Treehouse tETH Audit Report

Signature: f90caf1c6212d56a8b613162ec9031f9c166049838df63644bd05ed61c53667b

Treehouse tETH Audit Report

Executive Summary

From Sept 20, 2024, to Sept 30, 2024, the Treehouse team engaged Fuzzland to conduct a thorough security audit of their tETH project. The primary objective was to identify and mitigate potential security vulnerabilities, risks, and coding issues to enhance the project's robustness and reliability. Fuzzland conducted this assessment over 20 person-days, involving 2 engineers who reviewed the code over a span of 10 days. Employing a multifaceted approach that included static analysis, fuzz testing, formal verification, and manual code review, the Fuzzland team identified 10 issues across different severity levels and categories.

Scope

Project Name	Treehouse tETH
Repository	tETH-protocol
Commit	203f89837f0da1b64b462bbb390ba2c0b0e30a4d
Language	Solidity - Ethereum
Scope	**/*.sol

Disclaimer

The audit does not ensure that it has identified every security issue in the smart contracts, and it should not be seen as a confirmation that there are no more vulnerabilities. The audit is not exhaustive, and we recommend further independent audits and setting up a public bug bounty program for enhanced security verification of the smart contracts. Additionally, this report should not be interpreted as personal financial advice or recommendations.

Auditing Process

- Static Analysis: We perform static analysis using our internal tools and Slither to identify potential vulnerabilities and coding issues.
- Fuzz Testing: We execute fuzz testing with our internal fuzzers to uncover potential bugs and logic flaws.
- Invariant Development: We convert the project into Foundry project and develop Foundry invariant tests for the project based on the code semantics and documentations.
- Invariant Testing: We run multiple fuzz testing tools, including Foundry and ItyFuzz, to identify violations of invariants we developed.
- Formal Verification: We develop individual tests for critical functions and leverage Halmos to prove the functions in question are not vulnerable.
- Manual Code Review: Our engineers manually review code to identify potential vulnerabilities not captured by previous methods.

Vulnerability Severity

We divide severity into four distinct levels: high, medium, low, and info. This classification helps prioritize the issues identified during the audit based on their potential impact and urgency.

- **High Severity Issues** represent critical vulnerabilities or flaws that pose a significant risk to the system's security, functionality, or performance. These issues can lead to severe consequences such as fund loss, or major service disruptions if not addressed immediately. High severity issues typically require urgent attention and prompt remediation to mitigate potential damage and ensure the system's integrity and reliability.
- **Medium Severity Issues** are significant but not critical vulnerabilities or flaws that can impact the system's security, functionality, or performance. These issues might not pose an immediate threat but have the potential to cause considerable harm if left unaddressed over time. Addressing medium severity issues is important to maintain the overall health and efficiency of the system, though they do not require the same level of urgency as high severity issues.
- **Low Severity Issues** are minor vulnerabilities or flaws that have a limited impact on the system's security, functionality, or performance. These issues generally do not pose a significant risk and can be addressed in the regular maintenance cycle. While low severity issues are not critical, resolving them can help improve the system's overall quality and user experience by preventing the accumulation of minor problems over time.
- **Informational Severity Issues** represent informational findings that do not directly impact the system's security, functionality, or performance. These findings are typically observations or recommendations for potential improvements or optimizations. Addressing info severity issues can enhance the system's robustness and efficiency but is not necessary for the system's immediate operation or security. These issues can be considered for future development or enhancement plans.

Below is a summary of the vulnerabilities with their current status, highlighting the number of issues identified in each severity category and their resolution progress.

	Number	Resolved
High Severity Issues	0	0
Medium Severity Issues	0	0
Low Severity Issues	6	6
Informational Severity Issues	4	4

Findings

[Low] Chainlink's `latestRoundData` Might Return Stale or Incorrect Results

`ChainlinkRateProvider::getRate` does not check the time limit for the returning prices from the oracle. The protocol may get an expired price.

```
//ChainlinkRateProvider.sol
function getRate() external view override returns (uint256) {
  (, int256 price, , , ) = pricefeed.latestRoundData();
  require(price > 0, 'Invalid price rate response');
  return uint256(price) * _scalingFactor;
}
```

Recommendation:

Add timelimit checks.

```
function getRate() external view override returns (uint256) {
  (, int256 price, ,uint256 updateAt , ) = pricefeed.latestRoundData();
  if(updateAt < block.timestamp - 60*60 /** any time */){
    revert("stale price feed");
  }
  require(price > 0, 'Invalid price rate response');
  return uint256(price) * _scalingFactor;
}
```

Status: Acknowledged

[Low] Missing Derivative Limit and Deposit Availability Checks Will Revert The Whole Stake

In the `_lidoStake` and `_lidoStakeAndWrapWETH` functions, ETH is directly converted into derivatives like `stETH` and `wstETH`. However, the Lido protocol enforces a daily staking limit for both `stETH` and `wstETH`, as outlined in their documentation. The current daily limit is set at 150,000 ETH, and the `deposit()` function will revert if this limit is reached.

According to the documentation:

> In order to handle the staking surge in case of some unforeseen market conditions, the Lido protocol implemented staking rate limits aimed at reducing the surge's impact on the staking queue & Lido's socialized rewards distribution model. There is a sliding window limit that is parametrized with `_maxStakingLimit` and `_stakeLimitIncreasePerBlock`. This means it is only possible to submit this much ether to the Lido staking contracts within a 24-hours timeframe. Currently, the daily staking limit is set at 150,000 ether.
>
> You can picture this as a health globe from Diablo 2 with a maximum of `_maxStakingLimit` and regenerating with a constant speed per block. When you deposit ether to the protocol, the level of health is reduced by its amount and the current limit becomes smaller and smaller. When it hits the ground, the transaction gets reverted.
>
> To avoid that, you should check if `getCurrentStakeLimit() >= amountToStake`, and if it's not you can go with an alternative route. The staking rate limits are denominated in ether, thus, it makes no difference if the stake is being deposited for stETH or using the wstETH shortcut, the limits apply in both cases.

This check is not done in the code below.

```
// contracts/strategy/actions/lido/LidoStake.sol
function _lidoStake(Params memory _inputData) internal returns (uint
stEthReceivedAmount, bytes memory logData) {
  TokenUtils.withdrawWeth(_inputData.amount);
  uint stEthBalanceBefore = lidoStEth.getBalance(address(this));
  (bool sent, ) = payable(lidoStEth).call{ value: _inputData.amount }('');
  require(sent, 'Failed to send Ether');
  uint stEthBalanceAfter = lidoStEth.getBalance(address(this));
  stEthReceivedAmount = stEthBalanceAfter - stEthBalanceBefore;
  logData = abi.encode(_inputData, stEthReceivedAmount);
}

// contracts/strategy/actions/lido/LidoWrap.sol
function _lidoStakeAndWrapWETH(Params memory _inputData) internal returns
(uint wStEthReceivedAmount) {
  TokenUtils.withdrawWeth(_inputData.amount);

  uint wStEthBalanceBefore = lidoWrappedStEth.getBalance(address(this));
  (bool sent, ) = payable(lidoWrappedStEth).call{ value: _inputData.amount }
('');
  require(sent, 'Failed to send Ether');
  uint wStEthBalanceAfter = lidoWrappedStEth.getBalance(address(this));

  wStEthReceivedAmount = wStEthBalanceAfter - wStEthBalanceBefore;
}
```

Recommendation:

Check the daily limit via `getCurrentStakeLimit() >= _inputData.amount`

Status: Acknowledged

[Low] Insufficient Validation for Lido Withdrawal

The contract does not correctly validate the withdrawal amounts against the `MAX_STETH_WITHDRAWAL_AMOUNT` and `MIN_STETH_WITHDRAWAL_AMOUNT` limits set by the Lido protocol (WithdrawalQueueERC721 | Lido Docs).

> Each amount in `_amounts` must be greater than or equal to `MIN_STETH_WITHDRAWAL_AMOUNT` and lower than or equal to `MAX_STETH_WITHDRAWAL_AMOUNT`

```
function _lidoWithdraw(Params memory _inputData) internal returns (uint
requestId, bytes memory logData) {
  uint[] memory _amounts = new uint[](1);
  _amounts[0] = _inputData.amount;//@audit

  if (_inputData.useWStEth) {// wstETH
    TokenUtils.approveToken(lidoWrappedStEth, lidoUnStEth,
_inputData.amount);
    requestId = _lidoRequestWithdrawalsWStEth(_amounts)[0];
  } else {// stETH
    TokenUtils.approveToken(lidoStEth, lidoUnStEth, _inputData.amount);
    requestId = _lidoRequestWithdrawals(_amounts)[0];
  }

  logData = abi.encode(_inputData, requestId);
}
```

Recommendation:

Implement checks to ensure withdrawal amounts are within the allowed range before requesting the Lido protocol.

Status: Acknowledged

[Low] Upgradeable Contract Does Not Have `__gap[50]` Storage Variable

To allow for new storage variables in future upgrades in the `TAsset` contract, consider adding the `__gap[50]` variable. See this link for a description of the `__gap[50]` storage variable.

Recommendation:

Add an appropriate storage gap at the end of upgradeable contracts.

Status: Acknowledged

[Low] Missing Strategy Existence Check In `isActionWhitelisted`

The `isActionWhitelisted` function in the `StrategyStorage` contract does not verify if the given `strategy` address exists before checking if an action is whitelisted. This omission could lead to false positives, potentially allowing unauthorized actions to be executed.

```
function isActionWhitelisted(address _strategy, bytes4 _actionId) external
view returns (bool _isActionWhitelisted) {
  _isActionWhitelisted =
parameters[_strategy].whitelistedActions.contains(_actionId);//@audit
}
```

Recommendation:

```
function isActionWhitelisted(address _strategy, bytes4 _actionId) external
view returns (bool _isActionWhitelisted) {
  _isActionWhitelisted = strategies.contains(_strategy) &&
parameters[_strategy].whitelistedActions.contains(_actionId);
}
```

Status: Acknowledged

[Low] Lock When Redeeming Funds

In `TreehouseRedemption::finalizeRedeem`, `_returnAmount` depends on the current base interest rate. The amount returned can increase when the price rises, causing the calculated amount to be greater than `IERC20(_underlying).balanceOf(address(VAULT))`. When a certain asset is in a scenario with low liquidity and rising prices, the asset cannot be withdrawn.

```
function finalizeRedeem(
  uint _redeemIndex
) external nonReentrant whenNotPaused validateRedeem(msg.sender,
_redeemIndex) {
 ...
  uint _assets = IERC4626(TASSET).redeem(_redeem.shares, address(this),
address(this));
  redeeming[msg.sender] -= _redeem.shares;
  totalRedeeming -= _redeem.shares;
  address _underlying = VAULT.getUnderlying();
  uint _returnAmount = _getReturnAmount(_redeem.asset, _redeem.baseRate,
_assets, _getBaseRate());
  ...
  if (_returnAmount > _redeem.asset) revert RedemptionError();
  if (IERC20(_underlying).balanceOf(address(VAULT)) < _returnAmount) revert
InsufficientFundsInVault();
  IInternalAccountingUnit(IAU).burn(_returnAmount);
  ...
```

Recommendation:

To cope with low liquidity situations, we can maintain a liquidity reserve in the contract. This reserve can be used to supplement redemption demand in extreme market conditions.

Status: Acknowledged

[Info] `rateProvider` Lacks Update Validation

There are several potential problems with owners directly updating `rateProvider`

1. The updated `rateProvider` is not verified to be legitimate
2. There is a lack of time buffer, and in the event of a single point of account failure, `rateProvider` will be updated immediately, providing the potential for price manipulation to be impaired.

```
function update(address _asset, address _rateProvider) external onlyOwner {
  if (_asset == address(0) || _rateProvider == address(0)) revert
InvalidAddress();
  emit RateProviderUpdated(_asset, _rateProvider, rateProviders[_asset]);
  rateProviders[_asset] = _rateProvider;
}
```

Recommendation:

Introduce a time lock mechanism, and add `_rateProvider` legitimacy check.

Status: Acknowledged

[Info] Contract Address May Be Set to Zero Address

In the `ActionRegistry` contract, the `startContractChange` function allows setting a new contract address to the zero address (0x0), and the `approveContractChange` function does not perform an additional zero address check in subsequent operations. This could lead to contract addresses being set to zero address.

```
function startContractChange(bytes4 _id, address _newContractAddr) public
onlyOwner {
  if (!entries[_id].exists) {
    revert EntryNonExistentError(_id);
  }

  entries[_id].inContractChange = true;
  pendingAddresses[_id] = _newContractAddr;

  emit StartContractChange(msg.sender, _id, entries[_id].contractAddr,
_newContractAddr);
}
```

Recommendation:

Add a zero address check in the `startContractChange` function.

Status: Acknowledged

[Info] `ActionExecutor::executeActions` Does Not Check Lengths Of Input Arrays

If the length of `_actionIds` exceeds that of `_actionCallData` and `_paramMapping` , the function will revert. Conversely, if `_actionIds` is shorter than `_actionCallData` and `_paramMapping`, only a portion of the latter two arrays will be processed. This partial execution won't trigger a revert, potentially leading to unnoticed parameter omissions and subsequent issues.

```
function executeActions(
    bytes4[] calldata _actionIds,
    bytes[] calldata _actionCallData,
    uint8[][] calldata _paramMapping
  ) public payable {
    bytes32[] memory returnValues = new bytes32[](_actionCallData.length);
    for (uint i; i < _actionIds.length; ++i) {
      returnValues[i] = _executeAction(_actionIds[i], _actionCallData[i], _pa
ramMapping[i], returnValues);
    }
  }
```

Recommendation:

Add a check to see if the lengths of the three arrays are equal.

Status: Acknowledged

[Info] `StrategyExecutor::executeOnStrategy` Does Not Check Lengths Of Input Arrays

The `StrategyExecutor::executeOnStrategy` function only checks if `_actionCalldata.length` matches `_actionIds.length`, but overlooks verifying the length of `_paramMapping`. As indicated by the comment "`_paramMapping: list of param mappings for actions`", if `_paramMapping`'s length is insufficient or excessive, some operation parameters might be omitted or cause the function to revert.

```
function executeOnStrategy(
  uint _strategyId,
  bytes4[] calldata _actionIds,
  bytes[] calldata _actionCalldata,
  uint8[][] memory _paramMapping
) external payable {
  //......
  if (_actionCalldata.length != _actionIds.length) revert ArrayLengthMismatch
();

  //......

  IStrategy(_stratAddress).callExecute(
    ACTION_EXECUTOR,
    abi.encodeWithSelector(EXECUTE_ACTIONS_SELECTOR, _actionIds, _actionCalld
ata, _paramMapping)
  );

  emit ExecutionEvent(_actionIds, _strategyId);
}
```

Recommendation:

Add a check to see if the lengths of the three arrays are equal.

Status: Acknowledged