }
```

Figure 1.1: The `_stethToWsteth` function in `TreehouseRouter.sol`

The `TreehouseRouter.depositEth` function calls the `_ethToWsteth` function. This function first deposits the ETH into the `StEth` (=`Lido`) contract, and then the ETH amount is passed into the `_stethToWsteth` function. The `_stethToWsteth` function will call the `WstEth.wrap` function, which will use `safeTransferFrom` to transfer the `stETH` tokens into the `WstEth` contract, which will calculate and convert that amount of `stETH` tokens to `wstETH` tokens. At the end of the `WstEth.wrap` function, the amount of `wstETH` that was calculated and minted to the caller (=`TreehouseRouter`) is returned.

```
53    function wrap(uint256 _stETHAmount) external returns (uint256) {
54        require(_stETHAmount > 0, "wstETH: can't wrap zero stETH");
55        uint256 wstETHAmount = stETH.getSharesByPooledEth(_stETHAmount);
56        _mint(msg.sender, wstETHAmount);
57        stETH.transferFrom(msg.sender, address(this), _stETHAmount);
58        return wstETHAmount;
59    }
```

Figure 1.2: The `wrap` function in `WstEth.sol`

To deal with tokens that were accidentally transferred to the `TreehouseRouter` contract, the `TreehouseRouter` contract inherits the `Rescuable` contract, which allows a privileged

account to withdraw accidentally transferred tokens. Due to the above-described issue, any account could front-run such calls to sweep any of the accidentally transferred `wstEth` tokens from the `TreehouseRouter` contract.

Exploit Scenario

Alice accidentally transfers `wstETH` tokens directly to the `TreehouseRouter` contract. Eve detects this and calls the `deposit` function immediately after Alice's transaction. Eve's deposit now also includes Alice's accidentally transferred `wstETH` tokens.

Recommendations

Short term, update the `_stethToWsteth` function so that it returns the value returned from the `WstEth.wrap` call. This ensures that no accidentally transferred `wstETH` tokens can be stolen (they can, however, be "rescued" to return them to whoever accidentally transferred them into the `TreehouseRouter` contract).

Long term, when interacting with external protocols, ensure that the integration is correct by verifying that all of the correct input arguments are passed in and that the return arguments are used correctly and not ignored.

2. Underlying tokens can be “rescued”	
Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-TETH-2
Target: `contracts/libs/Rescuable.sol`, `contracts/TreehouseRedemption.sol`, `contracts/Vault.sol`, `contracts/TreehouseRouter.sol`	

Description

Contracts that inherit the `Rescuable` contract do not override its implementation to exclude tokens that are part of the protocol and should not be “rescuable.” This allows the privileged role in charge of “rescuing” tokens to withdraw underlying tokens. Note that this role is privileged (i.e., controlled by the Treehouse team), and this scenario is therefore unlikely to happen.

The `TreehouseRedemption`, `TreehouseRouter`, and `Vault` contracts all inherit the `Rescuable` contract so the Treehouse team can withdraw (i.e., “rescue”) any tokens that accidentally were transferred to these contracts. Although this is a wise safety precaution to prevent people from losing tokens that were transferred incorrectly, if not protected accordingly, this allows a privileged actor to withdraw user assets that were not accidentally transferred to the protocol.

For example, the `TreehouseRedemption` contract is used for redemptions in a two-stage process. After the first stage `TAsset` tokens will be held in the `TreehouseRedemption` contract. After a seven-day delay, the second stage can be executed, which will result in actually redeeming the `TAsset` tokens for the underlying tokens and transferring those to the caller. The `TAsset` tokens should therefore not be rescuable in the TreehouseRedemption contract since that allows them to be “rescued” after stage 1 but before stage 2.

Exploit Scenario

The `rescuer` account in the `TreehouseRedemption` contract is taken over by an attacker. The attacker waits for a call to `redeem` and immediately backruns it by calling `rescueERC20` to steal all of the `TAsset` tokens.

Recommendations

Short term, inside the `TreehouseRedemption`, `TreehouseRouter`, and `Vault` contracts, override the `Rescuable` functions so that, depending on the contract, certain tokens can be made non-rescuable.

Long term, when privileged actors can control tokens owned by protocol users, always account for the possibility that such privileged actors may lose access to their accounts or become rogue. Design and implement the system to remain robust in case this happens so that user assets stay safe.

3. WstEth.wrap expects stETH amount instead of ETH amount

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-TETH-3
Target: `contracts/TreehouseRouter.sol`	

Description

The `_ethToWstEth` function passes in the ETH amount instead of the `stETH` amount into the `_stethToWstEth` function. This could result in an incorrect amount in the unlikely event that `stETH` depegs from ETH.

```
164    function _ethToWsteth(uint amount) private returns (uint) {
165      IstETH(stETH).submit{ value: amount }(address(0));
166      return _stethToWsteth(amount);
167    }
```

Figure 3.1: The `_ethToWstEth` function in `TreehouseRouter.sol`

It would be safer to not rely on the assumption that `stETH` equals ETH.

Recommendations

Short term, update the implementation to pass the `stETH` amount instead of the ETH amount into the `_stethToWsteth` function. Take into account that `WstEth.wrap` expects "`stETH` amount" instead of "`stETH` shares."

Long term, do not rely on assumptions when it is possible to circumvent the reliance on such assumptions. This would result in fewer possible surprises once such assumptions no longer hold, which could cause severe problems for the protocol.

4. Single-asset vault is not a single-asset vault

Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-TETH-4
Target: `contracts/Vault.sol`	

Description

According to the Treehouse team, the `Vault` is a single-asset vault:

> *we went from multiple assets within a single vault, to multiple vaults w/ a single asset.*

However, the current `Vault` is not a single-asset vault, as demonstrated by the presence of functions to control the "allowed assets" in the `Vault` contract.

```
142    function addAllowableAsset(address _asset) external onlyOwner {
143      if (IERC20Metadata(_asset).decimals() > 18) revert UnsupportedDecimals();
144      RATE_PROVIDER_REGISTRY.checkHasRateProvider(_asset);
145
146      bool success = _allowableAssets.add(_asset);
147      if (!success) revert Failed();
148      emit AllowableAssetAdded(_asset);
149    }
```

Figure 4.1: The `addAllowableAsset` function in `Vault.sol`

Although this does not pose an immediate security risk, we do want to highlight this issue, as it complicates the `Vault` compared to an actual "single-asset vault."

Recommendations

Consider converting all assets (for example, in the `TreehouseRouter`) to a single underlying asset, and only allow that asset to be deposited into the `Vault`. This way the `Vault` could be a true "single-asset vault."

5. Dangerous storage variable in Strategy contract due to use of delegatecall

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-TETH-5
Target: `contracts/strategy/Strategy.sol`	

Description

The `Strategy` contract contains a single storage variable (`strategyExecutor`). Because the implementation uses a (double) delegatecall, it takes only one mistake in the enabled `Action` contracts to overwrite this very important storage variable. Currently, none of the `Action` contracts have a storage variable, so this is not currently an issue. However, when adding new Actions in the future, such an action might accidentally contain a storage variable.

```
26    address public strategyExecutor;
```

Figure 4.1: The `strategyExecutor` variable in `Strategy.sol`

The `strategyExecutor` storage variable contains the address of the account that is allowed to initiate action executions. If this variable ever gets overwritten, access to the actions and all of the assets within external protocols could be lost.

The use of delegatecall is generally discouraged due to the security risks it poses. The most important risk is accidentally and incorrectly overwriting storage variables in the initiating contract from within the contracts called through delegatecall. To protect against overwriting variables this way the recommendation is to not have any storage variables (declared in the traditional sense) in the initiating contract. The `Strategy` contract (the initiating contract) however does contain a single storage variable, and it is a very important one that determines who can initiate all of the actions.

None of the actions that currently exist in the repository declare any storage variables. In case a new action is added which does have a storage variable, then writing to this storage variable would result in overwriting the `strategyExecutor` storage variable.

Exploit Scenario

A new action is added that includes a storage variable. Upon execution of this new action the `strategyExecutor` address is overwritten to have a value of zero. None of the actions can now be called again since the account allowed to do so is address zero.

Recommendations

Short term, remove the `strategyExecutor` storage variable from the `Strategy` contract. In its place, declare an `immutable` variable that contains the address of the `StrategyStorage` contract. Within the `StrategyStorage` contract, add functionality to store and retrieve the `strategyExecutor`. Doing this will result in the `Strategy` contract no longer having any storage variables.

Long term, consider rewriting the implementation to use regular calls instead of delegatecalls. This would remove all of the dangers of using delegatecall.

6. Missing return value check can lead to incorrect event emission

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-TETH-6
Target: `contracts/strategy/StrategyStorage.sol`	

Description

The return value of the `EnumerableSet.add` function is not validated. As a result, having duplicates within `_allowedActions` or `_allowedAssets` does not result in a revert. Instead, the call will succeed and emit the `StrategyCreated` event with duplicate values within the `_allowedAssets` and/or `_allowedActions` event fields. This issue has no other effects besides an incorrect event emission.

```
67    function storeStrategy(
68      address _strategy,
69      bytes4[] calldata _allowedActions,
70      address[] calldata _allowedAssets
71    ) external onlyOwner returns (uint _strategyIndex) {
72      if (strategies.add(_strategy) == false) revert AlreadyExist();
73      _strategyIndex = strategies.length() - 1;
74
75      for (uint i; i < _allowedActions.length; ) {
76        parameters[_strategy].whitelistedActions.add(_allowedActions[i]);
77
78        unchecked {
79          ++i;
80        }
81      }
82
83      for (uint i; i < _allowedAssets.length; ) {
84        parameters[_strategy].whitelistedAssets.add(_allowedAssets[i]);
85
86        unchecked {
87          ++i;
88        }
89      }
90
91      parameters[_strategy].isActive = true;
92
93      emit StrategyCreated(_strategyIndex, _allowedAssets, _allowedActions);
94    }
```

Figure 6.1: The `storeStrategy` function in `StrategyStorage.sol`

Figure 6.2 shows that the add function (and the _add function) in the EnumerableSet library returns a Boolean indicating if the addition was successful. If, for example, the addition failed because that value was already in the set, the add function will return false.

```
function add(AddressSet storage set, address value) internal returns (bool) {
    return _add(set._inner, bytes32(uint256(uint160(value))));
}

function _add(Set storage set, bytes32 value) private returns (bool) {
    if (!_contains(set, value)) {
        set._values.push(value);
        // The value is stored at length-1, but we add 1 to all indexes
        // and use 0 as a sentinel value
        set._indexes[value] = set._values.length;
        return true;
    } else {
        return false;
    }
}
```

Figure 6.2: The add and _add functions in OpenZeppelin's EnumerableSet.sol

Recommendations

Short term, validate the return value of the call to EnumerableSet.add. Have it revert if the value is false.

Long term, read the source code of all external libraries/contracts to know how to write correct integrations. This helps to ensure error-free usage of external libraries and contracts.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Configuration	The configuration of system components in accordance with best practices
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Data Handling	The safe handling of user inputs and data processed by the system
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Maintenance	The timely maintenance of system components to mitigate risk
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Recommendations

The following recommendations are not associated with any specific vulnerabilities. However, they will enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **Change the inheritance order to remove unnecessary overriding of `transferOwnership` and `_transferOwnership`**. Several contracts override these two functions, which are originally defined in the `Ownable` and `Ownable2Step` contracts. Changing the inheritance order should eliminate the need to override these two functions.

```
176    function transferOwnership(address newOwner) public virtual
  override(Ownable2Step, Ownable) onlyOwner {
177      super.transferOwnership(newOwner);
178    }
179
180    function _transferOwnership(address newOwner) internal virtual
  override(Ownable2Step, Ownable) {
181      super._transferOwnership(newOwner);
182    }
```

Figure C.1: An example of overriding the `transferOwnership` and `_transferOwnership` functions in `TreehouseRouter.sol`)

- **Remove unnecessary `if` statement.** Line 94 in figure C.2 contains an `if` check that will always be `true`. This is due to a check on the first line in this function that will revert the call if `_newRedemption == address(0)`. Therefore, this `if` clause will never be `false`.

```
94    if (_newRedemption != address(0)) {
95      IERC20(getUnderlying()).approve(_newRedemption, type(uint).max);
96    }
```

Figure C.2: Excerpt from the `setRedemption` function in `Vault.sol`

- **Normalize all immutable variables by making them uppercase**. Most immutable variables throughout the codebase are uppercased. However, some are not. We recommend also uppercasing these:

```
20    address public immutable wstETH;
```

Figure C.3: Immutable variable whose name is not uppercased in `PnlAccountingHelper.sol`

```
28    AggregatorV3Interface public immutable pricefeed;
29
```

```
30    // Rate providers are expected to respond with a fixed-point value with 18
decimals
31    // We then need to scale the price feed's output to match this.
32    uint256 internal immutable _scalingFactor;
```

Figure C.4: Immutable variables whose name is not uppercased in `ChainlinkRateProvider.sol`

```
25    IwstETH public immutable wstETH;
26    IRateProvider public immutable stethRateProvider;
```

Figure C.5: Immutable variables whose name is not uppercased in `WstETHRateProvider.sol`

```
25    address public immutable vault;
```

Figure C.6: Immutable variable whose name is not uppercased in `Strategy.sol`

- **Update incorrect comment**. The comment indicates this is the `wstETH` interface, but it is the `stETH` (=Lido) interface.

```
4    /// @dev https://docs.lido.fi/contracts/wsteth
```

Figure C.7: Incorrect comment in `IstETH.sol`

- **Update confusing struct field name**. The `asset` field in the `RedemptionInfo` struct holds the amount of assets. The current name is confusing, as it sounds like this field holds the address of the asset, not the amount of assets. We recommend updating the name to **`assets`**.

```
35    uint128 asset;
```

Figure C.8: `asset` field of the `RedemptionInfo` struct in `TreehouseRedemption.sol`

- **Simplify code in redeem function**. The `previewRedeem` function internally simply calls `convertToAssets`. Therefore, this process could be made simpler and more gas efficient by first calling `previewRedeem`, and using the returned assets amount to perform the check that is currently performed on line 79.

```
79    if (IERC4626(TASSET).convertToAssets(_shares) < minRedeemInEth) revert
MinimumNotMet();
80
81    IERC20(TASSET).safeTransferFrom(msg.sender, address(this), _shares);
82
83    uint128 _assets = IERC4626(TASSET).previewRedeem(_shares).toUint128();
```

Figure C.9: Excerpt from the redeem function in `TreehouseRedemption.sol`

- **Remove unchecked for-loop increments**. Since Solc 0.8.22; the compiler automatically does this (under certain circumstances); since this project uses Solc 0.8.24, these increments can be removed from the implementation.

```
83    for (uint i; i < _allowedAssets.length; ) {
84      parameters[_strategy].whitelistedAssets.add(_allowedAssets[i]);
85
86      unchecked {
87        ++i;
88      }
89    }
```

Figure C.10: Example use of unchecked for-loop increments in `StrategyStorage.sol`

D. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On September 25, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the Treehouse team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

We performed the audit on the `0xhypn/tETH` private repo at commit `60c4c39c800d280057fefe2ce1945f61c0dc795d`. The fixes were then applied to the `0xhypn/tETH` repo with one commit per issue fix. We reviewed each of these commits specifically, and from that generated the below Detailed Fix Review Results.

After our review of the fixes, the Treehouse team performed updates to the `NavHelper` contract. We did not review these changes.

The Treehouse team also decided to create a new "clean" GitHub repository, `treehouse-gaia/tETH`, that contains a single commit with the final version of the codebase after all of our fixes were applied, as well as Treehouse's changes to the `NavHelper` contract. This fix review appendix will therefore link to the commit (`459ccb1`) in this new repository.

In summary, of the six issues described in this report, Treehouse has resolved four issues and has not resolved the remaining two issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Anyone can steal wstETH tokens accidentally transferred to the TreehouseRouter contract	Resolved
2	Underlying tokens can be “rescued”	Unresolved
3	WstEth.wrap expects stETH amount instead of ETH amount	Resolved
4	Single-asset vault is not a single-asset vault	Unresolved
5	Dangerous storage variable in Strategy contract due to use of delegatecall	Resolved
6	Missing return value check might lead to incorrect event emission	Resolved

Detailed Fix Review Results

TOB-TETH-1: Anyone can steal wstETH tokens accidentally transferred to the TreehouseRouter contract

Resolved in commit `459ccb1`. The implementation was updated to return the return value of the `wstEth.wrap` call instead of returning the wstEth balance of the contract itself. By doing this, only tokens belonging to the caller will be used for the deposit.

TOB-TETH-2: Underlying tokens can be “rescued”

Unresolved in commit `459ccb1`. The client provided the following context for not fixing this issue:

> *TOB-TETH-2: a centralization risk and we acknowledge it as such - it's conceptually no different to another EOA with privileged access.*

TOB-TETH-3: WstEth.wrap expects stETH amount instead of ETH amount

Resolved in commit `459ccb1`. The implementation was updated to pass the `stEth` amount, converted from the returned number of shares that were minted in the `stEth.submit` call.

TOB-TETH-4: Single-asset vault is not a single-asset vault

Unresolved in commit `459ccb1`. The client provided the following context for not fixing this issue:

TOB-TETH-4: is informational, and a subjective opinion on a certain design aspect. To accommodate that would mean making material changes to the codebase, with no corresponding benefit

TOB-TETH-5: Dangerous storage variable in Strategy contract due to use of delegatecall
Resolved in commit 459ccb1. The `strategyExecutor` storage variable was removed from the `Strategy` contract. In its place, an immutable variable holding the address of the `StrategyStorage` contract was added. The `StrategyStorage` contract was updated to include the `strategyExecutor` storage variable, and functionality for changing and retrieving this value was added.

TOB-TETH-6: Missing return value check might lead to incorrect event emission
Resolved in commit 459ccb1. The implementation was updated to check the return value of the `EnumerableSet.add` function. In the case that it is `false`, the execution is reverted.