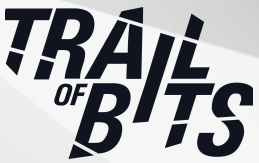

CAP Labs Covered Agent Protocol

Security Assessment

May 15, 2025

Prepared for:
CAP Labs

Prepared by: **Benjamin Samuels, Priyanka Bose, and Nicolas Donboly**

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 and 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 X 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

Trail of Bits performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan.

Security assessment projects are time-boxed and often rely on information 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 software controls and security properties. 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. A project's time and resource constraints also limit their use.

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineering director was associated with this project:

Jim Miller, Engineering Director, Blockchain and Cryptography
james.miller@trailofbits.com

The following consultants were associated with this project:

Benjamin Samuels, Consultant
benjamin.samuels@trailofbits.com

Priyanka Bose, Consultant
priyanka.bose@trailofbits.com

Nicolas Donboly, Consultant
nicolas.donboly@trailofbits.com

Project Timeline

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

Date	Event
February 21, 2025	Pre-project kickoff call
March 7, 2025	Status update meeting #1
March 14, 2025	Status update meeting #2
March 31, 2025	Delivery of report draft
March 31, 2025	Report readout meeting
April 10, 2025	Delivery of final comprehensive report
May 15, 2025	Fix review completed and delivered

Executive Summary

Engagement Overview

CAP Labs engaged Trail of Bits to review the security of their Covered Operator Protocol, a decentralized yield-bearing stablecoin protocol powered by lending markets and shared security models. The protocol enables allowlisted entities with proprietary yield-generating strategies to borrow from supplied assets, providing scalable yield regardless of market conditions.

A team of three consultants conducted the review from March 3 to March 28, 2025, for a total of nine engineer-weeks of effort. Our testing efforts focused on the protocol's borrowing and lending mechanisms, yield distribution, auction functionality, and shared security model. With full access to source code and technical documentation, we performed static and dynamic testing of the codebase, using both automated and manual processes.

Observations and Impact

Our review identified critical vulnerabilities that could significantly impact the protocol's security and economic stability:

- **Interest rate and pricing mechanisms**: We found several high-severity issues related to oracle functionality, including incorrect oracle staleness periods and price calculation errors that could lead to inaccurate asset pricing or denial of service (DoS).
- **Liquidation and collateral management**: We identified serious issues in the liquidation and collateral calculations where recent deposits may improve an agent's health factor but remain protected from slashing during liquidation, creating exploitable gaps.
- **Balance tracking and asset management**: The codebase contains inconsistent balance tracking that could cause the system to enter invalid states where assets can no longer be borrowed.
- **Fee auction vulnerabilities**: The fee auction system contains several vulnerabilities that allow attackers to purchase assets at below-market prices or execute front-running attacks.
- **Reward distribution**: The reward distribution mechanism enables front-running attacks where actors can add deposits just before repayment to capture rewards without having taken equivalent risk.

Recommendations

Based on the findings identified during the security review, Trail of Bits recommends that CAP Labs take the following steps before deployment:

- **Remediate the findings disclosed in this report.** These vulnerabilities expose the protocol and its users to significant risk, including potential fund loss, economic manipulation, and DoS conditions.
- **Implement comprehensive validation for all critical parameters.** Many issues stem from inadequate validation of function inputs, contract addresses, and parameter values. All inputs to critical functions should undergo thorough validation.
- **Strengthen accounting mechanisms.** Address the inconsistencies in balance tracking and implement robust accounting that maintains essential invariants, particularly for borrowing, liquidation, and reward distribution.
- **Expand test coverage with unit tests, then invariant-based tests.** Develop comprehensive tests that verify that critical system invariants are maintained across all operations, focusing on interactions between different components. See appendix D for more details.
- **Obtain an additional review**. We believe there are more high-severity issues present in the codebase that were not discovered during this engagement. Given the number of systemic high-severity findings and the likely presence of other high-severity issues, we recommend pursuing an additional security review after implementing the above mitigations.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	7
Medium	8
Low	6
Informational	8
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Auditing and Logging	1
Configuration	6
Data Validation	17
Denial of Service	1
Timing	3
Undefined Behavior	1

Project Goals

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

- Are the borrowing and lending mechanisms secure against manipulation, and do they properly maintain accounting invariants throughout the protocol's operations?
- Is the collateral-based lending system properly implemented to ensure that agents cannot borrow more than their delegated collateral allows?
- Are the liquidation mechanics correctly implemented to protect the protocol against undercollateralized positions while treating agents fairly?
- Is the yield distribution to stakers implemented securely, and are there any vulnerabilities in the reward mechanisms?
- Are the fee auction mechanisms secure against manipulation, and do they appropriately distribute fees to stakeholders?
- Is the oracle implementation resistant to manipulation, ensuring accurate pricing information?
- Does the shared security model with Symbiotic properly protect the protocol in case of agent defaults?
- Are there any vulnerabilities in the protocol's integration with external protocols and ERC4626 vaults?

Project Targets

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

Covered Agent Protocol

Repository	https://github.com/cap-labs-dev/cap-contracts
Commit	`56819219e935470271a03d4e3415fcbe810e6b0b`
Type	Solidity
Platform	EVM

Project Coverage

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

- **Static analysis of smart contracts**: We analyzed the codebase using automated tools such as Slither to identify common security vulnerabilities and code quality issues.
- **Manual code review**: We performed a detailed manual review of the codebase, focusing on the critical components including the lending pool, vaults, delegation mechanisms, and fee auctions.
- **Economic analysis**: We evaluated the economic mechanisms of the protocol, with particular focus on the fee auction system, yield distribution, and liquidation incentives.
- **Invariant identification and validation**: We identified critical system invariants and examined whether they are maintained across different protocol operations.
- **Integration assessment**: We evaluated the protocol's interactions with external systems, particularly the Symbiotic security model and external yield-generating vaults.

Vault Asset Management and Token Mechanics

- **Vault contract**. The core contract manages ERC20 backing assets and CAP tokens (cUSD), enabling mint/burn/redeem operations and serving as a liquidity source for borrowing. During the review, we investigated issues related to asset validation, slippage protection, and vault investment mechanisms. We found issues related to inconsistent balance tracking that led to asset borrowing DoS (TOB-CAP-5) and unsafe asset removal without borrow validation (TOB-CAP-6).
- **Minter**. This component includes contracts that implement asset mint/burn mechanisms with dynamic fee slopes designed to maintain optimal asset allocation throughout the system. We conducted an evaluation of potential fee manipulation vectors, parameter validation processes, and price computation accuracy. Our analysis resulted in a low-severity issue where interest rates could be manipulated through strategically timed frequent mint and burn operations (TOB-CAP-27)..
- **FractionalReserve**. This component includes contracts that manage idle capital by investing in external yield-generating vaults like Yearn. We analyzed investment/divestment logic, loss tracking mechanisms, and reserve threshold enforcement. Our review uncovered a significant medium-severity issue where

unaccounted external vault investment losses could create withdrawal shortfalls for users (TOB-CAP-16).

- **StakedCap token**. This ERC4626 vault allows users to stake cUSD to earn protocol yield with linear vesting. We evaluated the profit locking mechanism, reward distribution, and manipulation vectors. Our analysis uncovered issues related to yield distribution DoS through timer manipulation (TOB-CAP-4) and reward distribution front-running vulnerabilities (TOB-CAP-29).

Lending and Financial Operations

- **Lender**. This contract manages loans, interest calculations, and liquidations for agents. We examined borrowing limits, repayment logic, and liquidation thresholds. Our analysis Identified critical health calculation/slashable collateral discrepancies (TOB-CAP-14), and liquidation mechanism disabling risks (TOB-CAP-18).

- **Debt tokens**. These three token types track different aspects of debt (principal, interest, restaker). We assessed interest accrual, rounding behavior, and debt repayment priority.

- **FeeAuction**. This contract facilitates the conversion of excess yield into cUSD through reverse Dutch auctions. We investigated auction dynamics, price manipulation vectors, and potential auction sniping exploits. Our analysis revealed several vulnerabilities, including missing input validation that allows payment without asset transfer (TOB-CAP-1), two high-severity vulnerabilities where attackers can manipulate auction proceeds by adding tokens to the auction contract immediately before purchase (TOB-CAP-9), and a front-running attack vector where victims pay auction prices but receive nothing (TOB-CAP-13)

- **Oracle system**. Provides price and interest rate data through external adapters. We evaluated staleness handling, price validation, and rate calculations. We found issues related to incorrect staleness periods causing DoS (TOB-CAP-2), unvalidated vault addresses allowing interest rate manipulation (TOB-CAP-12), and decimal errors in price calculations (TOB-CAP-17).

Delegation, Agent Management, and Cross-Protocol Interactions

- **Delegation**. This component includes contracts that manage the shared security model where restakers provide collateral to back Agent borrowing. It handles critical agent management functions including agent registration, permission management, liquidation threshold configuration, and network relationship tracking. We looked for issues related to delegation tracking, network registration, and coverage calculations. Our review identified two significant issues: a medium-severity vulnerability allowing vaults to be added to the middleware multiple times, resulting

in delegations being double-counted (TOB-CAP-3), and a vulnerability where invalid network registration could cause DoS for affected agents (TOB-CAP-21).

- **Symbiotic Integration**. This set of contracts connect CAP Agents to Operators in Symbiotic through slashing and reward mechanisms. We analyzed delegation verification, slashing timing, and reward distribution. Our analysis uncovered Identified issues related to reward distribution front-running attacks (TOB-CAP-15) and epoch-based timing vulnerabilities in the slashing model.

- **Cross-Chain Components**. These contracts facilitate bridging and zapping capabilities between chains. We examined the token approval handling and cross-chain message composition.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

- Due to the high number of reported issues, we had to spend much more time reporting instead of reviewing code. Due to these time constraints, this review was less thorough than is normally desired for a manual review. After implementing the recommended mitigations and testing changes, we recommend obtaining another review.

- We could not thoroughly review the system's full end-to-end functionality and cross-chain functionality due to time constraints.

- We did not review the system's LayerZero/EigenLayer integrations because they were out of scope.

- We spent limited time on reviewing potential rounding issues due to time constraints.

- The system's MEV risks (front-running, back-running, and sandwiching) received limited review due to time constraints.

- The system's production-time security is critically dependent on how it is deployed and configured. However, the deployment scripts and access control configurations were out of scope. We recommend explicitly scoping these components as part of the next review.

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
Authentication / Access Controls	The protocol implements a granular access control mechanism that allows fine-grained control over function access. However, several functions lack proper validation of external addresses and input parameters.	**Satisfactory**
Complexity Management	The codebase shows significant complexity in its interrelated components, particularly in the delegation, liquidation, and interest accrual systems. Some core invariants are difficult to track across contract interactions, increasing the risk of subtle bugs like those described in TOB-CAP-3, TOB-CAP-5, and TOB-CAP-14.	**Weak**
Cryptography and Key Management	The protocol does not implement custom cryptographic operations beyond what is provided by underlying EVM mechanisms.	**Not Applicable**
Decentralization	The protocol has governance mechanisms through role-based access control, but relies heavily on admin functions for critical operations. The shared security model with Symbiotic provides some decentralization benefits, but governance remains centralized and all funds are at stake if the admin wallet is compromised. Since we did not review CAP's operational security practices as part of this audit, we believe further investigation is required to provide a meaningful measurement.	**Further Investigation Required**
Documentation	The codebase features comprehensive NatSpec documentation for most functions and contracts. Technical documentation clearly explains the purpose and interactions of major components.	**Satisfactory**

Low-Level Manipulation	Assembly is used in a limited and well-justified manner, primarily in storage access patterns following ERC-7201. No dangerous low-level manipulations were identified.	**Strong**
Testing and Verification	CAP's unit test suite coverage was measured at ~80%. However, the test suite contains multiple material weaknesses, such as limited testing of edge case handling, parameter validation, oracle calculations, integration testing between components, and testing of the code's unhappy paths. Multiple findings including TOB-CAP-1, TOB-CAP-11, TOB-CAP-12, and TOB-CAP-13 may have been discoverable with sufficient unit testing before the audit. While there is an invariant testing suite present, findings that are easily discoverable using invariant testing, such as TOB-CAP-5, indicate that the invariant test suite needs additional work.	**Weak**
Transaction Ordering	The protocol has several vulnerabilities related to transaction ordering, including front-running opportunities in the fee auction and reward distribution mechanisms. The codebase lacks sufficient protection against these threats.	**Weak**

Summary of Findings

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

ID	Title	Type	Severity
1	Missing input validation in FeeAuction.buy allows payment without asset transfer	Data Validation	Medium
2	Incorrect oracle staleness period leads to price feed DoS	Configuration	High
3	Vaults can be added to the middleware multiple times, leading to double-counted delegations	Configuration	Medium
4	StakedCap yield distribution DoS through timer manipulation	Timing	Medium
5	Inconsistent balance tracking in vault creates DoS for asset borrowing	Denial of Service	High
6	Unsafe asset removal without borrow validation	Data Validation	Medium
7	Initial auction start price can be lower than minimum start price	Configuration	Low
8	Immediate liquidation possible when reducing an agent's liquidation threshold	Data Validation	Informational
9	FeeAuction's buy function allows purchasing at sub-optimal prices by adding tokens during an active auction	Undefined Behavior	Low
10	Missing event emissions for critical parameter changes in VaultAdapter	Auditing and Logging	Informational
11	VaultAdapter's setSlopes function permits zero or maximum kink values that cause division by zero	Data Validation	Informational

12	Unvalidated _vault address in VaultAdapter allows interest rate manipulation	Data Validation	**High**
13	Fee auction allows buying zero assets, leading to front-running attacks	Data Validation	**High**
14	Discrepancy between health calculation and slashable collateral computation	Data Validation	**High**
15	Reward distribution enables front-running attacks and reward siphoning	Data Validation	**High**
16	Unaccounted external vault investment losses can create withdrawal shortfalls	Data Validation	**Medium**
17	Wrong capTokenDecimals value used in StakedCapAdapter.price causes inaccurate prices	Data Validation	**High**
18	Liquidation mechanism can be permanently disabled by misconfigured grace and expiry periods	Data Validation	**Medium**
19	Oracle update front-running allows extraction of value from vaults	Timing	**Medium**
20	Asset removal does not reset isBorrowing flag for agents	Data Validation	**Informational**
21	Invalid network registration in Delegation.registerNetwork can cause DoS	Configuration	**Low**
22	Lack of verification on ERC4626 vault withdrawal amounts	Data Validation	**Informational**
23	Agent LTV can be configured equal to or higher than liquidation threshold	Data Validation	**Low**
24	ZapOFTComposer._lzCompose may fail with USDT	Data Validation	**Informational**
25	Fee auction allows assets to be purchased for free	Configuration	**Informational**

26	Small borrows can create economically unviable liquidatable positions leading to bad debt accumulation	Data Validation	**Low**
27	Interest rate manipulation through frequent mints and burns	Data Validation	**Low**
28	Protocol lacks bad debt management mechanisms, risking permanent insolvency	Configuration	**Medium**
29	Reward distribution can be tricked by front-running notify calls	Timing	**Informational**

Detailed Findings

1. Missing input validation in FeeAuction.buy allows payment without asset transfer

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-CAP-1
Target: `contracts/feeAuction/FeeAuction.sol`	

Description

The `FeeAuction.buy` function lacks validation for the `_assets` parameter, allowing buyers to pay the auction price without receiving any assets when an empty array or 0 balance token addresses are provided.

```
function buy(address[] calldata _assets, address _receiver, bytes calldata
_callback) external {
    uint256 price = currentPrice();
    FeeAuctionStorage storage $ = get();
    $.startTimestamp = block.timestamp;

    uint256 newStartPrice = price * 2;
    if (newStartPrice < $.minStartPrice) newStartPrice = $.minStartPrice;
    $.startPrice = newStartPrice;

    uint256[] memory balances = _transferOutAssets(_assets, _receiver);

    if (_callback.length > 0) IAuctionCallback(msg.sender).auctionCallback(_assets,
balances, price, _callback);

    IERC20($.paymentToken).safeTransferFrom(msg.sender, $.paymentRecipient, price);

    emit Buy(msg.sender, price, _assets, balances);
}
```

Figure 1.1: contracts/feeAuction/FeeAuction.sol#L64-L80

```
function _transferOutAssets(address[] calldata _assets, address _receiver)
    internal
    returns (uint256[] memory balances)
{
    uint256 assetsLength = _assets.length;
    balances = new uint256[](assetsLength);
    for (uint256 i; i < assetsLength; ++i) {
        address asset = _assets[i];
        balances[i] = IERC20(asset).balanceOf(address(this));
        if (balances[i] > 0) IERC20(asset).safeTransfer(_receiver, balances[i]);
    }
}
```

Figure 1.2: contracts/feeAuction/FeeAuction.sol#L113-L124

Exploit Scenario

Alice sees a fee auction and decides to participate. She calls the buy function but mistakenly provides an empty array for `_assets` or includes token addresses not present in the contract. The transaction succeeds and Alice pays the full auction price, but she receives nothing in return, resulting in a complete loss of her funds.

Recommendations

Short term, add validation to ensure that the `_assets` array is not empty and that all existing assets in the contract have positive balances.

Long term, implement comprehensive input validation and error handling throughout the codebase.

2. Incorrect oracle staleness period leads to price feed DoS

Severity: **High**	Difficulty: **Low**
Type: Configuration	Finding ID: TOB-CAP-2
Target: `contracts/oracle/PriceOracle.sol`	

Description

The `PriceOracle` uses a unique staleness period for all price feeds, despite different assets having varying update frequencies (heartbeats), causing denial of service in price retrieval functions for some assets.

```
/// @dev Initialize unchained
/// @param _staleness Staleness period in seconds for asset prices
function __PriceOracle_init_unchained(uint256 _staleness) internal onlyInitializing
{
    getPriceOracleStorage().staleness = _staleness;
}
```

Figure 2.1: contracts/oracle/PriceOracle.sol#L21-L25

The assets supposed to be used in the CAP Protocol (stablecoins, ETH, BTC) have different heartbeats:

- USDC/USD 86400s
- USDT/USD 86400s
- USDe/USD 86400s
- DAI / USD 3600s
- ETH/USD 3600s
- BTC/USD 3600s

Currently, the staleness check is unique and is set to one hour in the `DeployInfra._deployInfra` function:

```
// init infra instances
AccessControl(d.accessControl).initialize(users.access_control_admin);
Lender(d.lender).initialize(d.accessControl, d.delegation, d.oracle, 1.33e27, 1
hours, 1 days, 0.1e27, 0.91e27);
Oracle(d.oracle).initialize(d.accessControl, 1 hours);
Delegation(d.delegation).initialize(d.accessControl, d.oracle, 1 days);
```

Figure 2.1: `contracts/deploy/service/DeployInfra.sol#L36-L40`

Exploit Scenario

Functions that rely on the `Oracle.getPrice` function fail for assets that have update frequencies different from the configured staleness period. If the staleness period is set to accommodate frequently updated assets (e.g., those that are updated once per hour), the `PriceOracle` will reject valid prices from slower-updating assets (e.g., those that are updated every 24 hours). Conversely, if the staleness period is set for slower assets, the `PriceOracle` will accept potentially stale prices from frequently updating assets.

Recommendations

Short term, modify the Oracle contract to support asset-specific staleness periods.
Long term, implement automated tests that verify oracle configuration against actual Chainlink price feed update frequencies.

Reference

- Price Feed Contract Addresses

3. Vaults can be added to the middleware multiple times, leading to double-counted delegations

Severity: **Medium**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-CAP-3
Target: `contracts/delegation/providers/symbiotic/NetworkMiddleware.sol`	

Description

The `NetworkMiddleware` contract, shown in figure 3.1, does not check whether a vault has been registered previously. This could potentially allow a vault to be registered multiple times and have its delegation counted multiple times for a single agent.

```
function registerVault(address _vault, address _stakerRewarder, address[] calldata
_agents)
    external
    checkAccess(this.registerVault.selector)
{
    _verifyVault(_vault);
    NetworkMiddlewareStorage storage $ = getNetworkMiddlewareStorage();
    $.stakerRewarders[_vault] = _stakerRewarder;
    for (uint256 i; i < _agents.length; ++i) {
        $.vaults[_agents[i]].push(_vault);
    }
    emit VaultRegistered(_vault);
}
```

Figure 3.1: The `registerVault` function does not check whether a vault has been previously registered, leading to duplicate vault entries introduced in the highlighted code. (`contracts/delegation/providers/symbiotic/NetworkMiddleware.sol#58–69`)

Each agent's coverage is calculated using the `coverage` function shown in figure 3.2. This function loops over each agent's vault list and counts the delegation from each one without performing deduplication. This means a vault that has been added to an agent twice will have its delegation for the agent counted twice.

```
function coverage(address _agent) public view returns (uint256 delegation) {
    NetworkMiddlewareStorage storage $ = getNetworkMiddlewareStorage();
    address[] memory _vaults = $.vaults[_agent];
    address _network = $.network;
    address _oracle = $.oracle;
    uint48 _timestamp = uint48(block.timestamp);

    for (uint256 i = 0; i < _vaults.length; i++) {
        (uint256 value,) = coverageByVault(_network, _agent, _vaults[i], _oracle,
_timestamp);
        delegation += value;
    }
}
```

Figure 3.2: The `coverage` function loops over all of the agent's vaults to count delegation without deduplicating the list.
(contracts/delegation/providers/symbiotic/NetworkMiddleware.sol#195–206)

Exploit Scenario

While configuring a new vault, the transaction becomes stuck, so a replacement transaction is erroneously created that uses a successive nonce, causing the vault to be configured twice for the same set of agents.

This causes the stake for the vault to be counted twice for each agent. Later, a liquidation is conducted, but the collateral provided by Symbiotic is not enough to cover the outsized borrow amount, leading to a loss of funds for the cUSD senior tranch.

Recommendations

Short term, modify the `registerVault` code to prevent vaults from being configured twice. If a change is required to add more agents, redundant logic must be in place to prevent duplicate vaults from being added to an agent's vault list.

Long term, use robust data structures like `EnumerableSet` for lists that should contain only unique elements. In addition, this finding could have been detected using an invariant that sums up the delegated power for all agents and ensures that it is less than or equal to the total amount in the Symbiotic vaults.

4. StakedCap yield distribution DoS through timer manipulation

Severity: **Medium**	Difficulty: **Low**
Type: Timing	Finding ID: TOB-CAP-4
Target: `contracts/token/StakedCap.sol`	

Description

The `StakedCap` contract contains a critical vulnerability in its yield distribution mechanism that allows an attacker to indefinitely delay the proper distribution of rewards to legitimate stakers.

The vulnerability stems from the interconnected relationship between three key functions in the `StakedCap` contract: `notify`, `lockedProfit`, and `totalAssets`. As the `notify` function is permissionless, whoever makes a minimal token contribution to the contract can trigger an update to the vesting timer (`$.lastNotify = block.timestamp`), as shown in the figure 4.1.

```
function notify() external {
    uint256 total = IERC20(asset()).balanceOf(address(this));
    StakedCapStorage storage $ = getStakedCapStorage();
    if (total > $.storedTotal) {
        uint256 diff = total - $.storedTotal;
        $.totalLocked = lockedProfit() + diff;
        $.storedTotal = total;
        $.lastNotify = block.timestamp;
        emit Notify(msg.sender, diff);
    }
}
```

Figure 4.1: Code snippet of the `notify` function in the `StakedCap` contract (`contracts/token/StakedCap.sol#L58-L69`)

Next, `notify` calls the `lockedProfit` function, which calculates how much yield remains locked based on the time elapsed since the last notification. Therefore, an attacker constantly resetting `$.lastNotify` ensures that elapsed remains small (figure 4.2), keeping most rewards locked.

```
function lockedProfit() public view returns (uint256 locked) {
    StakedCapStorage storage $ = getStakedCapStorage();
    if ($.lockDuration == 0) return 0;
    uint256 elapsed = block.timestamp - $.lastNotify;
    uint256 remaining = elapsed < $.lockDuration ? $.lockDuration - elapsed : 0;
    locked = $.totalLocked * remaining / $.lockDuration;
}

function totalAssets() public view override returns (uint256 total) {
    total = getStakedCapStorage().storedTotal - lockedProfit();
}
```

Figure 4.2: Code snippets of `lockedProfit` and `totalAssets` functions in the `StakedCap` contract (`contracts/token/StakedCap.sol#L73-L85`)

Since the `totalAssets` function subtracts `lockedProfit` from the total stored balance while computing the assets a user can withdraw, users cannot withdraw the artificially locked portion of their rewards, effectively preventing them from claiming their full rewards.

Exploit Scenario

Alice and other users have staked their CAP tokens in the `StakedCap` contract. The protocol distributes 100,000 tokens as yield, which should vest linearly over next seven days.

Mallory, noticing this distribution, sends 1 wei of the token to the contract and calls the `notify` function, resetting the vesting timer. Mallory repeats this action, resetting the timer again. By continuing this pattern, Mallory ensures that the elapsed time in `lockedProfit` is always minimal, keeping yields locked.

When Alice attempts to withdraw, the manipulated `totalAssets` value prevents her from accessing her earned rewards. This exploitation can continue indefinitely, creating a permanent DoS for yield distribution.

Recommendations

Short term, implement a time-based protection mechanism that prevents the `notify` function from being called again until the current vesting period has completed. This would ensure that once a notification occurs, the vesting schedule cannot be manipulated until the full `lockDuration` has passed.

Long term, implement invariant tests that verify withdrawal guarantees over time, ensuring that rewards become fully accessible within the defined vesting period regardless of external interactions.

5. Inconsistent balance tracking in vault creates DoS for asset borrowing	
Severity: **High**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-CAP-5
Target: `contracts/vault/Vault.sol`	

Description

The borrowing functionality for an asset in the vault is vulnerable to DoS attacks due to inconsistent accounting practices. The system can enter a state where the fundamental invariant "total supply of an asset should never be less than total borrows" is violated, resulting in negative available balances that break core protocol functionality.

The issue stems from inconsistent accounting approaches throughout the codebase, with some functions relying on `totalSupplies` and `totalBorrows` accounting variables of an asset while others use direct `balanceOf` token queries.

```
function burn(IVault.VaultStorage storage $, IVault.MintBurnParams memory params)
    external
    updateIndex($, params.asset)
{
    if (params.deadline < block.timestamp) revert PastDeadline();
    if (params.amountOut < params.minAmountOut) {
        revert Slippage(params.asset, params.amountOut, params.minAmountOut);
    }

    $.totalSupplies[params.asset] -= params.amountOut;
    IERC20(params.asset).safeTransfer(params.receiver, params.amountOut);
    ...
}
```

Figure 5.1: Code snippet of the `burn` function in the `VaultLogic` contract that uses `totalSupplies` (`contracts/vault/libraries/VaultLogic.sol#L111-L124`)

One such inconsistency is in the `burn` function shown in figure 5.1, which decreases the accounting variable `totalSupplies` without validating against the available balance, while relying on the actual token balance for transfers. Therefore, if the vault has physical tokens that are not tracked in `totalSupplies` but are sufficient for the withdrawal amount, the transaction will succeed, thus reducing `totalSupplies` further and potentially below `totalBorrows`.

```
function availableBalance(address _asset) external view returns (uint256 amount) {
    VaultStorage storage $ = getVaultStorage();
    amount = $.totalSupplies[_asset] - $.totalBorrows[_asset];
}
```

Figure 5.2: Code snippet from `availableBalance` function `Vault` contract (`contracts/vault/Vault.sol#L206-L209`)

This can lead to negative available balance, as the available balance is computed as the difference between `totalSupplies` and `totalBorrows` of an asset (figure 5.2).

```
uint256 assetDecimals = $.reservesData[_asset].decimals;
maxBorrowableAmount = remainingCapacity * (10 ** assetDecimals) / assetPrice;

// Get total available assets using the vault's availableBalance function
uint256 totalAvailable =
IVault($.reservesData[_asset].vault).availableBalance(_asset);

// Limit maxBorrowableAmount by total available assets
if (totalAvailable < maxBorrowableAmount) {
    maxBorrowableAmount = totalAvailable;
}
```

Figure 5.3: Code snippet of `maxBorrowable` function (`contracts/lendingPool/libraries/ViewLogic.sol#L83-L91`)

Once the invariant is broken for an asset, all borrowing operations for that asset will fail system-wide. This occurs because the `maxBorrowable` function relies on `availableBalance`, which returns a negative value when `totalSupplies` < `totalBorrows`.

Exploit Scenario

A vault for asset A has a total supply of `100` tokens. Bob, an authorized agent, borrows `90` tokens, increasing the total borrows to `90` while total supply remains at `100`.

Mallory, who wants to brick the functionality of the vault, wants to withdraw `15` tokens, but the vault's available balance is only `10` tokens. To circumvent this, she directly transfers `5` additional tokens to the vault (bypassing the mint function, so `totalSupplies` is not updated). She then initiates a burn operation to withdraw `15` tokens.

Since the contract physically has `15` tokens, the token transfer succeeds. The burn function decreases `totalSupplies` by `15`, from `100` to `85`, while `totalBorrows` remains at `90`.

As a result, the system enters an inconsistent state where `totalSupplies` (`85`) < `totalBorrows` (`90`), violating the theoretical definition that total supplies should always be greater than or equal to total borrows. When `availableBalance` is called, it returns `-5`, effectively breaking borrowing functionality for that specific asset across the protocol even though the contract may physically hold sufficient assets.

Recommendations

Short term, modify the burn function to validate withdrawals against the accounting-based available balance.

Long term, implement a consistent token tracking strategy across the system and create comprehensive invariant tests that verify the relationship between `totalSupplies`, `totalBorrows`, and the actual token balances under various scenarios.

6. Unsafe asset removal without borrow validation

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CAP-6
Target: `contracts/vault/libraries/VaultLogic.sol`	

Description

The `removeAsset` function in the Vault contract allows removing an asset from the vault's asset list without checking for outstanding borrows. This can lead to potential system inconsistencies and unexpected behavior in the lending protocol.

```
function removeAsset(IVault.VaultStorage storage $, address _asset) external {
    address[] memory cachedAssets = $.assets;
    uint256 length = cachedAssets.length;
    bool removed;
    for (uint256 i; i < length; ++i) {
        if (_asset == cachedAssets[i]) {
            $.assets[i] = cachedAssets[length - 1];
            $.assets.pop();
            removed = true;
            break;
        }
    }
    if (!removed) revert AssetNotSupported(_asset);
    emit RemoveAsset(_asset);
}
```

Figure 6.1: Code snippet of the function `removeAsset` in the `VaultLogic` contract (`contracts/vault/libraries/VaultLogic.sol#L192-L208`)

As shown in figure 6.1, the current implementation of removing an asset does not validate whether the asset has any outstanding borrows before removal. This can create an issue where an asset with active borrows is removed from the vault. As result, subsequent operations, such as repayments for the removed asset, could become impossible since repayment functions rely on the existence of an asset in the vault system.

Exploit Scenario

An admin mistakenly removes an asset from the vault while there are active loans. Borrowers attempt to repay their loans for this asset. Repayment fails because the asset is no longer in the vault's asset list. As a result, borrowers' funds become effectively locked, and they are unable to clear their debt.

Recommendations

Short term, add a validation check to ensure that total borrows for the asset are zero before removal.

Long term, develop a comprehensive suite of unit tests that rigorously validate the asset removal process, ensuring that assets with outstanding borrows cannot be removed and that the system maintains its integrity during asset management operations.

7. Initial auction start price can be lower than minimum start price	
Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-CAP-7
Target: `contracts/feeAuction/FeeAuction.sol`	

Description

The `setStartPrice` function lacks a validation mechanism to ensure that the start price meets the minimum established threshold. This oversight could inadvertently allow administrators to set an auction start price below the predefined minimum, potentially enabling users to acquire assets at prices lower than the protocol's intended economic safeguards.

```
function setStartPrice(uint256 _startPrice) external
checkAccess(this.setStartPrice.selector) {
    FeeAuctionStorage storage $ = get();
    $.startPrice = _startPrice;
    emit SetStartPrice(_startPrice);
}
```

Figure 7.1: Code snippet of setStartPrice function in the feeAuction contract (`contracts/feeAuction/FeeAuction.sol#L85-L89`)

As shown in figure 7.1, the current implementation allows setting the auction start price without validating against the minimum price threshold. This could potentially enable it to initiate an auction with extremely low or zero prices, effectively allowing unauthorized asset acquisition at negligible or no cost.

Exploit Scenario

An admin mistakenly sets the start price to zero or an extremely low value below the minimum allowable price. As a result, assets can be acquired at a fraction of their true market value.

Recommendations

Short term, implement a validation check in the `setStartPrice` function to ensure that the new start price is not less than the predefined `minStartPrice`.

Long term, develop a rigorous testing strategy that includes comprehensive unit tests and fuzzing techniques to systematically validate the system's critical invariants, ensuring robust behavior under various input conditions and potential edge cases.

8. Immediate liquidation possible when reducing an agent's liquidation threshold

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CAP-8
Target: `contracts/delegation/Delegation.sol`	

Description

The `modifyAgent` function allows reducing an agent's liquidation threshold without checking if this would immediately put the agent in a liquidatable position. If an agent's current position is near the threshold, an admin could unknowingly (or maliciously) trigger immediate liquidations by lowering the threshold below the agent's current position.

```
function modifyAgent(address _agent, uint256 _ltv, uint256 _liquidationThreshold)
    external
    checkAccess(this.modifyAgent.selector)
{
    // if liquidation threshold or ltv is greater than 100%, agent
    // could borrow more than they are collateralized for
    if (_liquidationThreshold > 1e27) revert InvalidLiquidationThreshold();
    if (_ltv > 1e27) revert InvalidLtv();

    DelegationStorage storage $ = getDelegationStorage();

    // Check that the agent exists
    if (!$.agentData[_agent].exists) revert AgentDoesNotExist();

    $.agentData[_agent].ltv = _ltv;
    $.agentData[_agent].liquidationThreshold = _liquidationThreshold;
    emit ModifyAgent(_agent, _ltv, _liquidationThreshold);
}
```

Figure 8.1: `contracts/delegation/Delegation.sol#L227-L244`

Recommendations

Short term, add a safety check to prevent setting a liquidation threshold that would immediately put an agent in a liquidatable position.

Long term, implement a time-delay mechanism for parameter changes that could adversely affect users, allowing agents sufficient time to react to upcoming changes before they take effect.

9. FeeAuction's buy function allows purchasing at sub-optimal prices by adding tokens during an active auction

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-CAP-9
Target: `contracts/feeAuction/FeeAuction.sol`	

Description

The `FeeAuction` contract allows users to purchase all tokens from the auction at a linearly decreasing price over time, but does not account for tokens being added during an active auction. An attacker can call the permissionless `realizeInterest` function on the `FractionalReserve` contract to add more tokens to the auction immediately before purchasing, effectively getting these additional tokens for free.

This finding's low severity assumes there are a variety of competent MEV searchers on the network. If not enough MEV searchers are present, the severity of this finding would become high because the reward from exploiting this finding increases drastically over time.

Exploit Scenario

An auction is selling 100 USDC for a current price of 105 cUSD, which no participants consider profitable enough to execute. Alice notices there is 200 USDC of unclaimed interest in the `FractionalReserve`. Alice executes the following transaction sequence:

1. She calls the `FractionalReserve.realizeInterest` function to send the 200 USDC to the `FeeAuction` contract.
2. She immediately calls the `FeeAuction.buy` function to purchase all 300 USDC for only 105 cUSD.
3. Alice effectively receives a 65% discount on the total value, making the auction profitable for her while depriving the protocol of fair compensation for the additional 200 USDC tokens.

Recommendations

Short term, modify the `FeeAuction` contract to determine token amounts at the beginning of the auction and transfer only those specific amounts, ignoring any tokens added during the active auction.

Long term, implement a more robust auction mechanism that determines price based on both time and the quantity of tokens being sold, and add comprehensive unit tests covering edge cases including mid-auction token additions.

10. Missing event emissions for critical parameter changes in VaultAdapter

Severity: **Informational**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-CAP-10
Target: `contracts/oracle/libraries/VaultAdapter.sol`	

Description

The VaultAdapter contract lacks event emissions for critical parameter changes in the `setSlopes` and `setLimits` functions. These functions modify key protocol parameters that directly affect interest rate calculations, but do not emit events to record these important changes, making it difficult to track changes or detect unauthorized modifications.

```
/// @notice Set utilization slopes for an asset
/// @param _asset Asset address
/// @param _slopes Slope data
function setSlopes(address _asset, SlopeData memory _slopes) external
checkAccess(this.setSlopes.selector) {
    getVaultAdapterStorage().slopeData[_asset] = _slopes;
}

/// @notice Set limits for the utilization multiplier
/// @param _maxMultiplier Maximum slope multiplier
/// @param _minMultiplier Minimum slope multiplier
/// @param _rate Rate at which the multiplier shifts
function setLimits(uint256 _maxMultiplier, uint256 _minMultiplier, uint256 _rate)
    external
    checkAccess(this.setLimits.selector)
{
    VaultAdapterStorage storage $ = getVaultAdapterStorage();
    $.maxMultiplier = _maxMultiplier;
    $.minMultiplier = _minMultiplier;
    $.rate = _rate;
}
```

Figure 10.1: contracts/oracle/libraries/VaultAdapter.sol#L51-L70

Recommendations

Short term, add events that emit relevant details when the `setSlopes` and `setLimits` functions are called.

Long term, implement a comprehensive event emission policy for all admin-controlled parameter changes across the protocol, and create an off-chain monitoring system that tracks these events to alert stakeholders of parameter changes.

11. VaultAdapter's setSlopes function permits zero or maximum kink values that cause division by zero

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CAP-11
Target: `contracts/oracle/libraries/VaultAdapter.sol`	

Description

The `setSlopes` function in `VaultAdapter.sol` allows admin to set a kink value without any validation. If kink is set to 0 or 1e27, the interest rate calculation will revert due to division by zero, as seen in figures 11.1 through 11.4:

```
/// @notice Set utilization slopes for an asset
/// @param _asset Asset address
/// @param _slopes Slope data
function setSlopes(address _asset, SlopeData memory _slopes) external
checkAccess(this.setSlopes.selector) {
    getVaultAdapterStorage().slopeData[_asset] = _slopes;
}
```

Figure 11.1: contracts/oracle/libraries/VaultAdapter.sol#L51-L56

```
/// @dev Slope data for an asset
struct SlopeData {
    uint256 kink;
    uint256 slope0;
    uint256 slope1;
}
```

Figure 11.2: contracts/interfaces/IVaultAdapter.sol#L16-L21

If the `kink` is ever set to `0`, it would revert in the `interestRate` calculation because of division by 0:

```
interestRate =
    (slopes.slope0 * _utilization / slopes.kink) *
$.utilizationData[_asset].utilizationMultiplier / 1e27;
```

Figure 11.3: contracts/oracle/libraries/VaultAdapter.sol#L104-L105

If the `kink` is ever set to `1e27`, it would revert in the `utilizationMultiplier` calculation because of division by 0:

```
$.utilizationData[_asset].utilizationMultiplier *=
    (1e27 + (1e27 * excess / (1e27 - slopes.kink)) * (_elapsed * $.rate / 1e27));
```

Figure 11.4: contracts/oracle/libraries/VaultAdapter.sol#L87-L88

This would break the core interest rate functionality of the protocol.

Recommendations

Short term, add input validation in the `setSlopes` function to ensure `kink` is never set to 0 or 1e27.

Long term, implement a comprehensive validation framework for all parameter-setting functions in the protocol, and add unit tests that verify system behavior with boundary values for all configurable parameters.

12. Unvalidated _vault address in VaultAdapter allows interest rate manipulation

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-CAP-12
Target: `contracts/oracle/libraries/VaultAdapter.sol`	

Description

The `VaultAdapter.rate` function accepts an unvalidated `_vault` address and makes external calls to it without verifying that it is a legitimate vault. An attacker can supply a malicious contract as the vault address, which would allow manipulation of critical storage values (`$.utilizationData[_asset].index`, `$.utilizationData[_asset].lastUpdate`, and `$.utilizationData[_asset].utilizationMultiplier`) that determine interest rates throughout the protocol, as shown in figure 12.1.

```
function rate(address _vault, address _asset) external returns (uint256
latestAnswer) {
    VaultAdapterStorage storage $ = getVaultAdapterStorage();

    uint256 elapsed;
    uint256 utilization;
    if (block.timestamp > $.utilizationData[_asset].lastUpdate) {
        uint256 index = IVault(_vault).currentUtilizationIndex(_asset);
        elapsed = block.timestamp - $.utilizationData[_asset].lastUpdate;

        /// Use average utilization except on the first rate update
        if (elapsed != block.timestamp) {
            utilization = (index - $.utilizationData[_asset].index) / elapsed;
        } else {
            utilization = IVault(_vault).utilization(_asset);
        }

        $.utilizationData[_asset].index = index;
        $.utilizationData[_asset].lastUpdate = block.timestamp;
    } else {
        utilization = IVault(_vault).utilization(_asset);
    }

    latestAnswer = _applySlopes(_asset, utilization, elapsed);
}
```

Figure 12.1: contracts/oracle/libraries/VaultAdapter.sol#L26-L49

Exploit Scenario

An attacker deploys a malicious contract implementing the `IVault` interface. They call the `VaultAdapter.rate` function with their malicious contract as the `_vault` parameter, which returns crafted values from the `currentUtilizationIndex` and `utilization` methods. This manipulates storage values in `VaultAdapter`, artificially inflating interest rates returned by `_applySlopes`. The inflated rates propagate through `RateOracle.utilizationRate` to `InterestDebtToken.nextInterestRate`, causing borrowers to pay excessive interest.

Recommendations

Short term, implement access control on the `rate` function or add an allowlist mechanism to validate vault addresses before making external calls to them.

Long term, add comprehensive input validation throughout the codebase to prevent similar issues in other functions.

13. Fee auction allows buying zero assets, leading to front-running attacks

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-CAP-13
Target: `contracts/feeAuction/FeeAuction.sol`	

Description

The `FeeAuction` contract's buy function allows users to purchase a basket of assets at a given price. However, there is no way for users to specify which auction their buy transaction should be applied to, leading to a race condition where the loser of the race will pay and receive no tokens.

```
function _transferOutAssets(address[] calldata _assets, address _receiver)
    internal
    returns (uint256[] memory balances)
{
    uint256 assetsLength = _assets.length;
    balances = new uint256[](assetsLength);
    for (uint256 i; i < assetsLength; ++i) {
        address asset = _assets[i];
        balances[i] = IERC20(asset).balanceOf(address(this));
        if (balances[i] > 0) IERC20(asset).safeTransfer(_receiver, balances[i]);
    }
}
```

Figure 13.1: Code snippet from the `_transferOutAssets` function (`contracts/feeAuction/FeeAuction.sol#L113-L124`)

The `_transferOutAssets` function in the `FeeAuction` contract transfers assets to a receiver only if the asset balance is positive, but it does not revert if there is no balance for the asset or if the buy call was intended for a previous auction.

This enables a front-running attack where a malicious actor can observe pending buy transactions in the mempool, execute them first to deplete asset balances, and force subsequent transactions to pay without receiving any assets. Since each successful transaction doubles the next auction's starting price, victims end up paying double the price without receiving anything in return.

Exploit Scenario

A malicious actor monitors the mempool for legitimate `FeeAuction` buy transactions. When one is detected, they front-run it with their own transaction to buy the same assets. When the victim's transaction executes, the asset balances are already zero, but the

payment is still taken. The victim receives nothing while paying double the price the attacker paid due to the price doubling after each successful purchase.

Recommendations

Short term, add a validation check in the `_transferOutAssets` function to ensure that the sum of asset balances is greater than zero before accepting payment, reverting the transaction if zero assets would be transferred.

Long term, implement a more robust auction system that reserves assets for specific bids during their processing, preventing front-running attacks. Additionally, consider adding comprehensive transaction simulation tests that verify that assets are actually transferred during an auction, and that edge cases like zero-balance transfers are properly handled.

14. Discrepancy between health calculation and slashable collateral computation

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-CAP-14
Target: `contracts/delegation/Delegation.sol`	

Description

The protocol shows a critical discrepancy between how an agent's health is calculated versus how slashable collateral is determined. This may lead to the protocol absorbing losses during liquidation events, undermining the security model and financial stability of the system.

```
function slashTimestamp(address _agent) public view returns (uint48 _slashTimestamp)
{
    DelegationStorage storage $ = getDelegationStorage();
    _slashTimestamp = uint48(Math.max((epoch() - 1) * $.epochDuration,
$.agentData[_agent].lastBorrow));
}
```

Figure 14.1: Code snippet from `slashTimeStamp` function (`contracts/delegation/Delegation.sol#L73-L76`)

As shown in figure 14.1 and 14.2, the liquidation mechanism contains a critical timing vulnerability where health factor evaluation and slashing operate on different time bases.

```
function coverage(address _agent) public view returns (uint256 delegation) {
    NetworkMiddlewareStorage storage $ = getNetworkMiddlewareStorage();
    ...
    uint48 _timestamp = uint48(block.timestamp);

    for (uint256 i = 0; i < _vaults.length; i++) {
        (uint256 value,) = coverageByVault(_network, _agent, _vaults[i], _oracle,
_timestamp);
        delegation += value;
    }
}
```

Figure 14.2: Code snippet from `coverage` function (`contracts/delegation/providers/symbiotic/NetworkMiddleware.sol#L195-L206`)

While the agent coverage calculation (figure 14.2) computes liquidation health factors using *all* current collateral, the slashable collateral is limited to deposits made *before* the most recent of either the last borrow time or the previous epoch.

This creates an exploitable gap where agents can add new collateral to improve their health metrics, but if liquidation occurs, these recent deposits remain protected from slashing.

Consequently, when liquidation is needed, the protocol is unable to recover sufficient collateral from these newer deposits, forcing it to absorb losses that should have been covered by slashable collateral. This fundamental misalignment undermines the protocol's security model and threatens its financial stability.

Exploit Scenario

An agent notices their position is close to liquidation. They coordinate with friendly delegators to add fresh collateral in the current epoch, which improves their health factor calculation. If liquidation still occurs, only the delegations from previous epochs will be slashed, while the "rescue delegators" remain protected. After the liquidation risk passes, these friendly delegators can withdraw their collateral without consequences. In extreme cases, if all existing delegations are from the current epoch, the protocol may be unable to slash any collateral, forcing the protocol to absorb the loss.

Recommendations

Short term, modify the slashable collateral calculation to ensure that all delegations that contribute to an agent's health factor can also be slashed proportionally, regardless of when they were contributed.

Long term, implement a comprehensive bonding mechanism where delegated collateral must remain locked for a minimum period before it can be withdrawn. This will ensure that all collateral that contributes to an agent's borrowing capacity is also available for slashing during the entire period it is being used as collateral.

15. Reward distribution enables front-running attacks and reward siphoning

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-CAP-15
Target: `contracts/delegation/NetworkMiddleware.sol`, `contracts/delegation/Delegation.sol`	

Description
The reward distribution mechanism, called during loan repayment, allocates rewards to all vaults backing an agent at the time of repayment with no consideration for the amount of time the restaker has been delegating to the agent.

This creates a significant timing vulnerability where actors can monitor pending repayment transactions in the mempool and strategically add deposits before repayment to capture a proportional share of rewards. This allows opportunistic actors to systematically siphon rewards from long-term backers who shouldered the actual lending risks.

Exploit Scenario
An agent takes out a large loan backed by Vault A, which has provided collateral for several months.

When the time comes to repay the loan with substantial accrued interest, Vault B front-runs the repayment transaction and strategically adds delegation just before repayment and rewards distribution.

When the agent repays, the interest rewards are distributed proportionally based on the current delegation, allowing Vault B to receive a significant portion of the rewards despite only backing the agent for a minimal time period. This allows Vault B to capture rewards without taking an equivalent risk.

Recommendations
Short term, modify the reward distribution mechanism to exclude any collateral added in the current epoch from reward eligibility. Consider adding a mechanism that allows rewards to vest based on the amount of delegation and time delegated instead of delegation amount at a specific time.

Long term, implement a comprehensive reward accounting system that tracks and logs the historical collateral contributions of each vault throughout the lifecycle of loans, ensuring

that reward distribution accurately reflects the risk and support provided by each vault over time.

16. Unaccounted external vault investment losses can create withdrawal shortfalls

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CAP-16
Target: `contracts/vault/libraries/FractionalReserveLogic.sol`, `contracts/vault/FractionalReserve.sol`	

Description
The protocol invests idle assets in external vaults like Yearn for yield generation, but fails to properly account for losses when divesting. When changing the address of an external vault, the protocol divests from the existing vault regardless of any losses incurred. These losses are not tracked or accounted for in the system. This structure creates an unfair "first out" advantage, allowing early withdrawers to receive their full amounts while later withdrawers may find their requests impossible to fulfill despite the system showing sufficient supply, as the physical assets no longer exist to back those withdrawals.

Exploit Scenario
The protocol has 100 tokens of Asset A with a total supply recorded as 100 tokens.

1. 90 tokens are invested in a Yearn vault.
2. The protocol admin changes the vault address, triggering divestment from Yearn.
3. Due to strategy losses, only 80 tokens are returned.
4. The protocol still records a total supply of 100 tokens, but physically has only 90 tokens available.

Early users can withdraw their tokens, but later users will be unable to withdraw their full amounts. The protocol has no mechanism to account for this 10-token loss or compensate users, creating a "first out" advantage where early withdrawers receive their full amounts while later withdrawers face shortfalls.

Recommendations
Short term, modify the divestment function to track and account for any losses when divesting from external vaults, accumulating them as protocol debt that gets paid down using future yield profits before sending excess returns to the fee auction.

Long term, implement a comprehensive risk management system that carefully evaluates external vault strategies, includes safeguards against significant losses, compensates for loss using future profits, and creates a loss-socialization mechanism that fairly distributes any unavoidable losses among all users rather than disadvantageing only the last withdrawers.

17. Wrong capTokenDecimals value used in StakedCapAdapter.price causes inaccurate prices

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-CAP-17
Target: `contracts/oracle/libraries/StakedCapAdapter.sol`	

Description

The `StakedCapAdapter` library contains critical decimal handling errors that will cause wildly inaccurate price calculations. The price calculation logic incorrectly uses the decimal count itself (e.g., 18) rather than the corresponding scaling factor (10^{18}) in two places, causing prices to be calculated incorrectly by several orders of magnitude.

```
function price(address _asset) external view returns (uint256 latestAnswer, uint256
lastUpdated) {
    address capToken = IERC4626(_asset).asset();
    (latestAnswer, lastUpdated) = IOracle(msg.sender).getPrice(capToken);
    uint256 capTokenDecimals = IERC20Metadata(capToken).decimals();
    uint256 pricePerFullShare = IERC4626(_asset).convertToAssets(capTokenDecimals);
    latestAnswer = latestAnswer * pricePerFullShare / capTokenDecimals;
}
```

Figure 17.1: contracts/oracle/libraries/StakedCapAdapter.sol#L15-L21

First, it incorrectly passes the decimal count (e.g., 18) to the `convertToAssets` function instead of passing a proper share amount (e.g., 10^{18}). The `convertToAssets` function expects a share amount, not a decimal count. Second, it divides by the decimal count itself (e.g., 18) rather than the decimal scaling factor (10^{18}), which would dramatically inflate the resulting price.

Exploit Scenario

A malicious actor observes that the `StakedCapAdapter` is reporting inaccurate prices for staked CAP tokens. They exploit this by executing trades that take advantage of the price disparity.

Recommendations

Short term, correct the decimal handling by using the proper scaling factors instead of the decimal counts.

Long term, implement comprehensive unit tests specifically for price oracles with edge cases around decimal handling, and add validation that compares calculated prices against expected ranges to catch potential errors.

18. Liquidation mechanism can be permanently disabled by misconfigured grace and expiry periods

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CAP-18
Target: `contracts/lendingPool/Lender.sol`	

Description

The Lender contract does not check during initialization whether `grace >= expiry`. If `grace` is greater than or equal to `expiry`, liquidations will never happen.

```
if (health >= 1e27) revert HealthFactorNotBelowThreshold();
if (emergencyHealth >= 1e27) {
    if (block.timestamp <= start + grace) revert GracePeriodNotOver();
    if (block.timestamp >= start + expiry) revert LiquidationExpired();
}
```

Figure 18.1: Code snippet from `validateLiquidation` function (`contracts/lendingPool/libraries/ValidationLogic.sol#L98-L102`)

As shown in figure 18.1, the `validateLiquidation` function requires `block.timestamp` to be both greater than `start+grace` and less than `start+expiry`, which is impossible when `grace >= expiry`. Since there is no way to reset the grace or expiry parameters after the contract is deployed, this misconfiguration would permanently disable the liquidation mechanism.

Exploit Scenario

An admin deploys the Lender contract with a grace period of two days and an expiry period of one day. When an agent's position becomes unhealthy and a liquidator initiates liquidation, the system will never allow the liquidation to be executed since the grace period exceeds the expiry period. This prevents timely liquidations and potentially results in protocol insolvency due to accumulating bad debt.

Recommendations

Short term, add a validation check in the initialize function and any functions that update these parameters to ensure that `grace < expiry`.

Long term, consider implementing an emergency mechanism that allows authorized parties to reset critical parameters in case of misconfiguration.

19. Oracle update front-running allows extraction of value from vaults	
Severity: **Medium**	Difficulty: **Medium**
Type: Timing	Finding ID: TOB-CAP-19
Target: `contracts/vault/Vault.sol`	

Description

CAP protocol's price-based minting and burning functions are vulnerable to oracle sandwiching. When Chainlink oracle updates occur, an attacker can exploit the timing difference between updates to extract value from the protocol in proportion to the size of the Chainlink oracle update. This attack is possible because the protocol uses the current oracle price without any protection against sandwiching of oracle updates.

The impact of this attack is limited to the sum of the allowable price deviation for each Chainlink price oracle (cUSD and the respective stablecoin).

Exploit Scenario

The USDC and cUSD oracle are initially both reporting a price of 1 USD for both 1 USDC and 1 cUSD. The attacker monitors the Chainlink USDC oracle feed for pending updates. A volatility event occurs, and the price of USDC decreases to 1.0025 USDC = 1 USD.

This variance is larger than the Chainlink trigger threshold, so Chainlink submits a price update transaction to their oracle. The attacker detects the imminent price update transaction, and they deposit 1,000,000 USDC to mint 1,000,000 cUSD at the current price.

The Chainlink USDC oracle now updates to reflect 1.0025 USDC = 1 USD. The exchange rate for USDC to cUSD is now 1.0025 USDC = 1 cUSD.

After the oracle update completes, they immediately burn their cUSD and receive 1,002,500 USDC based on the new price ratio, netting a 2,500 USDC profit. This attack exploits natural market movements and legitimate oracle updates rather than any manipulation of the oracle itself.

Recommendations

Short term, implement baseline fees for mint and burn operations that exceed the potential profit margin from oracle sandwiching. For the example above using two 0.25% deviation oracles, the minimum effective fee would be 0.5%. This is calculated using the most dangerous price conditions that may exist before a Chainlink oracle price update: one

where the USDC value has decreased 0.25% and has an update pending, plus one where the cUSD value has increased by 0.24% and does not have a price update pending.

Long term, consider implementing TWAP (Time-Weighted Average Price) mechanisms or add circuit breakers that temporarily pause mint/burn operations when large price movements are detected, allowing time for all relevant oracles to synchronize their prices.

20. Asset removal does not reset isBorrowing flag for agents

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-CAP-20
Target: `contracts/lendingPool/libraries/ValidationLogic.sol`	

Description
When an asset is removed from the system, the validation checks only if the principal debt is fully repaid, but it does not verify if the agent's `isBorrowing` flag for that asset is cleared. If the asset ID is later reused for a different asset, agents who previously borrowed the old asset will incorrectly have their borrowing flags set to true for the new asset, despite never having borrowed it.

Exploit Scenario
Agent A borrows Asset X, which uses reserve ID 5. The agent repays the principal but still has some interest debt. Asset X is removed from the system since `validateRemoveAsset` checks only principal debt. Later, a new Asset Y is added and gets assigned the same reserve ID 5. Agent A now has their `isBorrowing` flag incorrectly set to true for Asset Y, despite never borrowing it. This creates accounting inconsistencies and may affect the agent's health factor calculations, borrowing limits, or liquidation risks.

Recommendations
Short term, modify the asset removal process to iterate through all agents and clear their borrowing flags for the asset being removed.

21. Invalid network registration in Delegation.registerNetwork can cause DoS

Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-CAP-21
Target: `contracts/delegation/Delegation.sol`	

Description

The `Delegation` contract allows adding networks to agents through the `registerNetwork` function, but lacks validation for network addresses and provides no mechanism to remove invalid networks once added. If an invalid address (like `address(0)`) or a non-compliant contract is registered as a network for an agent, key functions like `coverage` and `slashableCollateral` will permanently revert for that agent due to failed external calls, effectively causing a denial of service. This issue is exacerbated by the absence of functions to either remove networks or remove the affected agent entirely.

```
function registerNetwork(address _agent, address _network) external
checkAccess(this.registerNetwork.selector) {
    DelegationStorage storage $ = getDelegationStorage();

    // Check for duplicates
    if ($.networkExistsForAgent[_agent][_network]) revert DuplicateNetwork();

    $.networks[_agent].push(_network);
    $.networkExistsForAgent[_agent][_network] = true;
    emit RegisterNetwork(_agent, _network);
}
```

Figure 21.1: contracts/delegation/Delegation.sol#L249-L258

Exploit Scenario

An admin registers a network for an agent but provides an incorrect address due to human error. The transaction succeeds, but now functions like `coverage` and `slash` will permanently revert for this agent when they attempt to call methods on the invalid network contract. With no way to remove the faulty network or the agent, the agent becomes permanently locked out of protocol functionality.

Recommendations

Short term, add basic address validation in the `registerNetwork` function to prevent registration of `address(0)` and implement an admin-controlled `unregisterNetwork` function to remove specific networks from an agent.

Long term, implement a more comprehensive agent management system including the ability to remove agents completely and re-add them if necessary.

22. Lack of verification on ERC4626 vault withdrawal amounts

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-CAP-22
Target: `contracts/vault/libraries/FractionalReserveLogic.sol`	

Description

The `divest` function does not verify the actual amount of assets returned from ERC4626 vault withdrawals against the expected `divestAmount`, which could lead to unexpected shortfalls.

```
if (divestAmount > 0) {
    $.loaned[_asset] -= divestAmount;
    IERC4626($.vault[_asset]).withdraw(divestAmount, address(this), address(this));
}
```

Figure 22.1: Code snippet from the `divest` function (contracts/vault/libraries/FractionalReserveLogic.sol#L63-L66)

As shown in figure 22.1, the loaned asset amount is reduced by the `divestAmount` before the actual withdrawal occurs, and the function does not capture the return value from the withdraw call. If the vault returns fewer assets than requested, the accounting system would incorrectly assume that the full amount was withdrawn, creating a discrepancy between the recorded loaned assets and the actual assets in the protocol's possession. This could lead to systematic asset shortfalls that accumulate over time.

Currently, CAP supports Yearn vaults that default to zero loss tolerance, causing transactions to revert if any loss occurs. However, if CAP integrates with other ERC4626 vaults that permit losses, the protocol would not account for this discrepancy.

Recommendations

Short term, have the code capture and verify the actual returned amount from ERC4626 vault withdrawals, comparing it against the expected `divestAmount` and reverting if the returned amount is insufficient.

Long term, implement a comprehensive risk management framework for external vault integrations, including thorough vetting of vault implementations, testing of edge cases like loss scenarios, and mechanisms to handle potential discrepancies between expected and actual withdrawal amounts.

23. Agent LTV can be configured equal to or higher than liquidation threshold

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-CAP-23
Target: `contracts/delegation/Delegation.sol`	

Description

The `addAgent` and `modifyAgent` functions allow setting an agent's loan-to-value (LTV) ratio equal to or higher than its liquidation threshold. If LTV is greater than or equal to the liquidation threshold, an agent could be liquidated immediately after borrowing, or worse, never reach liquidation even when underwater.

```
function addAgent(address _agent, uint256 _ltv, uint256 _liquidationThreshold)
    external
    checkAccess(this.addAgent.selector)
{
    // if liquidation threshold or ltv is greater than 100%, agent
    // could borrow more than they are collateralized for
    if (_liquidationThreshold > 1e27) revert InvalidLiquidationThreshold();
    if (_ltv > 1e27) revert InvalidLtv();
```

Figure 23.1: `contracts/delegation/Delegation.sol#L202-L209`

Exploit Scenario

An admin mistakenly sets an agent's LTV ratio equal to or higher than its liquidation threshold. When this agent borrows funds up to its allowed LTV, it would either become immediately liquidatable or, in the worse case, never reach the liquidation threshold even when undercollateralized, putting the protocol funds at risk.

Recommendations

Short term, add validation in both `addAgent` and `modifyAgent` functions to ensure that `_ltv < _liquidationThreshold`.

Long term, implement standard parameter validation patterns across the codebase and add comprehensive unit tests that verify proper validation of configuration parameters, especially those critical to the economic security of the protocol.

24. ZapOFTComposer._lzCompose may fail with USDT

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CAP-24
Target: `contracts/zap/ZapOFTComposer.sol`	

Description

The `ZapOFTComposer` contract uses the standard `approve` method despite importing `SafeERC20`, which can cause transactions to fail when processing certain tokens like USDT that do not allow approving a new non-zero amount when an existing non-zero approval is already in place. This issue can occur when, for example, a user creates a `zap` with multiple USDT inputs.

```
function _lzCompose(address, /*_oApp*/ bytes32, /*_guid*/ bytes calldata _message,
address, bytes calldata)
    internal
    override
{
    // Decode the payload to get the message
    bytes memory payload = OFTComposeMsgCodec.composeMsg(_message);
    IZapOFTComposer.ZapMessage memory zapMessage = abi.decode(payload,
(IZapOFTComposer.ZapMessage));

    // approve all inputs to the zapTokenManager
    IZapRouter.Input[] memory inputs = zapMessage.order.inputs;
    uint256 inputLength = inputs.length;
    for (uint256 i = 0; i < inputLength; i++) {
        IZapRouter.Input memory input = inputs[i];
        if (input.amount > 0) {
            IERC20(input.token).approve(zapTokenManager, input.amount);
        }
    }
```

Figure 24.1: `contracts/zap/ZapOFTComposer.sol#L39-L55`

Recommendations

Short term, have the code use the `forceApprove` function from the already-imported `SafeERC20` library to properly handle non-standard ERC20 tokens.

Long term, implement more comprehensive token compatibility checks across the protocol.

25. Fee auction allows assets to be purchased for free

Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-CAP-25
Target: `contracts/feeAuction/FeeAuction.sol`	

Description

The `FeeAuction` contract implements a Dutch auction mechanism where the price decreases linearly over time, eventually reaching zero if no one purchases the assets during the auction period. This allows users to claim valuable protocol fee tokens for free when an auction expires without bids. While comments in the code suggest this behavior is intentional, `/// @notice Current price in the payment token, linearly decays toward 0 over time`, we would recommend against it as it could lead to value loss for the protocol.

```
function currentPrice() public view returns (uint256 price) {
    FeeAuctionStorage storage $ = get();
    uint256 elapsed = block.timestamp - $.startTimestamp;
    if (elapsed > $.duration) elapsed = $.duration;
    price = $.startPrice * (1e27 - (elapsed * 1e27 / $.duration)) / 1e27;
}
```

Figure 25.1: `contracts/feeAuction/FeeAuction.sol#L52-L57`

Recommendations

Short term, modify the `currentPrice` function to enforce a minimum price floor greater than zero to ensure that protocol fees are never sold for free.

Long term, implement a dynamic price floor mechanism that adjusts based on the actual value of the tokens being auctioned.

26. Small borrows can create economically unviable liquidatable positions leading to bad debt accumulation

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-CAP-26
Target: `contracts/lendingPool/Lender.sol`	

Description

The `borrow` function in the `Lender` contract lacks a minimum borrow amount check, allowing agents to create positions as small as 1 wei. When these small positions become liquidatable, the gas costs of liquidation far exceed the potential liquidation value plus liquidation bonus, making them economically unviable to liquidate. This leads to accumulation of bad debt in the system, as these positions will likely never be liquidated.

The impact is limited by the fact that each agent can have only one borrowing position per asset type. This means the maximum number of economically unviable positions would be constrained by the number of borrowable assets in the system, limiting the total potential bad debt.

```
function borrow(address _asset, uint256 _amount, address _receiver) external {
    BorrowLogic.borrow(
        getLenderStorage(), BorrowParams({ agent: msg.sender, asset: _asset, amount:
_amount, receiver: _receiver })
    );
}
```

Figure 26.1: contracts/lendingPool/Lender.sol#L65-L69

Exploit Scenario

A borrower creates tiny borrow positions (1 wei each) that become liquidatable. The gas costs for liquidation (which involve multiple operations including debt token burns, transfers, oracle calls, and complex calculations) would far exceed the liquidation value plus bonus for such small positions. Since liquidators will avoid liquidating these positions as they would lose money, these positions will remain in the system as bad debt, though the total impact is limited by the number of borrowable assets.

Recommendations

Short term, implement a minimum borrow amount check in the `borrow` function that ensures that the borrowed amount would always be economically viable to liquidate.

Long term, add invariant tests that ensure the economic rationality of the main protocol operations in all cases.

27. Interest rate manipulation through frequent mints and burns

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CAP-27
Target: `contracts/vault/libraries/VaultLogic.sol`	

Description

The interest rate calculation in `VaultAdapter` depends on the elapsed time since the last update, but this timestamp is reset with every mint, burn, or borrow operation. An attacker can manipulate interest rates by performing frequent mint and burn operations, artificially reducing the elapsed time used in utilization multiplier calculations and dampening interest rate increases.

```
function _updateIndex(IVault.VaultStorage storage $, address _asset) internal {
    if (!_listed($, _asset)) revert AssetNotSupported(_asset);
    $.utilizationIndex[_asset] += utilization($, _asset) * (block.timestamp -
$.lastUpdate[_asset]);
    $.lastUpdate[_asset] = block.timestamp;
}
```

Figure 27.1: Code snippet from `_updateIndex` function (contracts/vault/libraries/VaultLogic.sol#L289-L293)

```
$.utilizationData[_asset].utilizationMultiplier *=
    (1e27 + (1e27 * excess / (1e27 - slopes.kink)) * (_elapsed * $.rate / 1e27));
```

Figure 27.2: Code snippet from `_applySlopes` function (contracts/oracle/libraries/VaultAdapter.sol#L87-L88)

As shown in figure 27.1 the `_updateIndex` function resets `lastUpdate` to the current timestamp with each operation. Figure 27.2 shows how the utilization multiplier calculation relies on `_elapsed` time, which can be artificially reduced through frequent transactions, thereby dampening interest rate increases that should occur during high utilization periods. Zero-value mints or burns make this vulnerability significantly worse, as they allow attackers to reset timestamps with minimal capital requirements while still receiving the full benefit of rate manipulation.

Exploit Scenario

Alice borrows a large amount of an asset, causing high utilization and triggering increased interest rates. To avoid paying these higher rates, she executes a series of small mint and burn transactions. Each operation resets the `lastUpdate` timestamp, causing the elapsed time used in rate calculations to remain small. This manipulation allows Alice to maintain a

large borrowed position while paying significantly lower interest than intended, undermining the protocol's risk management mechanisms.

Recommendations

Short term, disable zero-value mint and burn operations to prevent the most capital-efficient form of this attack, and implement proper rounding semantics by using round-up division for such calculations.

Long term, implement a time-weighted average utilization rate that is resistant to short-term manipulations by using a cumulative rate calculation model that considers historical utilization.

28. Protocol lacks bad debt management mechanisms, risking permanent insolvency

Severity: **Medium**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-CAP-28
Target: `contracts/lendingPool/Lender.sol`	

Description

The protocol lacks mechanisms to handle accumulated bad debt, which can occur when liquidations fail to execute or when positions become economically unviable to liquidate. This becomes especially problematic in conjunction with the protocol's allowance of small, economically unviable borrowing positions that can accumulate as bad debt over time (see TOB-CAP-26).

When bad debt occurs, the loss is not socialized across all users. Instead, the system creates a "last withdrawer" problem where early withdrawers can redeem their full amount, while later withdrawers may be unable to withdraw their funds as the protocol runs out of physical tokens while still reporting sufficient balances in `totalSupplies`.

Exploit Scenario

After an extended market downturn, numerous small borrowing positions become underwater but are too small to be economically viable for liquidators to process due to high gas costs. As a result, these positions accumulate as bad debt in the protocol. Since there is no mechanism to handle or recover this bad debt, the protocol becomes permanently insolvent.

Alice, who monitors on-chain activity closely, notices this bad debt accumulation and quickly withdraws her funds, receiving her full amount. Later, when Bob attempts to withdraw, he discovers that insufficient physical tokens remain in the vault to fulfill his withdrawal, despite the system showing he has the right to these funds. Since there is no mechanism to handle or recover this bad debt, Bob becomes the victim of the "last withdrawer" problem and effectively absorbs the losses of the protocol's uncollected bad debt.

Recommendations

Short term, implement an insurance fund that collects a small percentage of interest payments or liquidation fees to cover bad debt when it occurs.

Long term, implement a loss socialization mechanism that fairly distributes any unavoidable losses among all users proportional to their deposit size, rather than disadvantaging only the last withdrawers.

29. Reward distribution can be tricked by front-running notify calls

Severity: **Informational**	Difficulty: **Medium**
Type: Timing	Finding ID: TOB-CAP-29
Target: `contracts/token/StakedCap.sol`	

Description

The `notify` function in `StakedCap` can be front-run, allowing attackers to deposit large amounts of CAP tokens immediately before a notification event to receive an unfair portion of rewards. This allows sophisticated users to capture rewards without requiring long-term commitment.

```
function notify() external {
    uint256 total = IERC20(asset()).balanceOf(address(this));
    StakedCapStorage storage $ = getStakedCapStorage();
    if (total > $.storedTotal) {
        uint256 diff = total - $.storedTotal;
        $.totalLocked = lockedProfit() + diff;
        $.storedTotal = total;
        $.lastNotify = block.timestamp;
        emit Notify(msg.sender, diff);
    }
}
```

Figure 29.1: Code snippet from `notify` function (`contracts/token/StakedCap.sol#L58-L69`)

As shown in figure 29.1, the `notify` function detects new rewards and marks them for distribution, updating `lastNotify` to the current timestamp. These rewards are distributed proportionally based on current share ownership. Therefore, a user who deposits immediately before `notify` is called receives the same proportion of rewards as long-term stakers, undermining the incentive for long-term staking and allowing attackers to optimize reward capture through front-running.

Exploit Scenario

Alice monitors the mempool for upcoming `notify` transactions. When she observes such a transaction, she front-runs it with a large deposit into the `StakedCap` contract. After the notification event completes, Alice instantly becomes entitled to a share of the newly added rewards proportional to her deposit, despite having just joined. She can then withdraw her funds after the vesting period, capturing rewards that would have otherwise gone to long-term stakers.

Recommendations

Short term, add off-chain automation that ensures `notify` is called on a regular basis to prevent a large reward distribution from accumulating.

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
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
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 does not apply 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 Findings

This appendix contains findings that do not have immediate or obvious security implications. However, they may facilitate exploit chains targeting other vulnerabilities, become easily exploitable in future releases, or decrease code readability. We recommend fixing the issues reported here.

1. **Remove TODOs from the code.** There are multiple "`// TODO:`" comments that should be removed before deployment.

2. **Checks-effects-interactions pattern violation in `FeeAuction.buy`**. The buy function in `contracts/feeAuction/FeeAuction.sol#L75-L79` does not follow the checks-effects-interactions pattern, which could potentially lead to reentrancy vulnerabilities.

3. **`PriceOracle` and `RateOracle` could be `abstract` contracts**. The `PriceOracle` and `RateOracle` contracts implement functionality that is meant to be inherited rather than used directly. These should be marked as `abstract` contracts to prevent direct instantiation and to better communicate their intended usage.

4. **Function state mutability can be restricted to `view`**. If `VaultAdapter` is refactored as a `library` as seen in finding 12, several functions in the `RateOracle` contract could be marked as `view` instead of having default mutability:

   - `contracts/oracle/RateOracle.sol#L110` can use `.staticcall` instead of `.call`.
   - `contracts/oracle/RateOracle.sol#L109-L112` can be marked as `view`.
   - `contracts/oracle/RateOracle.sol#L26-L29` can be marked as `view`.
   - `contracts/oracle/RateOracle.sol#L34-L37` can be marked as `view`.

5. **The buy function does an unnecessary callback**. In `contracts/feeAuction/FeeAuction.sol#L64-L79`, the buy function performs an unnecessary callback, potential buyers should be using flashloans. If you want to keep the callback mechanism, this function should be protected with the `nonReentrant` modifier to prevent reentrancy attacks.

6. **Missing parameters in `ZapOFTComposer.constructor` NatSpecs**. In `contracts/zap/ZapOFTComposer.sol#L27-L31`, the NatSpec comments for the constructor do not document all parameters.

7. **The `Minter.__Minter_init` and `FractionalReserve.__FractionalReserve_init` functions are never called.**

These initialization functions are defined but never called in the codebase and should be removed.

8. **Incorrect debt repayment order in `BorrowLogic`.** The `repay` function in `contracts/lendingPool/libraries/BorrowLogic.sol` prioritizes paying principal first, then restaker debt, and interest last, contradicting its own documentation which states interest should be paid first.

9. **Multiple unused imports.** There are many unused imports throughout the codebase that should be removed. You can use VSCode Solidity Inspector extension for *"Inline highlighting in code editor for unused imports"*.
Please note that we have not necessarily vetted this extension or its author. Developers should perform their own due diligence before installing any third-party extensions.

10. **`ICapToken.initialize` and `CapToken.initialize` mismatch.** There's a mismatch between the interface in `contracts/interfaces/ICapToken.sol#L7-L8` and the implementation in `contracts/token/CapToken.sol#L23-L30`.

11. **Some contracts are not using the `_disableInitializers` pattern.** Contracts like `AccessControl`, `Network`, and `NetworkMiddleware` don't follow the recommended OpenZeppelin approach of using `_disableInitializers` function inside the `constructor`.
Reference: https://www.rareskills.io/post/initializable-solidity

12. **Recommend sanity checking the fee.** In `contracts/delegation/providers/symbiotic/NetworkMiddleware.sol#L71-L75`, there is no sanity check for the fee value.

13. **`Vault`, `FractionalReserve`, and `Minter` should be `abstract` contracts.** The `Vault`, `FractionalReserve`, and `Minter` contracts should be declared as `abstract` since they are meant to be inherited from rather than deployed directly.

14. **Like the main oracle, the backup oracle should check for staleness and zero prices**. The backup oracle in `contracts/oracle/PriceOracle.sol#L42` should also check for staleness and zero prices.

15. **`revokeAccess` should not allow the admin to revoke himself.** The `revokeAccess` function allows the admin to revoke their own access, which could lead to locking the protocol without any admin access.

16. **Unbounded array iterations risk out-of-gas failures.** Several functions iterate through uncapped arrays which could revert due to exceeding block gas limits as the arrays grow. It's recommended to set upper bounds on array sizes, or refactor to use mappings instead when possible.

D. Testing Recommendations

This appendix provides a testing strategy that will help the CAP team to improve the project's security posture and detect potential high-severity issues that we were unable to identify during this audit. In the Code Maturity Evaluation, we identified a potential systemic weakness in unit and invariant testing; the following recommendations address this area.

Unit Testing

We believe CAP's existing unit test suite is overly dependent on code coverage as a measure of the unit test suite's thoroughness. Although coverage is an adequate measure for more traditional Web2 projects, smart contracts require a more rigorous methodology.

While using Slither's Mutation Testing can help identify some shortcomings in a test suite after it has been written, we recommend following a more holistic approach due to the inherent limitations of mutation testing. The approach is divided into the following testing angles:

Function-by-function testing

Function-by-function testing is where a test approach is planned out on a function-level basis. This is one of the most fundamental types of testing, and CAP already implements it to some extent.

For each key function, an engineer should identify the following properties and convert them into tests:

Basic function properties

- What kinds of "happy paths" exist for the function? If they cause a state transition, do the unit tests explicitly verify it?
- What kinds of "unhappy paths" exist for the function? Unhappy paths can include scenarios such as authentication failures or swapping without providing enough input tokens, or can otherwise be paths that try to violate one of the function's invariants.
- What kinds of invalid input are there for the function? For example, what happens when the `FeeAuction.buy` function is called with an empty assets array and zero balance tokens? Another example is that adding a duplicate vault to an agent's vault list in the `Delegation` contract should revert.

Arithmetic function properties

- What is the desired rounding behavior for the function? If unsure about rounding behavior, refer to the Rounding Recommendations appendix.

- Is the arithmetic as it is implemented actually correct? Arithmetic-related functions should be verified with a variety of different input values, such as testing the `StakedCapAdapter.price` function with different decimal configurations.
- What are the boundary conditions of the function? Boundary conditions can be approached using two methods: testing boundary inputs (zero, maximum values) and ensuring the output is correct, and ensuring that the outputs are bounded within an expected range. Use unit testing for the former, and fuzzing for the latter.

System integration properties

- What dependencies does this function have on the rest of the system? Are additional happy or unhappy paths present based on the status of other components in the system?
- How can a malicious actor impact the function's behavior indirectly? What would happen if an attacker front-runs or sandwiches a call to the function or one of its dependents?
- How does the function behave if called in an unexpected sequence of calls?

Integration testing

Integration tests address the system as a whole, especially with external components included. One shortcoming in CAP's integration testing is that it uses mocks for Symbiotic's smart contracts. While this is fine for function-level testing that simply requires Symbiotic's contracts to respond in a specific way, we recommend avoiding the use of mocks for integration testing.

External mocks are often built using a partial understanding of the protocol being mocked, leading to behavioral inconsistencies and false assumptions about behavior during the project's testing phase. Catching these inconsistencies earlier using integration testing and before deploying to testnet can reduce the cost of fixing behavioral issues and improve code quality.

Instead of mocks, we recommend using Foundry fork testing against a specific block height for integration testing without having to set up Symbiotic's entire stack locally in Foundry.

Integration tests should verify the different parts of the user journey that trigger interactions with the external smart contracts. Ideally, these tests should also verify against unhappy paths that are expected when interacting with the external component.

Invariant Testing

Invariant testing can be thought of as the "next level" after unit testing. Do not pursue invariant testing until the unit-testing suite is as complete as possible. While invariant testing is valuable, it is less cost effective to find bugs using invariant testing compared to unit testing.

Invariant testing is much more open-ended than unit testing, and a wide variety of invariants can be implemented for CAP. Given the time constraints, we recommend focusing on relatively small, stateless, easy-to-verify invariants. If the invariant is simple enough, CAP can use Halmos to formally verify its correctness instead of fuzzing.

Trail of Bits has identified the following invariants as being relatively simple, valuable, and ideal for stateless verification in CAP:

- **`physicalBalanceMatchesAccounting`**: The physical token balance (`token.balanceOf`) of the Vault contract should match or exceed the accounting balance (`totalSupplies - totalBorrows`).
- **`totalSuppliesExceedTotalBorrows`**: The Vault's `totalSupplies` should always be greater than or equal to `totalBorrows`.
- **`decimalScalingConsistency`**: Decimal scaling operations must produce consistent results across different token denominations. This could also be a good unit test since only a few decimals settings are generally used (6, 8, 18, etc.)
- **`totalDelegationsMatchSum`**: The total delegation for all agents should equal the sum of all individual delegations.
- **`debtTokenSupplyMatchesLenderAccounting`**: For each asset, the sum of circulating debt tokens (principal, interest, restaker) should match the total debt tracked in the `Lender` contract.
- **`withdrawalsAlwaysPossible`**: Given a specific vault state, withdrawals must always be possible up to the reported available amount.

E. Rounding Recommendations

This appendix provides guidance on how arithmetic calculations should be rounded to ensure that the CAP protocol remains solvent. These recommendations should be followed for all arithmetic operations relating to functionality such as token deposits, withdrawals, transfers, interest accrual, and fee calculations.

Rounding guidance is necessary because certain arithmetic operations lose precision; this means that the final result of an operation is an approximation instead of the true result. The difference between an approximated result and the true result is called epsilon. Given CAP's role as a decentralized yield-bearing stablecoin protocol with lending functionality, it is critical to be aware of where epsilon will manifest in arithmetic calculations and correctly account for it to prevent insolvency.

Trail of Bits maintains roundme, a human-assisted tool for determining rounding directions for individual operations in an arithmetic formula. After reading the sections below, engineers should determine which direction high-level formulas should round in, then use roundme to determine the rounding directions of the individual arithmetic operations that make up the formula.

Recommendations for Determining Rounding Directions

- **Identify where funds are sent from the protocol to users and ensure that the associated operations round down their results.** This will minimize the number of tokens that are sent out to users.
- **Identify where funds are sent from users to the protocol and ensure that the associated operations round up their results.** This will maximize the number of tokens that are kept by the pool.
- **Identify where funds are transferred from the protocol for fees and ensure that the associated operations round up their results.** This will maximize the number of tokens that stay inside the protocol.
- **Trace each variable backward through the project to verify that the rounding directions used to generate it are consistent.** This will help reduce the time it takes to debug more complex rounding issues.

Fixed point math best practices

CAP should consistently use a proper fixed-point math library throughout the codebase to handle decimal arithmetic with precision.

- CAP is already using WadRayMath. This is good practice, but CAP should ensure that this or a similar fixed-point library is used consistently across all components.

- Use the appropriate conversion functions (`wadToRay` and `rayToWad`) when switching between precision domains.
- When implementing new mathematical operations, add them to the central `WadRayMath` library rather than recreating similar functions across different contracts.
- Implement explicit rounding direction functions in the math library:
    - Use `divRoundUp` or a similar function when calculating amounts users need to repay to ensure protocol solvency.
    - Use `divRoundDown` or a similar function when calculating interest earned by lenders to maintain conservative projections.
- Document the expected rounding behavior for each calculation to ensure consistent implementation across the protocol.

By standardizing on `WadRayMath` throughout the codebase for all decimal calculations, CAP can avoid subtle rounding issues and inconsistencies that might otherwise arise from mixing different approaches to decimal arithmetic.

Reasoning about Rounding Directions

To determine whether an operation should round up or down, one must reason about how the outcome of the operation impacts token transfers.

Consider the following equation used by a vault contract to calculate how many `share` tokens a user should be credited for a deposit of `assets`:

$$sharesToMint = (\frac{assets * supply}{totalAssets})$$

In order for this calculation to benefit the vault, `sharesToMint` should be rounded down. Variables that need to be rounded down are annotated with (↘):

$$sharesToMint\searrow = (\frac{assets\searrow * supply\searrow}{totalAssets\nearrow})\searrow$$

These variables must now be traced back through the contract to verify they are rounded correctly. Sometimes a system is designed in such a way that not all variables can be rounded correctly; in cases like this, the protocol must be redesigned to facilitate correct rounding.

Integer Arithmetic Primitives

In integer arithmetic, division operations lose precision due to rounding. Therefore, division operations need two primitive operators to correctly approximate results: one where the quotient is rounded up and one where the quotient is rounded down.

Rounding down is the default behavior:

$$div_{down}(a, b) = \frac{a}{b}$$

The following is one approximation for rounding up an integer division operation:

$$div_{up}(a, b) = \begin{cases} \frac{a}{b}, & a \bmod b = 0 \\ \frac{a}{b} + 1, & a \bmod b \neq 0 \end{cases}$$

div_{down} and div_{up} can then be used as primitives to build the following:

- $mulDiv_{up}$
- $mulDiv_{down}$

Example: Rounding in Current Price Calculation in FeeAuction

The `currentPrice` function in the FeeAuction contract illustrates the importance of proper rounding:

```
function currentPrice() public view returns (uint256 price) {
    FeeAuctionStorage storage $ = get();
    uint256 elapsed = block.timestamp - $.startTimestamp;
    if (elapsed > $.duration) elapsed = $.duration;
    price = $.startPrice * (1e27 - (elapsed * 1e27 / $.duration)) / 1e27;
}
```

Here, the rounding direction in (`elapsed * 1e27 / $.duration`) matters. Since a higher result in this division will produce a lower final price, the division should round down to ensure that the protocol receives at least the minimum price. The current implementation correctly rounds down in this division operation (per Solidity's default integer division behavior).

However, in the final division by 1e27, the function should round up rather than down. The current implementation uses Solidity's default rounding, which could result in the protocol receiving slightly less than the intended minimum price due to precision loss.

To ensure that the protocol consistently receives at least the calculated minimum price, the final division should use explicit round-up logic.

F. 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.

From April 28 to April 30, 2025, Trail of Bits reviewed the fixes and mitigations implemented by the CAP Labs 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 29 issues described in this report, CAP Labs has resolved 24 issues, and has accepted risk on five issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Missing input validation in FeeAuction.buy allows payment without asset transfer	Medium	Resolved
2	Incorrect oracle staleness period leads to price feed DoS	High	Resolved
3	Vaults can be added to the middleware multiple times, leading to double-counted delegations	Medium	Resolved
4	StakedCap yield distribution DoS through timer manipulation	Medium	Resolved
5	Inconsistent balance tracking in vault creates DoS for asset borrowing	High	Resolved
6	Unsafe asset removal without borrow validation	Medium	Resolved
7	Initial auction start price can be lower than minimum start price	Low	Resolved
8	Immediate liquidation possible when reducing an agent's liquidation threshold	Informational	Risk Accepted

9	FeeAuction's buy function allows purchasing at sub-optimal prices by adding tokens during an active auction	Low	Risk Accepted
10	Missing event emissions for critical parameter changes in VaultAdapter	Informational	Resolved
11	VaultAdapter's setSlopes function permits zero or maximum kink values that cause division by zero	Informational	Resolved
12	Unvalidated _vault address in VaultAdapter allows interest rate manipulation	High	Resolved
13	Fee auction allows buying zero assets, leading to front-running attacks	High	Resolved
14	Discrepancy between health calculation and slashable collateral computation	High	Resolved
15	Reward distribution enables front-running attacks and reward siphoning	High	Resolved
16	Unaccounted external vault investment losses can create withdrawal shortfalls	Medium	Risk Accepted
17	Wrong capTokenDecimals value used in StakedCapAdapter.price causes inaccurate prices	High	Resolved
18	Liquidation mechanism can be permanently disabled by misconfigured grace and expiry periods	Medium	Resolved
19	Oracle update front-running allows extraction of value from vaults	Medium	Resolved
20	Asset removal does not reset isBorrowing flag for agents	Informational	Resolved

21	Invalid network registration in Delegation.registerNetwork can cause DoS	Low	Resolved
22	Lack of verification on ERC4626 vault withdrawal amounts	Informational	Resolved
23	Agent LTV can be configured equal to or higher than liquidation threshold	Low	Resolved
24	ZapOFTComposer._lzCompose may fail with USDT	Informational	Resolved
25	Fee auction allows assets to be purchased for free	Informational	Resolved
26	Small borrows can create economically unviable liquidatable positions leading to bad debt accumulation	Low	Resolved
27	Interest rate manipulation through frequent mints and burns	Low	Risk Accepted
28	Protocol lacks bad debt management mechanisms, risking permanent insolvency	Medium	Resolved
29	Reward distribution can be tricked by front-running notify calls	Informational	Risk Accepted

Detailed Fix Review Results

TOB-CAP-1: Missing input validation in FeeAuction.buy allows payment without asset transfer
Resolved in PR #92. The buy function now reverts when passed a zero-length assets array.

TOB-CAP-2: Incorrect oracle staleness period leads to price feed DoS
Resolved in PR #93. The price oracle's staleness check has been modified from a single global setting to a per-asset configuration.

TOB-CAP-3: Vaults can be added to the middleware multiple times, leading to double-counted delegations
Resolved in PR #94. The `_verifyVault` function now checks that the vault is not already registered.

TOB-CAP-4: StakedCap yield distribution DoS through timer manipulation
Resolved in PR #95. The `notify` function can no longer be called during the lock period.

TOB-CAP-5: Inconsistent balance tracking in vault creates DoS for asset borrowing
Resolved in PR #96. A new `_verifyBalance` function has been added that checks if the vault has enough available balance before withdrawals and borrows.

TOB-CAP-6: Unsafe asset removal without borrow validation
Resolved in PR #97. The `removeAsset` function now ensures that the total supply of the asset to be removed is 0.

TOB-CAP-7: Initial auction start price can be lower than minimum start price
Resolved in PR #92. The `setStartPrice` function now reverts if provided a start price lower than the minimum.

TOB-CAP-8: Immediate liquidation possible when reducing an agent's liquidation threshold
Risk accepted. CAP Labs has acknowledged the risk of this finding and has stated that it is an intentional design choice.

TOB-CAP-9: FeeAuction's buy function allows purchasing at sub-optimal prices by adding tokens during an active auction
Risk accepted. CAP Labs has acknowledged the risk of this finding and has stated that it is an intentional design choice.

TOB-CAP-10: Missing event emissions for critical parameter changes in VaultAdapter
Resolved in PR #98. Events for setting slopes and limits have been added.

TOB-CAP-11: VaultAdapter's setSlopes function permits zero or maximum kink values that cause division by zero
Resolved in PR #99. A validation check has been added in the `setSlopes` function.

TOB-CAP-12: Unvalidated _vault address in VaultAdapter allows interest rate manipulation
Resolved in PR #100. The `utilizationData` mapping in `IVaultAdapter` has been refactored to ensure that each vault-asset pair has utilization data, rather than just the asset.

TOB-CAP-13: Fee auction allows buying zero assets, leading to front-running attacks
Resolved in PR # 92. The buy function now has a `_maxPrice` slippage parameter to protect the buyer from front-running attacks.

TOB-CAP-14: Discrepancy between health calculation and slashable collateral computation
Resolved in PR #115. Health and coverage is now calculated using the minimum of the slashable collateral and current delegation, addressing any potential discrepancy between the two.

TOB-CAP-15: Reward distribution enables front-running attacks and reward siphoning
CAP Labs acknowledged the finding and states that it will be addressed by using a single delegator-to-agent relationship. This fix is an operational-level policy, and does not have associated code to enforce the single delegator-to-agent relationship.

TOB-CAP-16: Unaccounted external vault investment losses can create withdrawal shortfalls
Risk accepted. CAP Labs acknowledged the finding.

TOB-CAP-17: Wrong capTokenDecimals value used in StakedCapAdapter.price causes inaccurate prices
Resolved in PR #105. The `price` function is now correctly using the proper scaling factor to calculate `capTokenDecimals`.

TOB-CAP-18: Liquidation mechanism can be permanently disabled by misconfigured grace and expiry periods
Resolved in PR #106. The `initialize` function of the `Lender` contract now has a validation check to ensure that `_grace` is less than or equal to `_expiry`.

TOB-CAP-19: Oracle update front-running allows extraction of value from vaults
Resolved in PR #109. This PR addresses the finding by implementing a fixed minting fee, in addition to the basket ratio fee.

TOB-CAP-20: Asset removal does not reset isBorrowing flag for agents
Resolved in PR #110. The `validateRemoveAsset` function now also checks that `interestDebtToken` and `restakerDebtToken`, in addition to `principalDebtToken`, are all 0.

TOB-CAP-21: Invalid network registration in Delegation.registerNetwork can cause DoS
Resolved in commit 13a1b06. The `registerNetwork` function now has a check to ensure that the `_network` address is not the zero address.

TOB-CAP-22: Lack of verification on ERC4626 vault withdrawal amounts
Resolved in PR #104. The `divest` function now reverts if the `FractionalReserve` contract redeems fewer assets than it is loaned.

TOB-CAP-23: Agent LTV can be configured equal to or higher than liquidation threshold
Resolved in PR #111. The `Delegation` contract now has an adjustable buffer between LTV and LT.

TOB-CAP-24: ZapOFTComposer._lzCompose may fail with USDT
Resolved in PR #108. The `_lzCompose` function now correctly uses `forceApprove`.

TOB-CAP-25: Fee auction allows assets to be purchased for free
Resolved in PR #92. The buy function `price` now decays to 10% of the start price instead of 0.

TOB-CAP-26: Small borrows can create economically unviable liquidatable positions leading to bad debt accumulation
Resolved in PR #107. The `Lender` contract now has a minimum borrow amount (`minBorrow`).

TOB-CAP-27: Interest rate manipulation through frequent mints and burns
Risk accepted.

TOB-CAP-28: Protocol lacks bad debt management mechanisms, risking permanent insolvency
Resolved in PR #109. An insurance fund has been added that collects fees in mint, burn and redeem operations and the corresponding new fee-related logic is introduced (`minMintFee`).

TOB-CAP-29: Reward distribution can be tricked by front-running notify calls
Risk accepted. CAP Labs has acknowledged the risk of this finding and has stated that it is an intentional design choice.

G. 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.
Risk Accepted	The client has accepted the risk of the finding
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.