Tempus Raft

Security Assessment

May 15, 2023

Prepared for:
Đorđe Mijović
Tempus

Prepared by: **Justin Jacob and Michael Colburn**

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.
228 Park Ave S #80688
New York, NY 10003
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

Executive Summary

Engagement Overview

Tempus engaged Trail of Bits to review the security of its Raft stablecoin. From April 24 to April 28, 2023, a team of two consultants conducted a security review of the client-provided source code, with two person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the target system, including access to the source code and documentation. We performed static testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

The audit did not uncover any significant flaws or defects that could impact system confidentiality, integrity, or availability.

EXPOSURE ANALYSIS

Severity	*Count*
Medium	3
Low	1
Informational	4

CATEGORY BREAKDOWN

Category	*Count*
Configuration	2
Data Validation	4
Denial of Service	1
Undefined Behavior	1

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Anne Marie Barry, Project Manager
annemarie.barry@trailofbits.com

The following engineers were associated with this project:

Justin Jacob, Consultant
justin.jacob@trailofbits.com

Michael Colburn, Consultant
michael.colburn@trailofbits.com

Project Timeline

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

Date	Event
April 21, 2023	Pre-project kickoff call
May 1, 2023	Delivery of report draft
May 1, 2023	Report readout meeting
May 15, 2023	Delivery of final report

Project Goals

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

- Can a malicious user prevent liquidations or force collateralized positions to be liquidated?
- How does the system handle oracle usage and validate returndata?
- Are the arithmetic calculations computed correctly and robust against precision loss?
- Can a malicious user create insolvent positions?
- Can an attacker steal funds from the protocol?
- Does the system correctly manage the internal accounting when funds enter and exit the system?

Project Targets

The engagement involved a review and testing of the following target.

Raft contracts

Repository	https://github.com/tempusfinance/raft-contracts
Version	`615bb30f190549bced2bcfcc60227fea9231a74e`
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 and their results include the following:

- **The `MathUtils` library**. This library is used throughout the codebase and contains all the mathematical operations necessary to compute position management, liquidations, and the base rate. We focused on finding any rounding errors and approximation errors, as well as verifying the constants and operations for correctness. This led to the discovery of TOB-RAFT-3.

- **Oracle usage and the `PriceFeed` contract**. The Chainlink and Tellor oracles are used throughout the codebase to calculate the prices of assets and liabilities. We investigated how oracle data is consumed by the `PriceFeed` contract, how the data is validated, and how secondary oracles are implemented in the codebase. We also investigated how these oracles retrieve data (TOB-RAFT-6) and how robust the system is against oracle outages. This led to the discovery of TOB-RAFT-2.

- **The `ERC20Indexable` contracts**. These contracts are used for internal accounting purposes and to keep track of a user's collateral and debt. When a user enters or exits a position, a corresponding amount of `raftDebtToken` and `raftCollateralTokens` are minted or burned accordingly. We reviewed the access controls on the `ERC20Indexable` contracts and their ERC-20 conformance.

- **The `PositionManager` and `SplitLiquidationCollateral` contracts**. The core of the system lies in the `PositionManager` contract, which contains the logic responsible for creating a position, borrowing further, adding collateral, and liquidating insolvent users. The `SplitLiquidationCollateral` contract contains logic for computing rewards during liquidations and redistributions. We manually reviewed the logic used when a position is created, when debt and collateral are added or removed from the system, and when positions are closed, focusing on ways a malicious user could bypass a collateral transfer or leave the system with bad debt. We reviewed the arithmetic used to compute collateral and debt adjustment, as well as the calculations used to compute rewards during liquidations and protocol redistributions. This led to the discovery of TOB-RAFT-8. In addition, we compared the documentation regarding the initial configuration of the system and the corresponding implementation in the code, which led to the discovery of TOB-RAFT-7. Finally, we reviewed the implications of invalid positions being created, which led to the discovery of TOB-RAFT-5.

- **The `OneStepLeverage` contract**. This contract allows a user to flash mint R tokens and swap them for collateral tokens on an automated market maker (AMM), effectively giving a user a temporary leveraged position. We manually reviewed the potential state changes possible when managing a leveraged position and investigated whether a malicious user could create a denial of service, steal funds, or avoid repaying the flash loan.

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:

- Integrations with the AMM and its respective callback as used by the `OneStepLeverage` contract
- Dynamic and end-to-end testing of the system using Echidna

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	All the arithmetic in the codebase is checked for overflow by default with Solidity v0.8. Mathematical functions are abstracted into a separate library to allow easy reuse. The system would benefit from additional fuzz and differential testing of this library.	**Satisfactory**
Auditing	All important state changes in the contracts emit events to facilitate off-chain monitoring. However, it is unclear whether the Tempus organization has off-chain monitoring systems in place or an incident response plan in case of an unexpected event.	**Moderate**
Authentication / Access Controls	Appropriate access controls are in place for all privileged operations. Additional documentation describing these roles (i.e., each contract's owner and the delegate mechanism) and their capabilities should be added.	**Satisfactory**
Complexity Management	The codebase is broken down into appropriate components, and the logic is straightforward to understand, relative to what the code does. The code is overall well documented, though NatSpec comments are split between the interface and implementation contracts. The `OneStepLeverage` contract is an exception that would benefit from additional in-line comments.	**Satisfactory**

Decentralization	The Raft protocol documentation describes the initial governance structure for Raft as a 3-of-5 multisignature with capabilities limited to adjusting protocol fees within hard-coded bounds. However, there are additional capabilities that could allow the multisignature to have undue influence on the protocol. The owner of the `PositionManager` contract could add new tokens as collateral with a corresponding new price feed. While the address of a price feed in the `PositionManager` contract cannot be updated, the owner of the `PriceFeed` contract can update the primary or secondary oracle addresses at any time.	**Weak**
Documentation	The protocol has thorough user and developer documentation that includes formulas and valid ranges for system parameters. Additional documentation to describe privileged roles and diagrams similar to the Liquidation Flow flowchart for the other operations would be beneficial. In addition, documentation about oracle usage and the circuit breakers used in the codebase would be beneficial.	**Satisfactory**
Front-Running Resistance	Because of the complex nature of transaction reordering on a blockchain, we did not have sufficient time to investigate the implications of front-running and its potential risks throughout the codebase. Further investigation is required to understand the consequences of front-running.	**Further Investigation Required**
Low-Level Manipulation	The codebase does not include any in-line assembly, uses low-level calls only when converting ETH to wstETH, and performs the necessary checks when doing so.	**Satisfactory**
Testing and Verification	The protocol would benefit from more in-depth testing of arithmetic operations, both through fuzz and differential testing, as well as integration tests incorporating real oracles and AMMs.	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	Solidity compiler optimizations can be problematic	Undefined Behavior	**Informational**
2	Issues with Chainlink oracle's return data validation	Data Validation	**Low**
3	Incorrect constant for 1000-year periods	Configuration	**Informational**
4	Inconsistent use of safeTransfer for collateralToken	Data Validation	**Medium**
5	Tokens may be trapped in an invalid position	Denial of Service	**Informational**
6	Price deviations between stETH and ETH may cause Tellor oracle to return an incorrect price	Data Validation	**Informational**
7	Incorrect constant value for MAX_REDEMPTION_SPREAD	Configuration	**Medium**
8	Liquidation rewards are calculated incorrectly	Data Validation	**Medium**

Detailed Findings

1. Solidity compiler optimizations can be problematic	
Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-RAFT-1
Target: `foundry.toml`	

Description

The Raft Finance contracts have enabled compiler optimizations. There have been several optimization bugs with security implications. Additionally, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild use them, so how well they are being tested and exercised is unknown.

High-severity security issues due to optimization bugs have occurred in the past. For example, a high-severity bug in the emscripten-generated solc-js compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity `CHANGELOG`. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity v0.5.6. More recently, a bug due to the incorrect caching of Keccak-256 was reported.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe. It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Exploit Scenario

A latent or future bug in Solidity compiler optimizations causes a security vulnerability in the Raft Finance contracts.

Recommendations

Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

2. Issues with Chainlink oracle's return data validation	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-RAFT-2
Target: `contracts/Oracles/ChainlinkPriceOracle.sol`	

Description
Chainlink oracles are used to compute the price of a collateral token throughout the protocol. When validating the oracle's return data, the returned price is compared to the price of the previous round.

However, there are a few issues with the validation:

- The increase of the `currentRoundId` value may not be statically increasing across rounds. The only requirement is that the `roundID` increases monotonically.
- The `updatedAt` value in the oracle response is never checked, so potentially stale data could be coming from the `priceAggregator` contract.
- The `roundId` and `answeredInRound` values in the oracle response are not checked for equality, which could indicate that the answer returned by the oracle is fresh.

```
function _badChainlinkResponse(ChainlinkResponse memory response) internal view
returns (bool) {
    return !response.success || response.roundId == 0 || response.timestamp == 0
        || response.timestamp > block.timestamp || response.answer <= 0;
}
```

Figure 2.1: The Chainlink oracle response validation logic

Exploit Scenario
The Chainlink oracle attempts to compare the current returned price to the price in the previous `roundID`. However, because the `roundID` did not increase by one from the previous round to the current round, the request fails, and the price oracle returns a failure. A stale price is then used by the protocol.

Recommendations
Short term, have the code validate that the `timestamp` value is greater than `0` to ensure that the data is fresh. Also, have the code check that the `roundID` and `answeredInRound` values are equal to ensure that the returned answer is not stale. Lastly check that the `timestamp` value is not decreasing from round to round.

Long term, carefully investigate oracle integrations for potential footguns in order to conform to correct API usage.

References

- The Historical-Price-Feed-Data Project

3. Incorrect constant for 1000-year periods

Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-RAFT-3
Target: `contracts/Dependencies/MathUtils.sol`	

Description

The Raft finance contracts rely on computing the exponential decay to determine the correct base rate for redemptions. In the `MathUtils` library, a period of 1000 years is chosen as the maximum time period for the decay exponent to prevent an overflow. However, the `_MINUTES_IN_1000_YEARS` constant used is currently incorrect:

```
/// @notice Number of minutes in 1000 years.
uint256 internal constant _MINUTES_IN_1000_YEARS = 1000 * 356 days / 1 minutes;
```

Figure 3.1: The declaration of the `_MINUTES_IN_1000_YEARS` constant

Recommendations

Short term, change the code to compute the `_MINUTES_IN_1000_YEARS` constant as `1000 * 365 days / 1 minutes`.

Long term, improve unit test coverage to uncover edge cases and ensure intended behavior throughout the system. Integrate Echidna and smart contract fuzzing in the system to triangulate subtle arithmetic issues.

4. Inconsistent use of safeTransfer for collateralToken

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-RAFT-4
Target: `PositionManager.sol`, `PositionManagerStETH.sol`	

Description

The Raft contracts rely on ERC-20 tokens as collateral that must be deposited in order to mint R tokens. However, although the `SafeERC20` library is used for collateral token transfers, there are a few places where the `safeTransfer` function is missing:

- The transfer of `collateralToken` in the `liquidate` function in the `PositionManager` contract:

```
if (!isRedistribution) {
    rToken.burn(msg.sender, entirePositionDebt);
    _totalDebt -= entirePositionDebt;
    emit TotalDebtChanged(_totalDebt);

    // Collateral is sent to protocol as a fee only in case of liquidation
    collateralToken.transfer(feeRecipient, collateralLiquidationFee);
}

collateralToken.transfer(msg.sender, collateralToSendToLiquidator);
```

Figure 4.1: Unchecked transfers in `PositionManager.liquidate`

- The transfer of stETH in the `managePositionStETH` function in the `PositionManagerStETH` contract:

```
{
    if (isCollateralIncrease) {
        stETH.transferFrom(msg.sender, address(this), collateralChange);
        stETH.approve(address(wstETH), collateralChange);
        uint256 wstETHAmount = wstETH.wrap(collateralChange);
        _managePosition(
            ...
        );
    } else {
        _managePosition(
            ...
        );
        uint256 stETHAmount = wstETH.unwrap(collateralChange);
        stETH.transfer(msg.sender, stETHAmount);
    }
}
```

Figure 4.2: Unchecked transfers in PositionManagerStETH.managePositionStETH

Exploit Scenario
Governance approves an ERC-20 token that returns a Boolean on failure to be used as collateral. However, since the return values of this ERC-20 token are not checked, Alice, a liquidator, does not receive any collateral for performing a liquidation.

Recommendations
Short term, use the `SafeERC20` library's `safeTransfer` function for the `collateralToken`.

Long term, improve unit test coverage to uncover edge cases and ensure intended behavior throughout the protocol.

5. Tokens may be trapped in an invalid position	
Severity: **Informational**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-RAFT-5
Target: `PositionManager.sol`	

Description

The Raft finance contracts allow users to take out positions by depositing collateral and minting a corresponding amount of R tokens as debt. In order to exit a position, a user must pay back their debt, which allows them to receive their collateral back. To check that a position is closed, the `_managePosition` function contains a branch that validates that the position's debt is zero after adjustment. However, if the position's debt is zero but there is still some collateral present even after adjustment, then the position is considered invalid and cannot be closed. This could be problematic, especially if some dust is present in the position after the collateral is withdrawn.

```
if (positionDebt == 0) {
    if (positionCollateral != 0) {
        revert InvalidPosition();
    }
    // position was closed, remove it
    _closePosition(collateralToken, position, false);
} else {
    _checkValidPosition(collateralToken, positionDebt, positionCollateral);

    if (newPosition) {
        collateralTokenForPosition[position] = collateralToken;
        emit PositionCreated(position);
    }
}
```

Figure 5.1: A snippet from the `_managePosition` function showing that a position with no debt cannot be closed if any amount of collateral remains

Exploit Scenario

Alice, a borrower, wants to pay back her debt and receive her collateral in exchange. However, she accidentally leaves some collateral in her position despite paying back all her debt. As a result, her position cannot be closed.

Recommendations

Short term, if a position's debt is zero, have the `_managePosition` function refund any excess collateral and close the position.

Long term, carefully investigate potential edge cases in the system and use smart contract fuzzing to determine if those edge cases can be realistically reached.

6. Price deviations between stETH and ETH may cause Tellor oracle to return an incorrect price

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-RAFT-6
Target: `TellorPriceOracle.sol`	

Description

The Raft finance contracts rely on oracles to compute the price of the collateral tokens used throughout the codebase. If the Chainlink oracle is down, the Tellor oracle is used as a backup. However, the Tellor oracle does not use the stETH/USD price feed. Instead it uses the ETH/USD price feed to determine the price of stETH. This could be problematic if stETH depegs, which can occur during black swan events.

```
function _getCurrentTellorResponse() internal view returns (TellorResponse memory
tellorResponse) {
    uint256 count;
    uint256 time;
    uint256 value;

    try tellor.getNewValueCountbyRequestId(ETHUSD_TELLOR_REQ_ID) returns (uint256
count_) {
        count = count_;
    } catch {
        return (tellorResponse);
    }
```

Figure 6.1: The Tellor oracle fetching the price of ETH to determine the price of stETH

Exploit Scenario

Alice has a position in the system. A significant black swan event causes the depeg of staked Ether. As a result, the Tellor oracle returns an incorrect price, which prevents Alice's position from being liquidated despite being eligible for liquidation.

Recommendations

Short term, carefully monitor the Tellor oracle, especially during any sort of market volatility.

Long term, investigate the robustness of the oracles and document possible circumstances that could cause them to return incorrect prices.

7. Incorrect constant value for MAX_REDEMPTION_SPREAD

Severity: **Medium**	Difficulty: **Low**
Type: Configuration	Finding ID: TOB-RAFT-7
Target: `PositionManager.sol`	

Description

The Raft protocol allows a user to redeem their R tokens for underlying wstETH at any time. By doing so, the protocol ensures that it maintains overcollateralization. The redemption spread is part of the redemption rate, which changes based on the price of the R token to incentivize or disincentivize redemption. However, the documentation says that the maximum redemption spread should be 100% and that the protocol will initially set it to 100%.

In the code, the `MAX_REDEMPTION_SPREAD` constant is set to 2%, and the `redemptionSpread` variable is set to 1% at construction. This is problematic because setting the rate to 100% is necessary to effectively disable redemptions at launch.

```
uint256 public constant override MIN_REDEMPTION_SPREAD = MathUtils._100_PERCENT /
10_000 * 25; // 0.25%
uint256 public constant override MAX_REDEMPTION_SPREAD = MathUtils._100_PERCENT /
100 * 2; // 2%
```

Figure 7.1: Constants specifying the minimum and maximum redemption spread percentages

```
constructor(ISplitLiquidationCollateral newSplitLiquidationCollateral)
FeeCollector(msg.sender) {
    rToken = new RToken(address(this), msg.sender);
    raftDebtToken = new ERC20Indexable(
        address(this),
        string(bytes.concat("Raft ", bytes(IERC20Metadata(address(rToken)).name()),
" debt")),
        string(bytes.concat("r", bytes(IERC20Metadata(address(rToken)).symbol()),
"-d"))
    );
    setRedemptionSpread(MathUtils._100_PERCENT / 100);
    setSplitLiquidationCollateral(newSplitLiquidationCollateral);

    emit PositionManagerDeployed(rToken, raftDebtToken, msg.sender);
}
```

Figure 7.2: The redemption spread being set to 1% instead of 100% in the `PositionManager`'s `constructor`

Exploit Scenario

The protocol sets the redemption spread to 2%. Alice, a borrower, redeems her R tokens for some underlying wstETH, despite the developers' intentions. As a result, the stablecoin experiences significant volatility.

Recommendations

Short term, set the `MAX_REDEMPTION_SPREAD` value to 100% and set the `redemptionSpread` variable to `MAX_REDEMPTION_SPREAD` in the `PositionManager` contract's constructor.

Long term, improve unit test coverage to identify incorrect behavior and edge cases in the protocol.

8. Liquidation rewards are calculated incorrectly

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-RAFT-8
Target: `SplitLiquidationCollateral.sol`	

Description

Whenever a position's collateralization ratio falls between 100% and 110%, the position becomes eligible for liquidation. A liquidator can pay off the position's total debt to restore solvency. In exchange, the liquidator receives a liquidation reward for removing bad debt, in addition to the amount of debt the liquidator has paid off. However, the calculation performed in the split function is incorrect and does not reward the liquidator with the `matchingCollateral` amount of tokens:

```
function split(
    uint256 totalCollateral,
    uint256 totalDebt,
    uint256 price,
    bool isRedistribution
)
    external
    pure
    returns (uint256 collateralToSendToProtocol, uint256
collateralToSentToLiquidator)
{
    if (isRedistribution) {
        ...
    } else {
        uint256 matchingCollateral = totalDebt.divDown(price);
        uint256 excessCollateral = totalCollateral - matchingCollateral;
        uint256 liquidatorReward =
excessCollateral.mulDown(_calculateLiquidatorRewardRate(totalDebt));
        collateralToSendToProtocol = excessCollateral - liquidatorReward;
        collateralToSentToLiquidator = liquidatorReward;
    }
}
```

Figure 8.1: The calculations for how to split the collateral between the liquidator and the protocol, showing that the `matchingCollateral` is omitted from the liquidator's reward

Exploit Scenario

Alice, a liquidator, attempts to liquidate an insolvent position. However, upon liquidation, she receives only the `liquidationReward` amount of tokens, without the `matchingCollateral`. As a result her liquidation is unprofitable and she has lost funds.

Recommendations

Short term, have the code compute the `collateralToSendToLiquidator` variable as `liquidationReward + matchingCollateral`.

Long term, improve unit test coverage to uncover edge cases and ensure intended behavior throughout the protocol.

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
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
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
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

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 specific vulnerabilities. However, they enhance code readability and may prevent the introduction of future vulnerabilities.

- **Fix incorrect comments.** The comment below refers to the secondary oracle, not the primary oracle.

```
// If primary oracle is broken or frozen, both oracles are untrusted, and return
last good price
if (secondaryOracleResponse.isBrokenOrFrozen) {
    return lastGoodPrice;
}
```

Figure C.1: The comment in the `fetchPrice` function (`PriceFeed.sol#L70–L73`)

- **Declare variables once to avoid unnecessary storage reads.**

```
uint256 icr = MathUtils._computeCR(
    raftCollateralTokens[collateralToken].token.balanceOf(position),
raftDebtToken.balanceOf(position), price
);
if (icr >= MathUtils.MCR) {
    revert NothingToLiquidate();
}

uint256 entirePositionDebt = raftDebtToken.balanceOf(position);
uint256 entirePositionCollateral =
raftCollateralTokens[collateralToken].token.balanceOf(position);
```

Figure C.2: The multiple storage reads in the `liquidate` function (`PositionManager.sol#L175–L183`)

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 May 10, 2022, Trail of Bits reviewed the fixes and mitigations implemented by the Raft team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the eight issues described in this report, Tempus has resolved seven issues and has not resolved the remaining issue. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Solidity compiler optimizations can be problematic	Informational	Unresolved
2	Issues with Chainlink oracle's return data validation	Low	Resolved
3	Incorrect constant for 1000-year periods	Informational	Resolved
4	Inconsistent use of safeTransfer for collateralToken	Medium	Resolved
5	Tokens may be trapped in an invalid position	Informational	Resolved
6	Price deviations between stETH and ETH may cause Tellor oracle to return an incorrect price	Informational	Resolved
7	Incorrect constant value for MAX_REDEMPTION_SPREAD	Medium	Resolved
8	Liquidation rewards are calculated incorrectly	Medium	Resolved

Detailed Fix Review Results

TOB-RAFT-1: Solidity compiler optimizations can be problematic
Unresolved.

TOB-RAFT-2: Issues with Chainlink oracle's return data validation
Resolved in PR #281. The Raft team added checks to catch whether the `roundID` and `answeredInRound` values do not match, as well as additional validation of the `timestamp` response. These checks cover cases of invalid responses from Chainlink. However, the validation logic still assumes `roundIDs` always increment by 1 between valid rounds. This is not guaranteed to be true, especially when the underlying aggregator is updated (i.e., when the `phaseID` in the proxy, which is incorporated into the most significant bytes of the `roundID`, is incremented). This would result in the `PriceFeed` temporarily falling back to the secondary oracle until the next round data is available from the Chainlink oracle, despite receiving a valid response. The infrequency of Chainlink upgrades and graceful oracle fallback and recovery make it unlikely that this edge case will impact system availability.

TOB-RAFT-3: Incorrect constant for 1000-year periods
Resolved in PR #275. The constant was updated to the correct value.

TOB-RAFT-4: Inconsistent use of safeTransfer for collateralToken
Resolved in PR #265. The `PositionManager` contract has been updated to use the `safeERC20` library's `safeTransfer` function for `collateralToken` transfers. Calls to `stETH.transferFrom` were not updated, but this is not necessary because the contract is specific to stETH and its semantics are known.

TOB-RAFT-5: Tokens may be trapped in an invalid position
Resolved in PR #264 and PR #267. The `managePosition` function (name altered during fix review) now correctly closes a position when all the debt is repaid.

TOB-RAFT-6: Price deviations between stETH and ETH may cause Tellor oracle to return an incorrect price
Resolved in PR #279. The Tellor oracle has been updated to fetch the stETH price directly instead of assuming stETH/ETH parity.

TOB-RAFT-7: Incorrect constant value for MAX_REDEMPTION_SPREAD
Resolved in PR #263. The constants for the redemption spread have been updated to the correct values.

TOB-RAFT-8: Liquidation rewards are calculated incorrectly
Resolved in PR #246. Liquidations now correctly return the matched collateral and the liquidator reward to the liquidator.