Treehouse tETH

Security Assessment

August 27, 2024

Prepared for:
Ben Loh
Treehouse Finance

Prepared by: **Michael Colburn, Justin Jacob, Damilola Edwards, Emilio López and David Pokora**

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:

David Pokora, Engineering Director, Application Security
david.pokora@trailofbits.com

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

The following consultants were associated with this project:

Michael Colburn, Consultant
michael.colburn@trailofbits.com

David Pokora, Consultant
david.pokora@trailofbits.com

Damilola Edwards, Consultant
damilola.edwards@trailofbits.com

Justin Jacob, Consultant
justin.jacob@trailofbits.com

Emilio López, Consultant
emilio.lopez@trailofbits.com

Project Timeline

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

Date	Event
July 8, 2024	Pre-project kickoff call
July 18, 2024	Status update meeting #1
July 29, 2024	Delivery of report draft
July 29, 2024	Report readout meeting
August 27, 2024	Delivery of report with fix review appendix

Executive Summary

Engagement Overview

Treehouse engaged Trail of Bits to review the security of the tETH contracts and offchain code. 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.

A team of two consultants from the blockchain team conducted a review focusing on the smart contracts from July 10 to July 23, 2024, for a total of two engineer-weeks of effort. Another team of two consultants from the appsec team conducted a separate review in parallel focusing on the off-chain components from July 10 to July 26, for a total of two engineer-weeks 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 and off-chain components, using automated and manual processes. The final off-chain code was delivered a few days after the review started, on July 15. Towards the end of the smart contract review period the Treehouse team provided additional code for review at commits 728d47 and a930e0 which was reviewed on a best effort basis.

Observations and Impact

The tETH smart contracts relies on privileged actors to manually perform necessary operations; for example, operations related to PnL distribution, funding the redemption contract to enable user withdrawals, updates to state variables that directly impact users' solvency and funds, investments into and divestments from strategies. Additionally we identified two issues related with integration with external protocols (TOB-TETH-1) and (TOB-TETH-2). It is therefore important to highlight the need for a careful review of the documentation and guidelines of protocols the system interacts with to ensure that the integrations are done in line with the recommended best practices. Treehouse should also pay attention to the security of the privileged actor accounts. The Treehouse team mentioned they plan to use a Gnosis multi-signature wallet for this purpose, but the support for this is not yet implemented in the offchain codebase.

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.**

- **Implement a secure way to sign transactions on the off-chain component, that preferably holds the keys on one or more secure hardware devices and requires multi-party approval for a transaction to be processed.** Currently, the system only supports signing transactions with a hardcoded EOA wallet that is embedded in the codebase, and the contracts are not controlled through a multi-signature wallet. The Treehouse team mentioned they will be using multi-signatory with multi-party approval for the transaction to be processed

- **Significantly improve testing of off-chain components.** Currently the off-chain codebase does not have unit tests and test automation, and relies on a manual scenario simulation script for manual testing.

- **Implement automated CI/CD processes for the off-chain components.** These should include automated testing, dependency vulnerability checks (e.g. via Dependabot), source code static analysis (e.g. via Semgrep or CodeQL) and pull request review and approval criteria.

- **Determine if there is a risk in interacting with public RPC providers in the off-chain codebase and adjust accordingly.** Relying on a single external RPC provider as a source of truth could lead to a skewed view of the protocol state if the provider is compromised or their nodes fork off the canonical chain. Sending transactions through the public mempool could also allow for third-parties to perform, for example, sandwich attacks. Consider performing RPC calls to one or more private or self hosted nodes in parallel and compare their results. Evaluate using a private mempool service to submit transactions to the chain.

- **Use integer values for off-chain arithmetic.** Floating point numbers may lose precision in counterintuitive ways. For financial applications in which precision is important, fixed-point math using big integers is a well-established best practice. Python integers are of arbitrary length out of the box.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	Count
High	1
Medium	0
Low	2
Informational	5
Undetermined	1

CATEGORY BREAKDOWN

Category	Count
Access Controls	1
Data Exposure	1
Data Validation	6
Patching	1

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:

- Could an attacker steal funds from the system?
- Are appropriate access controls in place?
- Are the arithmetic calculations performed during token minting and redeeming operations correct?
- Is the protocol vulnerable to denial-of-service (DoS) attacks?
- Is the arithmetic for handling various types of collateral performed correctly?
- Are user-provided parameters sufficiently validated?
- Are there any economic attack vectors in the system?
- Does the protocol convert tokens to and from shares correctly?
- Is the share price prone to manipulation?
- Could the use of low-level calls in the codebase cause any problems?
- Could a user's funds become stuck in the system?
- Do the off-chain components query the chain state adequately?
- Are different chain states sufficiently validated in the off-chain components?

Project Targets

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

tETH protocol

Repository	https://github.com/treehouse-gaia/tETH-protocol
Version	`02c3ab1fafa7610ba43fc3cc905ccad504b39cf3`
Type	Solidity
Platform	EVM

tETH offchain

Repository	https://github.com/treehouse-gaia/tETH-offchain
Version	`2539d30504aec46d2a753fac2c18a3872691507a`
Type	Python
Platform	Linux

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:

- **Deposits :** The TreehouseRouter contract serves as the gateway for deposits into the protocol, all deposited assets are sent to the vault and the depositor receives tETH token in return. The contracts relevant to the deposit execution flow include the `TreehouseRouter`, `Vault`, `iETH` and `tETH` contracts. We conducted the following manual and automated reviews of these contracts:
    - We reviewed the conversion of assets to shares to ensure they were performed correctly
    - We reviewed the use of access control modifiers to ensure that necessary access controls are in place for privileged operations, this led to the identification of issue (TOB-TETH-3) which allows users to directly convert tETH tokens for iETH and potentially introducing errors in PnL accounting.
    - We reviewed the deposit flow to ensure that users cannot lose funds through theft or unintended locks.
    - We reviewed the integration and interactions with external protocols to ensure that the assumptions made do not introduce flaws in the system. Two issues were identified in this regard (TOB-TETH-1) and (TOB-TETH-2).
- **Redemption:** Redemption requests are handled via the `TreehouseRedemption` contract, after the minimum waiting period is passed, users can then proceed to finalize the withdrawal process, at this point, the underlying ETH/WETH is transferred to the user. We conducted a the following manual and automated reviews of the contracts relevant to the redemption flow:
    - We reviewed the access controls on the functions to ensure that only privileged actors could update critical system values
    - We reviewed the redemption finalization flow to ensure that waiting periods could not be bypassed
    - We reviewed the state changes that occur during the creation of redemption requests, cancellations and finalizations to ensure consistency and the possibility of re-entrances and replay attacks
- **Accounting :** The accounting mechanism employed by the tETH protocol involves the use of two separate tokens, `tETH` a yield-bearing ERC-4626 vault token which

represent shares and iETH an internal accounting unit representing the total value in the vault and used for PnL calculation after harvest from strategies. We conducted a the following manual and automated reviews of the contracts relevant to the internal accounting:

  - We reviewed the contract for flaws that would allow users to manipulate share prices.
  - We reviewed the interest accrual process to determine whether it is vulnerable to front-running or sandwich attacks.
  - We reviewed the arithmetic that is performed and the state changes that occur during deposits, redemption requests, cancellations and finalizations to identify any edge cases that may result in undefined behavior.

- **Rate providers:** The system relies on the rate provider contracts to query price feeds and asset values, we conducted a manual review on these contracts to ensure proper integration and data staleness checks, we found one issue related to this (TOB-TETH-2)
- **Strategies :** The `strategy` folder consists of multiple contracts relating to strategies and actions, we conducted a manual review on these components to ensure general correctness and that the functions have the correct access controls in place.
- **Off-chain scripts:** The system uses external programs that query the chain state through a RPC provider, and can suggest and eventually execute rebalancing operations to maintain the protocol strategy in a healthy state. We performed automated and manual review of the code to check that its interaction with the chain is correct and that it handles multiple states adequately. We identified several issues in this component, including ones related to maintainability (TOB-TETH-4, TOB-TETH-5), unsafe use of language functionality (TOB-TETH-6), the interaction with the chain (TOB-TETH-8), and state analysis (TOB-TETH-7, TOB-TETH-9).

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:

- We did not extensively search for front-running vulnerabilities.
- We did not review the high-level economic incentives and disincentives imposed by the system.

- We did not review the off-chain arithmetic in depth, nor the associated strategy logic, parameters, and its soundness in the context of the system. In particular, we did not analyze the impact of performing floating-point arithmetic and the risk of rounding errors it entails.
- In addition, the report does not include an integration found post review: the TreehouseRedemption contract calls the WETH.withdraw function, but the redemption contract is lacking a fallback or receive function. As a result WETH's transfer will revert.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

- Slither: A static analysis framework that can statically verify algebraic relationships between Solidity variables
- Medusa: A cross-platform go-ethereum-based smart contract fuzzer inspired by Echidna
- Semgrep: An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time
- CodeQL: A code analysis engine developed by GitHub to automate security checks

Test Results

The tables below summarizes the type, property conditions and result of each invariants fuzzed on the smart contract codebase. We ran the fuzzer both locally and on the cloud.

Functional Invariants

We ran the following invariants using Medusa to test functions in the `TreehouseRouter` and `TreehouseRedemption` contracts to ensure that they behave as expected. They include checks of preconditions and postconditions expected to hold in the system.

ID	Property	Result
F-TETH-1	stETH/wsETH/ETH balance of depositor should always decrease after a deposit	**Passed**
F-TETH-2	Vault's stETH/wsETH/ETH balance should always increase after a deposit	**Passed**
F-TETH-3	Total supply of tETH should always increase after deposits	**Passed**
F-TETH-5	Depositor's balance of tETH should always increase after a deposit	**Passed**

F-TETH-6	Total supply of tETH should always decrease after redeem	**Passed**
F-TETH-7	User's balance of tETH should always decrease after redeem	**Passed**
F-TETH-8	cancelRedeem should always increase the user's tETH balance	**Passed**
F-TETH-9	finalizeRedeem should always increase user's ETH/WETH balance	**Passed**
F-TETH-10	finalizeRedeem should always reduce vault's ETH/WETH balance	**Passed**

System Invariants

Using medusa, we also added system invariants that check the relationship between global system states. These invariants test the relationships between variables in the contract. Unlike functional invariants, these invariants should hold true regardless of the functions that are executed.

ID	Property	Result
S-TETH-1	Total ETH + WETH balance in the vault should never exceed deposit cap	**Passed**
S-TETH-2	Total supply of IETH should equal total asset value in vault	**Failed**
S-TETH-3	Redemption timelocks should not be bypassable	**Passed**
S-TETH-4	User with no access to ETH/WETH/wsETH should have no tETH shares	**Passed**
S-TETH-5	Users ETH/WETH balance should never exceed provided amount	**Passed**

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, and most of the operations are documented with inline documentation. Asset calculations rely on the assumption that the dollar value of stETH would always be equal to ETH, this could potentially introduce accounting issues if stETH depegs. The offchain code performs arithmetic using floating point numbers, which may introduce precision errors.	**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. The offchain components provide sufficient logs for tracking their internal state.	**Satisfactory**
Authentication / Access Controls	Most functions within the contracts are restricted by access controls in place, permitting only privileged actors to execute them. Users have limited control, primarily restricted to deposits, creating and canceling redemption requests, and finalizing redemptions. However, an issue was identified due to the lack of access control on an implicitly inherited function in the tETH contract (TOB-TETH-3). 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	**Moderate**

	reason through and most of the complexity is left to be handled by the strategy manager via the off-chain 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, are well documented, have a clear purpose, and are appropriately tested. The off-chain codebase is also generally modularized and separated into functions; however it contains multiple instances of code duplication, commented-out code, and special-casing, which reduce maintainability, readability, and hamper reasoning about the code.	
Configuration	As the system is in development, a production configuration is not yet available. The current development configuration contains hard-coded keys, which are not suitable for a production launch. The team expressed that the off-chain component will gain support for managing funds through a multi-signature setup before launch.	**Not Considered**
Cryptography and Key Management	The system does not perform cryptographic operations directly on the off-chain component, and relies on third-party libraries such as web3.py to perform operations such as transaction signing. However, multiple API and wallet keys are currently hardcoded in the code or committed as part of the repository. There is no implemented support for safe runtime provisioning of secrets, e.g. via a secrets vault or password manager.	**Weak**
Data Handling	The system generally validates the data it operates on. We did however find some issues related to data validation in the on-chain (TOB-TETH-2) and off-chain (TOB-TETH-6, TOB-TETH-7) components.	**Moderate**
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.	**Weak**

	Additionally, the Rescuable contract includes a provision that allows the retrieval of any uninvested funds from the Vault.	
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 were 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. It is advisable to establish guidelines for writing delegate calls to prevent such issues. Additionally, the use of assembly is currently limited to checking the return value of delegate calls.	**Moderate**
Maintenance	The off-chain code is organized into logical modules; however, duplicated and commented-out code reduce readability and hinder maintainability. The lack of unit tests also make it difficult to introduce changes to the system with confidence that they do not include regressions or unexpected changes in behavior. The system could also benefit from the use of tools such as Dependabot in CI to automatically keep dependencies up to date in the repository, as well as automated CI/CD workflows to execute tests, perform static analysis and enforce a coding style.	**Weak**
Memory Safety and Error Handling	The off-chain components are built using Python, which is memory safe. Errors are generally checked and handled appropriately.	**Satisfactory**
Testing and Verification	The smart contract codebase contains several unit and integration tests, these tests appear to cover most common use cases of the protocol 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 operations, invariants, or function properties. Furthermore, there is no mutation testing. These methodologies can expose unforeseen edge cases	**Moderate**

	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. These can help ensure the resistance of the application against potential unusual inputs or behaviors. The off-chain component is currently lacking unit testing. While there is a “stress test” script that works as a sort of fuzz test, fuzzing specific functionality could also prove beneficial.	
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	Incorrect accounting logic for stETH deposits	Data Validation	Low
2	Chainlink oracles could return stale price data	Data Validation	Informational
3	Users can redeem tETH tokens to iETH	Access Controls	Informational
4	Secrets checked into source code	Data Exposure	Low
5	Use of outdated libraries	Patching	Informational
6	Potential code execution through deserialization	Data Validation	High
7	Overlapping and non-exhaustive conditions while analyzing cases	Data Validation	Undetermined
8	Potentially duplicate event fetching	Data Validation	Informational
9	Potentially misleading order comparison	Data Validation	Informational

Detailed Findings

1. Incorrect accounting logic for stETH deposits

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TETH-1
Target: `contracts/TreehouseRouter.sol`, `contracts/periphery/Converter.sol`	

Description

An edge case in the way stETH token transfers work may result in a small wei discrepancy when depositing tokens via the `TreehouseRouter` contract or when converting stETH via the `Converter` contract.

stETH is a rebasing token that updates daily to account for staking yield. To support this behavior, the stETH contract tracks each user's shares of the overall pool of ether, which is used to determine their balance of stETH tokens. When transferring tokens, the value is first converted to shares, and it is these shares that are actually transferred from one address to another. Converting between tokens and shares requires integer division that may result in slightly fewer stETH tokens being transferred than expected.

The `TreehouseRouter` contract's `deposit` function considers the ether value of the deposited stETH to correspond to the `amount` value passed as a parameter. As a result, the contract may then mint slightly more iETH tokens to the caller than they actually deposited. The maximum size of this discrepancy is expected to grow over time as Lido continues to grow and accrue staking rewards.

```
 77    function deposit(address _asset, uint256 _amount) public nonReentrant
whenNotPaused {
 78     if (IVault(VAULT).isAllowableAsset(_asset) == false) revert
NotAllowableAsset();
 79     uint _valueInEth;
 80
 81     if (_asset == stETH) {
 82       _valueInEth = _amount;
 83
 84       IERC20(stETH).safeTransferFrom(msg.sender, address(this), _amount);
 85       uint wstethAmount = IwstETH(payable(wstETH)).wrap(_amount);
 86       IERC20(wstETH).transfer(VAULT, wstethAmount);
 87     } else if (_asset == wstETH) {
 88       _valueInEth = IwstETH(payable(wstETH)).getStETHByWstETH(_amount);
```

```
89
90        IERC20(wstETH).safeTransferFrom(msg.sender, VAULT, _amount);
91      } else {
92        _valueInEth = _getDepositInEth(_asset, _amount);
93        IERC20(_asset).safeTransferFrom(msg.sender, VAULT, _amount);
94      }
95
96      _checkEthCap(_valueInEth);
97      uint _shares = _mintAndStake(_valueInEth);
98      emit Deposited(_asset, _amount, _valueInEth, _shares);
99    }
```

Figure 1.1: The deposit function from the `TreehouseRouter` contract (tETH-protocol/contracts/TreehouseRouter.sol#L77–L99)

Exploit Scenario

Many users deposit stETH into the protocol which results in many instances of small amounts of excess iETH being minted. Over time this tracking error may become large enough to have a noticeable impact on PnL accounting or other unexpected side effects.

Recommendations

Short term, snapshot the contract's stETH balance before and after the `safeTransferFrom` call and set `_valueInEth` and the value passed to `wrap` to the difference in the balance to accurately reflect the amount of stETH that was actually taken from the caller.

Long term, carefully review the Lido integration documentation and ensure all known edge cases are accounted for when designing new features.

References

- Lido tokens integration guide

2. Chainlink oracles could return stale price data

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-TETH-2
Target: `contracts/rate-providers/ChainlinkRateProvider.sol`	

Description

The `latestRoundData()` function from Chainlink oracles returns five values: `roundId`, `answer`, `startedAt`, `updatedAt`, and `answeredInRound`. The `ChainlinkRateProvider` contract reads only the `answer` value and discards the rest. This can cause outdated prices to be used for token conversions.

```
45    function getRate() external view override returns (uint256) {
46     (, int256 price, , , ) = pricefeed.latestRoundData();
47     require(price > 0, 'Invalid price rate response');
48     return uint256(price) * _scalingFactor;
49    }
```

Figure 2.1: All returned data other than the `answer` value is ignored during the call to a Chainlink feed's `latestRoundData` method. (`tETH-protocol/contracts/rate-providers/ChainlinkRateProvider.sol#L45-L49`)

According to the Chainlink documentation, if the `latestRoundData()` function is used, the `updatedAt` value should be checked to ensure that the returned value is recent enough for the application.

Recommendations

Short term, make sure that the oracle queries check for up-to-date data and revert or return a sentinel value (e.g., 0) to indicate stale data.

Long term, review the documentation for Chainlink and other oracle integrations to ensure that all of the security requirements are met to avoid potential issues, and add tests that take these possible situations into account.

3. Users can redeem tETH tokens to iETH

Severity: **Informational**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-TETH-3
Target: `Contracts/tETH.sol`	

Description

The tETH contract exposes the `withdraw` and `redeem` functions from the inherited ERC4626 token contract, this allows any user to redeem their tETH tokens for iETH.

iETH tokens are minted purely for accounting purposes, during deposits, the iETH token is minted equivalent to the ETH value of the amount of asset deposited and burned during the finalization of redemption. The iETH token is also used in estimating total profit or loss accrued over a period of time and then is rebased to maintain a 1:1 peg between tETH and ETH. Ideally the total supply of iETH should be held in the tETH contract since it represents the total share value.

However, due to the absence of access controls on the inherited `withdraw` and `redeem` functions in the tETH contract, users can directly convert their tETH tokens to iETH, although iETH cannot be directly converted to ETH within the protocol, this action could lead to unintended side effects like potentially introducing accounting miscalculations (depending on how pnl accounting is performed on the offchain side) or possible frontrunning/backrunning attacks.

```
 210    function redeem(uint256 shares, address receiver, address owner) public
virtual returns (uint256) {
 211        uint256 maxShares = maxRedeem(owner);
 212        if (shares > maxShares) {
 213            revert ERC4626ExceededMaxRedeem(owner, shares, maxShares);
 214        }
 215
 216        uint256 assets = previewRedeem(shares);
 217        _withdraw(_msgSender(), receiver, owner, assets, shares);
 218
 219        return assets;
 220    }
```

Figure 3.1: Invokable redeem function in the OZ ERC4626 token contract implementation.

Recommendations

Short term, consider adding access controls on the inherited redeem and withdraw functions within the tETH contract in order to revoke direct access or allow only specific users access.

Long term, carefully review all public and external functions within imported libraries/dependencies and add proper access controls on functions that should not be invoked directly by users.

4. Secrets checked into source code

Severity: **Low**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-TETH-4

Target: `tETH-offchain/utils/web3_func.py`, `tETH-offchain/src/env_handler/fork_blockchain.py`, `tETH-offchain/config/thirdparty_config/strat_config_thirdparty.yml`, `tETH-offchain/config/deploy_config/strat_config_deploy.yml`

Description

Several secrets, including API keys and Ethereum private keys, are checked into the source code repository and present in the Git history. If attackers have access to the application source code, they would have access to said secrets. Additionally, checking the shared secret into the source code repository gives all employees and contractors with access to the repository access to the secrets. Secret values such as API keys and Ethereum private keys should never be stored in plaintext in source code repositories, as they can become valuable tools to attackers if the repository is compromised. The figures below show a few samples of the identified secrets, but these are not an exhaustive list.

```
"eth":
f"https://api.etherscan.io/api?module=contract&action=getabi&address={cid}&apikey=RE
DACTED_KEY",
```

Figure 4.1: Example Etherscan API key present in the repository (`tETH-offchain/utils/web3_func.py`)

```
DEFAULT_RPC_URL = "https://mainnet.infura.io/v3/REDACTED_KEY"
```

Figure 4.2: Example Infura key present in the codebase (`tETH-offchain/src/env_handler/fork_blockchain.py`)

```
DEPLOYER_PRIVATE_KEY: '0xREDACTED'
PREFERRED_NODE_URL: https://rpc.buildbear.io/REDACTED
```

Figure 4.3: Example private key and Buildbear key (`tETH-offchain/config/deploy_config/strat_config_deploy.yml`)

The severity has been marked as low, as the system is not yet in production and these keys correspond to testing instances.

Exploit Scenario
An attacker obtains a copy of the source code from a former employee. She extracts the API keys for Etherscan and the Ethereum node RPC and performs multiple requests, causing increased monetary expenses for the Treehouse team, or a denial of service due to quota exhaustion when the Treehouse portfolio manager attempts to run the off-chain components. She also extracts the deployer private key and performs unauthorized operations on-chain with it.

Recommendations
Short term, remove the hard-coded secrets from source code and rotate their values.

Long term, consider storing the secrets in a secret management solution such as 1Password or Hashicorp Vault. Use tools such as Trufflehog on your CI/CD pipeline to detect secrets mistakenly committed to the repository.

References
- GitHub: Removing sensitive data from a repository

5. Use of outdated libraries

Severity: **Informational**	Difficulty: **Undetermined**
Type: Patching	Finding ID: TOB-TETH-5
Target: `tETH-offchain/requirements.txt`	

Description

We used `pip audit` to detect the use of outdated dependencies in the offchain codebase, which identified a number of vulnerable packages referenced by the `requirements.txt`.

The following is a list of the vulnerable dependencies used in the offchain codebase, and known vulnerabilities that affect the versions currently used by the codebase:

- `aiohttp` (PYSEC-2024-24, PYSEC-2023-250, PYSEC-2023-251, PYSEC-2024-26, GHSA-7gpw-8wmc-pm8g, GHSA-5m98-qgg9-wh84)
- `certifi` (GHSA-248v-346w-9cwc)
- `eth-abi` (GHSA-3qwc-47jf-5rf7)
- `idna` (PYSEC-2024-60)
- `requests` (GHSA-9wx4-h78v-vm56)
- `urllib3` (GHSA-34jh-p97f-mpxf)

In many cases, the use of a vulnerable dependency does not necessarily mean the application is vulnerable. Vulnerable methods from such packages need to be called within a particular (exploitable) context. To determine whether the offchain applications are vulnerable to these issues, each issue will have to be manually triaged. The severity is marked informational as upon preliminary inspection, these issues do not appear to impact the offchain codebase.

Recommendations

Short term, update system dependencies to their latest versions wherever possible. Use tools such as `pip audit` to confirm that no vulnerable dependencies remain.

Long term, implement these checks as part of the CI/CD pipeline of application development. Integrate an automated solution such as Dependabot into your development process to assist in promptly detecting and updating dependencies with known security problems.

6. Potential code execution through deserialization

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-TETH-6
Target: `tETH-offchain/utils/web3_func.py`	

Description

The offchain codebase loads contract ABIs from files and deserializes them into Python objects through the `pickle.load` function. If any of this data comes from untrusted input controlled by an attacker, this can lead to remote code execution (since pickle can execute arbitrary code that would be encoded within the data).

```
def load_contract_abi(chain: str, address: str):
    # (...)
    file_name = f"abi_{chain}_{address}.pkl"
    file_path = os.path.join(ABI_DIR, file_name)

    if os.path.isfile(file_path):
        with open(file_path, "rb") as f:
            abi = pickle.load(f)
        return abi
    else:
        msc_logger.error(f"Cannot find abi file locally for {address}")
        raise Exception(f"Cannot find abi file locally for {address}")
```

Figure 6.1: The data is loaded and deserialized using `pickle.load` (`tETH-offchain/utils/web3_func.py`)

While these files appear to be generated from the program itself during execution as a sort of caching mechanism, there is no validation performed to ensure that they are trustworthy and have not been tampered with.

Exploit Scenario

An attacker with access to the code repository or the portfolio manager's computer replaces one of the pickle files with a malicious copy that, when loaded, patches the executing code to silently modify the on-chain transactions generated by the program. When the portfolio manager executes the offchain code and the pickle file gets loaded, the process produces malicious transactions, leading to unexpected system state or a loss of funds.

Recommendations

Short term, consider using a different data file format that is not prone to the same vulnerabilities (e.g., JSON). If pickle files are essential to the system, ensure that all pickle files come from trusted sources and are explicitly reviewed. If possible, consider signing the pickle file to ensure that unreviewed pickle files are not executed by the system. Additionally, add relevant code comments to inform future reviewers that the specific use is safe.

References

- Never a dill moment: Exploiting machine learning pickle files

7. Overlapping and non-exhaustive conditions while analyzing cases

Severity: **Undetermined**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-TETH-7
Target: `tETH-offchain/utils/print_func.py`	

Description

The codebase has a `printAnalysisResult` function that interprets a value in the context of a set of bounds and thresholds and returns an identifier for each type of state. This identifier is sometimes used by the caller to determine if further actions need to be taken. However, the function's implementation does not exhaustively cover all possibilities for the "crossing up" case, and checks overlapping cases on the "crossing down" case. This could eventually result in a misinterpretation of the data being shown or used.

For the "crossing up" case, assuming `lower_bound <= upper_bound <= threshold`, we can see in figure 7.1 that the first conditional covers the `[lower_bound, upper_bound]` range (1), the second conditional covers the `(-inf, lower_bound)` range (2), the third conditional covers the `(upper_bound, threshold)` value (3), and the fourth conditional covers the `(threshold, +inf)` range (4). When combined, these ranges cover the majority of the values, with the exception of the threshold value itself.

```
if threshold_type == "crossing_up":
    if lower_bound <= value <= upper_bound: # (1)
        sta_logger.info(f"[{LVL_1}] {metric_name} is within the bounds.")
        # alert = f"metric_name_{LVL_1}"
        alert_level = f"{LVL_1}_within_bounds"
    elif value < lower_bound: # (2)
        sta_logger.info(f"[{LVL_2}]  {metric_name} is below lower bound.")
        # action needed
        alert_level = f"{LVL_2}_below_lower"
    elif upper_bound < value < threshold: # (3)
        sta_logger.info(f"[{LVL_2}]  {metric_name} is above upper bound.")
        # manager decision
        alert_level = f"{LVL_2}_above_upper"
    elif value > threshold: # (4)
        sta_logger.info(f"[{LVL_3}]  {metric_name} is above threshold level.")
        # action needed
        alert_level = f"{LVL_3}_above_thres"
```

Figure 7.1: The "crossing up" logic in `printAnalysisResult` (`tETH-offchain/utils/print_func.py`)

This means that, if `value` is equal to `threshold`, `alert_level` will not be set and nothing will be logged, which is likely not intentional.

In the "crossing down" case, assuming `threshold <= lower_bound <= upper_bound`, we can see in figure 7.2 that the first conditional covers the `[lower_bound, upper_bound]` case (1), the second conditional covers the `(-inf, lower_bound)` range (2), the third conditional covers the `(upper_bound, +inf)` range (3) and the fourth conditional (4) is dead code – any such cases will be covered by (2) already, as `threshold <= lower_bound`).

```
elif threshold_type == "crossing_down":
    if lower_bound <= value <= upper_bound: # (1)
        sta_logger.info(f"[{LVL_1}] {metric_name} is within the bound.")
        # no action needed
        alert_level = f"{LVL_1}_within_bounds"

    elif value < lower_bound: # (2)
        sta_logger.info(f"[{LVL_3}]  {metric_name} is below lower bound.")
        # action needed
        alert_level = f"{LVL_3}_below_lower"

    elif upper_bound < value: # (3)
        sta_logger.info(f"[{LVL_2}]  {metric_name} is above upper bound.")
        # manager decision
        alert_level = f"{LVL_2}_above_upper"

    elif value < threshold: # (4)
        sta_logger.info(f"[{LVL_3}]  {metric_name} is below threshold level.")
        # action needed
        alert_level = f"{LVL_3}_below_thres"
```

Figure 7.2: The "crossing down" logic in `printAnalysisResult` (`tETH-offchain/utils/print_func.py`)

This means that the threshold alert will never trigger on the "crossing down" case, which is unlikely to be the intended behavior.

Recommendations

Short term, adjust the conditionals so that they cover the expected ranges and work as intended. Document the relationship between threshold, lower and upper bound values. Write unit tests for this function to ensure it continues to behave as intended.

Long term, enhance the testing suite of off-chain components to verify functions perform as expected. implement automated runs of said tests as part of the CI/CD pipeline of application development.

8. Potentially duplicate event fetching

Severity: **Informational**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-TETH-8
Target: `tETH-offchain/utils/pool_func.py`	

Description

The codebase has a `getPoolEvents` function that is currently used to collect Lido's `TokenRebased` events from the chain. This function repeatedly queries the RPC for any events between a series of blocks, combines them into a single list, and returns the information. However, the function may collect and return the same event more than once, which can be unexpected and may skew the results of, for instance, the Lido staking APR SMA.

```
start = _start_block
end = _end_block if _end_block is not None else web3.eth.get_block_number()
event_list = []
step = (end - start) // delta

for i in range(0, step + 1, 1):
    sBlock = start + i * delta
    eBlock = sBlock + delta
    # (...) get the events in blocks [sBlock, min(eBlock, end)]
```

Figure 8.1: The logic used to split a large block range into smaller ones (`tETH-offchain/utils/pool_func.py`)

The function will perform a series of queries that each span a `delta` amount of blocks. The end block used on a query will be the start block of the following query. However, the RPC queries used to fetch the logs take an inclusive `[fromBlock, toBlock]` range, as seen on the implementation by Go Ethereum and on ethers.js documentation. This means that any events that happen on a block number that is on the edge of a query will be received twice.

For example, for a delta of 49999, a start block of 10000 and end of 109998, the code will query the ranges `[10000, 59999]`, `[59999, 109998]`, `[109998, 109998]`. Any events on blocks 59999 and 109998 will be duplicated.

Recommendations

Short term, verify this behavior with your RPC provider and update the code to not query events on the same block twice. Anecdotal evidence on the Internet suggests this behavior may vary on other RPC implementations. Alternatively, deduplicate events based on unique

data such as their transaction hash. Add unit tests to ensure that the function behaves as expected on edge cases such as this one.

Long term, review the documentation when integrating with third-party libraries and services and be aware of their specific behavior on edge cases.

9. Potentially misleading order comparison

Severity: **Informational**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-TETH-9
Target: `tETH-offchain/src/execution/analyze_execution.py`	

Description

The codebase has a `compare_order` function that is used to compare an on-chain (simulated) order with an off-chain suggestion. According to the function documentation, if the on-chain order value matches, within a certain tolerance, the value computed for the off-chain suggestion, the function returns true, otherwise it returns false. However, the implementation will also always return true if the on-chain order has not yet been simulated, irrespective of the order value, which could be unexpected and misleading.

This behavior is documented with a TODO comment in the implementation code, as shown on figure 9.1.

```
isSimulated = onchain_order["isSimulated"]

# TODO: The logic here need to be updated.
# Technically the unsimulated order should not reach here. But the current logic
# may reach here. So I just keep this first, avoiding affecting the whole script.
if isSimulated:
    # (...) perform the comparison and return True or False
else:
    return True
```

Figure 9.1: The logic used to handle unsimulated orders (`tETH-offchain/src/execution/analyze_execution.py`)

Recommendations

Short term, update the function to throw an error or return false or a different sentinel value if comparing unsimulated orders is unacceptable. Correct any calling code paths to ensure no unsimulated orders reach this function. Add a test to ensure unsimulated orders are identified and handled correctly.

Long term, document any specific requirements in the function documentation, so that users are aware of such caveats. Follow the principle of least astonishment when implementing helper functions.

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.

- There are many instances where an if statement is compared against the Boolean `true` or `false` directly. The comparison to `true` or `false` values can be dropped to simplify the code.

```
78    if (IVault(VAULT).isAllowableAsset(_asset) == false) revert
  NotAllowableAsset();
```

Figure C.1: An example of an if statement with an unnecessary Boolean comparison. (tETH-protocol/contracts/TreehouseRouter.sol#78)

- The variable names of the constants of the `MainnetLidoAddresses` contract do not follow the `ALL_CAPS` naming convention for constant values.
- There are several instances of unchecked blocks being used to manually optimize simple loop increments. As these contracts specify Solidity 0.8.24, they benefit from the built-in optimization added in version 0.8.22 that automatically optimizes this as part of the compiling process, rendering these `unchecked` blocks redundant.

```
 101    function whitelistActions(uint _strategyId, bytes4[] calldata
_whitelistedActions) external onlyOwner {
 102     for (uint i; i < _whitelistedActions.length; ) {
 103       if
(parameters[_safeGetStrategyAddress(_strategyId)].whitelistedActions.add(_whiteliste
dActions[i]) == false)
 104         revert AlreadyExist();
 105
 106       emit ActionWhitelisted(_whitelistedActions[i]);
 107
 108       unchecked {
 109         ++i;
 110       }
 111     }
 112    }
```

Figure C.2: An example of an unnecessary unchecked block. (tETH-protocol/contracts/strategy/StrategyStorage.sol#101–112)

- There are large amounts of commented-out code on the offchain codebase. If the code is no longer needed, it should be removed to improve readability and maintainability.

- The offchain codebase contains several blocks of code that are duplicated several times with minimal changes, for example in the `validate_order` function. Such code should be refactored to improve maintainability and readability.
- Multiple files in the offchain codebase contain code that adjusts `sys.path` in runtime. The code should be reorganized to make proper use of packages and modules, and a way to install the solution should be added to the repository.
- The offchain codebase contains multiple magic values hardcoded throughout the files, a few examples are shown below. These should be either converted to constants or moved to the configuration files. Some don't match the documentation that accompanies them (e.g., in figure C.4 it says 20% but calculates 30% instead).

```
tolerance = 1 / 100  # 1% slippage
```

Figure C.3: A hardcoded tolerance value (tETH-offchain/src/execution/analyze_execution.py#112)

```
STAKE_RATE_LOWER = STAKE_RATE_THRES * 1.05  # 5% higher than borrow rate
STAKE_RATE_UPPER = STAKE_RATE_THRES * 1.3  # 20% higher than borrow rate
```

Figure C.4: Hardcoded lower and upper percentage bounds (tETH-offchain/src/state_handler/analyze_state.py#139–140)

- The `fork_blockchain` function currently sleeps for 10 seconds while `anvil` starts and the forked chain becomes usable. This could be improved and made more reliable by performing a health check of the forked chain instead.

D. Automated Static Analysis

This appendix describes the setup of the automated analysis tools used during this audit for the off-chain components.

Though static analysis tools frequently report false positives, they detect certain categories of issues, such as memory leaks, misspecified format strings, and the use of unsafe APIs, with essentially perfect precision. We recommend periodically running these static analysis tools and reviewing their findings.

Semgrep

To install Semgrep, we used `pip` by running `python3 -m pip install semgrep`.

To run Semgrep on the codebase, we ran the following command in the root directory of the project (running multiple predefined rules simultaneously by providing multiple `--config` arguments):

```
semgrep --config "p/trailofbits" --config "p/ci" --config "p/python"
--config "p/security-audit" --metrics=off
```

We also used `semgrep-rules-manager` to fetch and run other third-party rules.

We recommend integrating Semgrep into the project's CI/CD pipeline. To thoroughly understand the Semgrep tool, refer to the Trail of Bits Testing Handbook, where we aim to streamline the use of Semgrep and improve security testing effectiveness. Also, consider doing the following:

- Limit results to error severity only by using the `--severity` ERROR flag.
- Focus first on rules with high confidence and medium- or high-impact metadata.
- Use the SARIF format (by using the `--sarif` Semgrep argument) with the SARIF Viewer for Visual Studio Code extension. This will make it easier to review the analysis results and drill down into specific issues to understand their impact and severity.

CodeQL

We installed CodeQL by following CodeQL's installation guide.

After installing CodeQL, we ran the following command to create the project database for the Treehouse offchain repository:

```
codeql database create treehouse.db --language=python
```

We then ran the following command to query the database:

```
codeql database analyze treehouse.db --format=sarif-latest
--output=codeql_res.sarif -- python-lgtm-full
python-security-and-quality python-security-experimental
```

For more information about CodeQL, refer to the CodeQL chapter of the Trail of Bits Testing Handbook.

E. 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 August 9, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the Treehouse team for the off-chain issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

While the team provided commentary as to the status of each issue, they did not provide specific references to independent commits or pull requests that address each finding. Instead, they provided us with a new version of the `tETH-offchain` repository, identified by the hash `c6a6a11a46b28a52f69c98ddb62e8853d0bcc23c`. This new version of the codebase is a major rewrite of the off-chain code, and as such, the affected code may not be present in its original form on the newer codebase. Instead of a direct fix review, we sought to see, within reason and time constraints, if the same problems that were reported originally are present in the new codebase.

On August 23, 2024, Trail of Bits reviewed an additional fix for issue TOB-TETH-7 contained in commit `84ef318974eff0c2f0e1c29d0b74416233ae361a`.

On August 27, 2024, Trail of Bits reviewed an additional commit (`c00db745fdac4dbd8f07635026fd193cc1abaf5c`) that includes the fixes for the on-chain issue TOB-TETH-3.

In summary, of the 9 off issues described in this report, Treehouse has resolved 4 issues, and has not resolved the remaining 5 issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Incorrect accounting logic for stETH deposits	Unresolved
2	Chainlink oracles could return stale price data	Unresolved
3	Users can redeem tETH tokens to iETH	Resolved
4	Secrets checked into source code	Unresolved
5	Use of outdated libraries	Unresolved

6	Potential code execution through deserialization	Resolved
7	Overlapping and non-exhaustive conditions while analyzing cases	Resolved
8	Potentially duplicate event fetching	Unresolved
9	Potentially misleading order comparison	Resolved

Detailed Fix Review Results

TOB-TETH-1: Incorrect accounting logic for stETH deposits
Unresolved in commit `c00db74`. The client provided the following context for not fixing this issue:

> *TOB-TETH-1: After extensive discussion, it is concluded that this issue results from a rounding error on Lido's end and is not economically exploitable. The discrepancy is minimal, affecting the vault by at most 2 wei of wstETH per deposit, regardless of the deposit size. For instance, 100,000 deposits of 1 stETH each would lead to a shortfall of only 0.000000000000200000 wstETH, which is negligible. This issue will be acknowledged as "will not fix" since it does not pose any significant economic risk or exploitation potential.*

TOB-TETH-2: Chainlink oracles could return stale price data
Unresolved in commit `c00db74`. The client provided the following context for not fixing this issue:

> *TOB-TETH-2: We primarily use Chainlink oracles to price our vault NAV during accounting. Many protocols use oracles directly without implementing staleness checks. We can address this issue off-chain by performing our own staleness checks before running our accounting processes. This issue will be acknowledged as "will not fix," but we will note that staleness checks will likely be performed off-chain before accounting.*

TOB-TETH-3: Users can redeem tETH tokens to iETH
Resolved in commit `c00db74`. The implementation was updated to override the `_deposit` and `_withdraw` functions of the inherited ERC4626 contract. Therefore, the exposed `redeem` and `withdraw` functions (from the inherited ERC4626 contract) can no longer be called by any account, only by the account with the `Minter` role. Additionally, this limits the exposed `mint` function to only be callable by an account with the `Minter` role.

TOB-TETH-4: Secrets checked into source code
Unresolved in commit `c6a6a11`. While Treehouse has done a first step towards resolving this issue and refactored some of the hard-coded secrets into configuration variables, we

still find valid, unrevoked secrets present in the repository (for instance, several node URLs and an Etherscan API key on file `config/thirdparty_config/strat_config_thirdparty.yml`, or NodeReal keys hardcoded in tests), as well as committed on the repository's Git history (such as an Infura API key, `86c…b5f`).

The team has noted through comments in the codebase that they intend to move these secrets to a safer place (like a secrets vault) before moving to production. This is, however, not yet implemented on the codebase as of commit `d00c0a9`, and we did not observe any work towards integration with a secrets vault solution in the codebase.

TOB-TETH-5: Use of outdated libraries
Unresolved in commit c6a6a11. Treehouse has only updated one dependency in the codebase, `urllib3`. We do not observe any new processes or workflows in the repository for becoming aware of vulnerable dependencies nor for taking actions to update them.

The client provided the following context for this finding's fix status:

> *TOB-TETH-5: We commonly use stable versions of packages, which are not necessarily the latest versions. It is impractical to mandate the use of the latest versions for all dependencies. Therefore, this recommendation is not feasible. No action needed*

TOB-TETH-6: Potential code execution through deserialization
Resolved in commit c6a6a11. The codebase has been refactored and no longer uses pickle files for any of its functionality.

TOB-TETH-7: Overlapping and non-exhaustive conditions while analyzing cases
Resolved in commit 84ef318. The code shown in the finding has been reworked and rewritten, and its logic now lives as function `analyze_metrics` in file `src/state_handler/analyze_state.py`. The original fix reviewed in commit c6a6a11 was still affected by both logic errors explained in the finding, and sample test cases are provided in appendix G to more easily showcase said issues; however this has now been resolved in commit 84ef318.

It is worth noting that some of the new test cases introduced to test this functionality in function `test_analyze_metrics_higher` from file `tests/state_handler/test_analyze_state.py` misuse the `analyze_metrics` function by providing a threshold value higher than the upper bound on the `higher_better` case, while the function expects a threshold value lower than the lower bound. This expected usage is exemplified by the arithmetic relation between the arguments passed by codebase, e.g. while calculating the Lido stake rate. The new test functions also do not exercise all edge cases. The expected usage and the relationship between threshold, lower and upper bound values has also not been documented in the codebase.

The client provided the following context for this finding's fix status:

> *Since we are upgrading our code base for v2 vault, many of the test cases may not be applicable anymore after some core functions are changed. The issue related with test cases are acknowledged and we will be fixing all the test cases together after we finish code updates for v2 vault.*

TOB-TETH-8: Potentially duplicate event fetching
Unresolved in commit c6a6a11. The code, now living on function `get_pool_events` in file `utils/web3_func.py`, continues to exhibit the same issue of overlapping range generation, which might result in duplicated events. A sample test case is provided in appendix G to more easily showcase the issue.

The client provided the following context for this finding's fix status:

> *TOB-TETH-8: This issue is deemed unlikely to affect our implementation as we are querying for a single event type. The possibility of duplicating a single event is negligible. The overall code structure has been improved, and the current approach is considered sufficient for our needs.*

TOB-TETH-9: Potentially misleading order comparison
Resolved in commit c6a6a11. The `compare_order` function now logs an error and returns false if an unsimulated order is detected.

F. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

G. Fix Review Test Cases

The following test functions exercise the functionality provided by the `analyze_metrics` function, in both "`lower_better`" and "`higher_better`" scenarios, to cover the function edge cases. The test failures observed by running these tests against the codebase as of commit `c6a6a11` are included as figure G.2.

```
def test_analyze_metrics_lower(self) -> None:
    """Test the "lower_better" case of the analyze_metrics function.

    This assumes the following range setup:

    Lower          Upper         Threshold
    20             60            100
    """
    cases = [
        (10, "MODERATE_below_lower", "below lower bound"),
        (20, "BAU_within_bounds", "exactly lower bound"),
        (40, "BAU_within_bounds", "within [L, U] bounds"),
        (60, "BAU_within_bounds", "exactly upper bound"),
        (80, "MODERATE_above_upper", "above upper bound, under threshold"),
        (100, "EXTREME_above_thres", "exactly threshold"),
        (120, "EXTREME_above_thres", "above threshold"),
    ]

    for (value, expected_result, msg) in cases:
        with self.subTest(msg=msg, value=value, expected_result=expected_result):
            result = analyze_metrics("Metric", value, 60, 20, 100, "lower_better")
            self.assertEqual(result, expected_result)

def test_analyze_metrics_higher(self) -> None:
    """Test the "higher_better" case of the analyze_metrics function.

    This assumes the following range setup:

    Threshold      Lower         Upper
    20             60            100
    """
    cases = [
        (10, "EXTREME_below_thres", "below threshold"),
        (20, "EXTREME_below_thres", "exactly threshold"),
        (40, "EXTREME_below_lower", "below lower bound, greater than threshold"),
        (60, "BAU_within_bounds", "exactly lower bound"),
        (80, "BAU_within_bounds", "within [L, U] bounds"),
        (100, "BAU_within_bounds", "exactly upper bound"),
        (120, "MODERATE_above_upper", "above upper bound"),
    ]

    for (value, expected_result, msg) in cases:
        with self.subTest(msg=msg, value=value, expected_result=expected_result):
```

```
            result = analyze_metrics("Metric", value, 100, 60, 20, "higher_better")
            self.assertEqual(result, expected_result)
```

Figure G.1: Test functions for `analyze_metrics`

```
======================================================================
ERROR: test_analyze_metrics_lower (__main__.TestAnalyzeState) [exactly threshold] (value=100,
expected_result='EXTREME_above_thres')
Test the "lower_better" case of the analyze_metrics function.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "DIR/tests/state_handler/test_analyze_state.py", line 47, in test_analyze_metrics_lower
    result = analyze_metrics("Metric", value, 60, 20, 100, "lower_better")
  File "DIR/tests/state_handler/../../src/state_handler/analyze_state.py", line 136, in
analyze_metrics
    raise Exception("Not found matched alert level.")
Exception: Not found matched alert level.

======================================================================
FAIL: test_analyze_metrics_higher (__main__.TestAnalyzeState) [below threshold] (value=10,
expected_result='EXTREME_below_thres')
Test the "higher_better" case of the analyze_metrics function.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "DIR/tests/state_handler/test_analyze_state.py", line 71, in test_analyze_metrics_higher
    self.assertEqual(result, expected_result)
AssertionError: 'EXTREME_below_lower' != 'EXTREME_below_thres'
- EXTREME_below_lower
?               ^^^ ^
+ EXTREME_below_thres
?               ^^^ ^

======================================================================
FAIL: test_analyze_metrics_higher (__main__.TestAnalyzeState) [exactly threshold] (value=20,
expected_result='EXTREME_below_thres')
Test the "higher_better" case of the analyze_metrics function.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "DIR/tests/state_handler/test_analyze_state.py", line 71, in test_analyze_metrics_higher
    self.assertEqual(result, expected_result)
AssertionError: 'EXTREME_below_lower' != 'EXTREME_below_thres'
- EXTREME_below_lower
?               ^^^ ^
+ EXTREME_below_thres
?               ^^^ ^
```

Figure G.2: Test failures observed by running the tests in figure G.1

The following sample test function exercises the functionality provided by the `get_pool_events` function. The test failure observed by running this test is included as figure G.4.

```
def test_get_pool_events_no_duplicate(self) -> None:
    events = get_pool_events(
            "eth",
            "0xae7ab96520de3a18e5e111b5eaab095312d7fe84",
```

```
            20412190-49999,
            20412190+49999,
            "TokenRebased",
            "https://eth-mainnet.nodereal.io/v1/REDACTED_SECRET",
        )
    tx_hashes = [event["transactionHash"] for event in events]

    # use a set to deduplicate events. If there are no repeated
    # events / tx hashes, both the set and the list should be the
    # same length
    self.assertEqual(len(set(tx_hashes)), len(tx_hashes))
```

Figure G.3: Test function for `get_pool_events`

```
======================================================================
FAIL: test_get_pool_events_no_duplicate (__main__.TestWeb3Func)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "DIR/tests/utils/test_web3_func.py", line 35, in
test_get_pool_events_no_duplicate
    self.assertEqual(len(set(tx_hashes)), len(tx_hashes))
AssertionError: 13 != 14
```

Figure G.4: Test failure observed by running the test in figure G.3