CAP Audit Report

Prepared by CAP Security Cartel

Version 1.2

Lead Auditors

0xleastwood

devtooligan

zigtur

June 23, 2025

Contents

1 About CAP Security Cartel

CAP Security Cartel is a Web3 security organization aiming to protect projects and their partners against malicious actors. We aim to provide holistic improvements to a project's security stack in a safe and realiable way.

0xleastwood, devtooligan, and zigtur and are independent smart contract security researchers. All three are Security Researchers at Spearbit, and have a background in competitive audits and live vulnerability disclosures. As a team, they are working together to conduct audits on Cap Labs' codebase, and are operating as part of the larger "Cap Security Cartel".

0xleastwood can be reached on Twitter at @0xleastwood, zigtur can be reached on Twitter at @zigtur and devtooligan can be reached on Twitter at @devtooligan.

2 Disclaimer

The CAP Security Cartel team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Cap Labs is developing CAP, a stablecoin protocol to outsource yield generation in a programmatically secured manner. The protocol issues cUSD, a stablecoin backed 1:1 by bluechip stablecoins, and coordinates three key actors: minters (stablecoin holders), operators (institutions and DeFi protocols that generate yield), and restakers (who provide shared security through restaked ETH). cUSD can be redeemed for any of it's available assets (i.e. assets not being borrowed). Built on shared security marketplaces like Symbiotic, CAP enables scalable yield generation regardless of market conditions. For more information, visit Cap Labs.

5 Audit Scope

The audit was started on commit ad1f8d7 for **15** days and a subsequent fix review was conducted on final commit 9819fd7 that went for **2** days.

The review scope included the complete CAP protocol smart contract suite, covering the following key components:

- **Access Control**: Permission management and authentication systems
- **Delegation**: Symbiotic network integration and middleware components
- **Lending Pool**: Core lending functionality including borrow/repay logic, liquidations, reserves, and validation
- **Oracle System**: Price and rate oracles with multiple adapter implementations (Chainlink, Aave, etc.)
- **Token Infrastructure**: CAP token, staked CAP, debt tokens, and cross-chain OFT implementations
- **Vault System**: Asset management, minting/burning, and fractional reserve logic

- **Fee Management**: Fee auction and receiver mechanisms
- **Cross-Chain Integration**: LayerZero OFT composer and zap functionality
- **Storage Utilities**: Upgradeable storage patterns for all major components
- **Interfaces**: Complete interface definitions for all protocol components

The audit covered **112 contract files** across 10 major modules, including all related external dependencies and integrations.

6 Executive Summary

Over the course of **15** days, the CAP Security Cartel team conducted an audit on the CAP smart contracts provided by CAP. In this period, a total of **27** issues were found.

Summary of the audit findings and any additional executive comments.

Summary

Project Name	CAP
Repository	cap-contracts
Initial Commit	ad1f8d772585...
Final Commit	9819fd7029e5...
Audit Timeline	April 14th - May 2nd
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	2
Medium Risk	8
Low Risk	8
Informational	7
Gas Optimizations	2
Total Issues	27

Summary of Findings

[H-1] Interest double counting in `maxRealization`	Verified Fix
[H-2] Interest rate manipulation in `VaultAdapter` due to multiplier calculation	Verified Fix
[M-1] Incorrect asset calculation when burning entire token supply	Verified Fix

[M-2] Incorrect time scaling in interest calculation	Verified Fix
[M-3] Admin can drain invested assets via `rescueERC20` function	Verified Fix
[M-4] Burn fee underestimated due to incorrect denominator	Verified Fix
[M-5] `ERC4626` rounding issues may cause system-wide failures in fractional reserve logic	Verified Fix
[M-6] Mint/burn kink curve overestimate fees due to incorrect formula	Verified Fix
[M-7] Interest rate manipulation for new borrowers	Verified Fix
[M-8] First-depositor share price inflation attack	Acknowledged
[L-1] Incorrect `AccessControl` address causes access control bypass	Verified Fix
[L-2] Repayment and liquidation may leave debt below minimum borrow threshold	Verified Fix
[L-3] Zero amount rejection prevents redemption of small balances	Verified Fix
[L-4] Inconsistent amount calculation for initial mint	Verified Fix
[L-5] Insufficient balance verification in vault `burn` and `redeem` functions	Verified Fix
[L-6] Insufficient divestment in burn function excludes fees	Verified Fix
[L-7] Incorrect ratio calculation causes excessive fees on first mint	Verified Fix
[L-8] Interest rate inaccuracy due to averaged utilization calculations	Acknowledged
[I-1] Redundant assignment of `lastUpdated` variable	Verified Fix
[I-2] Unsafe casting of potentially negative Chainlink price data	Verified Fix
[I-3] Missing vault existence check in `invest` function	Verified Fix
[I-4] Borrows can be griefed when token reserves are low	Verified Fix
[I-5] Emergency liquidation bonus depends on liquidation start timestamp	Verified Fix
[I-6] Inconsistent reserve pausing implementation	Verified Fix
[I-7] Missing parameter validation	Verified Fix
[G-1] Redundant contains check in agent addition	Verified Fix
[G-2] Avoid redundant contains check before add in `EnumerableSet`	Verified Fix

7 Findings

7.1 High

7.1.1 Interest double counting in `maxRealization`

Description: In the `maxRealization()` function, the maximum interest that can be realized is calculated as the difference between the total debt token supply and the reserve's debt:

```
if (totalDebt > reserve.debt) {
    realization = totalDebt - reserve.debt;
}
```

However, this calculation does not account for previously tracked unrealized interest from restakers. The issue occurs because:

1. The total debt token supply (`totalDebt`) includes all debt tokens, including those minted for unrealized restaker interest
2. The reserve's debt (`reserve.debt`) does not include unrealized restaker interest
3. This means the difference between them includes unrealized interest from restakers

When `realizeInterest()` calls `maxRealization()` and subsequently borrows from the vault, it will count unrealized interest from restakers again, even though this interest has already been minted as debt tokens and will be paid to restakers when repaid.

This significantly affects the accounting system and could lead to double realization of interest when `realizeInterest()` is called after some unrealized restaker interest has accumulated.

Recommendation: The interest accounting mechanism must be reviewed such that the unrealized restaker interest is not being accounted twice. This can be done by tracking the total unrealized interest and subtracting it from `realization` in the `maxRealization()` calculation.

```
function maxRealization(ILender.LenderStorage storage $, address _asset)
    internal
    view
    returns (uint256 realization)
{
    ILender.ReserveData storage reserve = $.reservesData[_asset];
    uint256 totalDebt = IERC20(reserve.debtToken).totalSupply();
    uint256 reserves = IVault(reserve.vault).availableBalance(_asset);

    if (totalDebt > reserve.debt) {
        realization = totalDebt - reserve.debt - reserve.totalUnrealizedInterest;
    }
    if (reserves < realization) {
        realization = reserves;
    }
}
```

CAP Labs: Fixed in PR 145. The protocol now tracks `totalUnrealizedInterest` in the `ReserveData` struct.

CAP Security Cartel: Verified fix.

7.1.2 Interest rate manipulation in `VaultAdapter` due to multiplier calculation

Summary: The `VaultAdapter` contract contains a vulnerability in the `_applySlopes()` function where the interest rate can be manipulated by repeatedly calling the `rate()` function. When consecutive calls to `rate()` occur in the same block, the elapsed time parameter is zero, which leads to an incorrect multiplier calculation that allows attackers to drive interest rates to artificially high or low values.

Description: In the `VaultAdapter` contract, the `_applySlopes()` function calculates interest rates based on asset utilization and a multiplier factor. When utilization exceeds the kink point, the multiplier increases over time, and when utilization is below the kink point, the multiplier decreases over time.

The issue appears in how the multiplier is updated when `_elapsed` is zero, which happens when `rate()` is called multiple times in the same block. The relevant code:

```
if (_utilization > slopes.kink) {
    uint256 excess = _utilization - slopes.kink;
    utilizationData.multiplier *= (1e27 + (1e27 * excess / (1e27 - slopes.kink)) * (_elapsed * $.rate /
↪   1e27));
    // ...
} else {
    utilizationData.multiplier /=
        (1e27 + (1e27 * (slopes.kink - _utilization) / slopes.kink) * (_elapsed * $.rate / 1e27));
    // ...
}
```

When `_elapsed` is zero, the terms `(_elapsed * $.rate / 1e27)` evaluate to zero, resulting in:

- For utilization above kink: `utilizationData.multiplier *= 1e27`
- For utilization below kink: `utilizationData.multiplier /= 1e27`

This means each call to `rate()` within the same block essentially multiplies or divides the multiplier by `1e27`, allowing the caller to manipulate `utilizationData.multiplier` to be at `$.maxMultiplier` or `$.minMultiplier`.

Recommendation: To fix this vulnerability, the `_applySlopes()` function should be modified to multiply or divide `utilizationData.multiplier` by `1e27` depending on whether or not it is above or below `slopes.kink`. This will also handle the case where `_elapsed` is zero.

CAP Labs: Fixed in PR 127 by implementing the suggested recommendation.

CAP Security Cartel: Verified fix.

7.2 Medium

7.2.1 Incorrect asset calculation when burning entire token supply

Description: The contract's burn mechanism incorrectly calculates the amount of assets to return when the entire supply of tokens is being burned. The current implementation uses a relative proportion calculation that fails to account for the edge case of burning the entire supply, resulting in asset tokens being stranded in the contract. This effectively causes users to lose funds when the protocol has accrued yield or has experienced slashing events.

In the `_amountOutBeforeFee` function in MinterLogic.sol, the logic for burning tokens has a flaw when the entire supply is being burned:

```
if (params.amount == capSupply) {
    newRatio = RAY_PRECISION;
    amount = params.amount * assetDecimalsPow / capDecimalsPow;
} else {
    // ... regular burn logic ...
}
```

When burning the entire supply, the function scales the amount of cap tokens being burned by the decimal difference between the cap token and the asset. This approach does not accurately reflect the actual asset value in the contract, as it ignores any yield accrual or slashing events that may have occurred since the tokens were minted.

The correct approach when burning the entire supply would be to return all assets of that type remaining in the contract, since there would be no other token holders with claims on those assets.

Proof of Concept: The provided PoC demonstrates two scenarios:

1. When the contract has been slashed (lost assets), burning the entire supply fails because the calculation expects more assets than exist in the contract.
2. When the contract has accrued yield, burning the entire supply leaves assets stranded in the contract.

```
function test_burn_all() public {
    _fundUser();

    vm.startPrank(user);
    uint shsReceived = cUSD.mint(address(usdt), 100e6, 0, user, block.timestamp);

    // slashing example -- cannot burn all shares because they are overvalued
    _slash(address(cUSD), 10e6); // slash 10 USDT from the vault
    vm.expectRevert(); // reverts because there is not enough assets in the vault
    cUSD.burn(address(usdt), shsReceived, 0, user, block.timestamp);


    // yield example -- burning all shares leaves totalSupply == 0 and totalAssets > 0
    _addYield(address(cUSD), 20e6); // add back the 10 slashed above, plus 10 more
    _transferFeeSharesToUser(); // transfer fee shares from insurance fund to user so we can burn 100%
↪   of the supply
    cUSD.burn(address(usdt), cUSD.balanceOf(user), 0, user, block.timestamp);

    assertEq(cUSD.totalSupply(), 0); // total supply == 0
    assertEq(usdt.balanceOf(address(cUSD)), 10000000); // assets > 0
}
```

Recommendation: Modify the `_amountOutBeforeFee` function to use `assetValue` when burning the entire supply:

```
if (params.amount == capSupply) {
    newRatio = RAY_PRECISION;
-   amount = params.amount * assetDecimalsPow / capDecimalsPow;
+   amount = assetValue * assetDecimalsPow / assetPrice;
} else {
```

```
    // ... regular burn logic ...
}
```

This change ensures that when a user burns the entire supply, they receive assets based on the actual asset value in the contract, properly accounting for any yield accrual or slashing events that have occurred. Using `assetValue` is appropriate here because it represents the true value of assets allocated to the token, respecting the buffering system rather than directly checking token balances.

CAP Labs: Fixed in PR 134 and implemented as suggested in the recommendation.

CAP Security Cartel: Verified fix. Recommendation has been applied.

7.2.2 Incorrect time scaling in interest calculation

Description: The `accruedRestakerInterest()` function in the `ViewLogic` contract incorrectly scales the interest rate calculation by not accounting for the yearly to per-second rate conversion. This results in significantly inflated interest accrual, potentially causing users to see incorrect balances and leading to system insolvency.

The `accruedRestakerInterest()` function calculates the accrued interest by multiplying the debt balance by the rate and elapsed time:

```
function accruedRestakerInterest(ILender.LenderStorage storage $, address _agent, address _asset)
    public
    view
    returns (uint256 accruedInterest)
{
    ILender.ReserveData storage reserve = $.reservesData[_asset];
    uint256 totalInterest = IERC20(reserve.debtToken).balanceOf(_agent);
    uint256 rate = IOracle($.oracle).restakerRate(_agent);
    uint256 elapsedTime = block.timestamp - reserve.lastRealizationTime[_agent];

    accruedInterest = totalInterest * rate * elapsedTime / 1e27;
}
```

The issue is that `rate` is defined as a yearly interest rate (as indicated in the specification and comments), but it's being applied directly to `elapsedTime` which is in seconds. This means the calculated interest is approximately 31.5 million times higher than intended (365 days * 24 hours * 60 minutes * 60 seconds).

For example, if the yearly rate is 5% (0.05 encoded as 5 * 10^25 in ray format), and a week passes, the function would calculate interest as if the rate were 5% per second rather than 5% per year, resulting in an astronomical interest amount.

Recommendation: Modify the `accruedRestakerInterest()` function to properly scale the yearly rate to a per-second rate by dividing by the number of seconds in a year:

```
function accruedRestakerInterest(ILender.LenderStorage storage $, address _agent, address _asset)
    public
    view
    returns (uint256 accruedInterest)
{
    ILender.ReserveData storage reserve = $.reservesData[_asset];
    uint256 totalInterest = IERC20(reserve.debtToken).balanceOf(_agent);
    uint256 rate = IOracle($.oracle).restakerRate(_agent);
    uint256 elapsedTime = block.timestamp - reserve.lastRealizationTime[_agent];

-   accruedInterest = totalInterest * rate * elapsedTime / 1e27;
+   // 365.25 days * 24 hours * 60 minutes * 60 seconds = 31557600 seconds per year
+   uint256 SECONDS_PER_YEAR = 31557600;
+   accruedInterest = totalInterest * rate * elapsedTime / (1e27 * SECONDS_PER_YEAR);
}
```

Alternatively, make the constant a state variable to ensure consistent usage across the system:

```
+ // In the contract or library context
+ uint256 private constant SECONDS_PER_YEAR = 31557600;

function accruedRestakerInterest(ILender.LenderStorage storage $, address _agent, address _asset)
    public
    view
    returns (uint256 accruedInterest)
{
    // ...
-   accruedInterest = totalInterest * rate * elapsedTime / 1e27;
+   accruedInterest = totalInterest * rate * elapsedTime / (1e27 * SECONDS_PER_YEAR);
}
```

CAP Labs: Fixed in PR 133.

CAP Security Cartel: Verified fix.

7.2.3 Admin can drain invested assets via `rescueERC20` function

Description: The `rescueERC20()` function is designed with a protection mechanism that prevents the removal of listed assets from the vault.

```
function rescueERC20(IVault.VaultStorage storage $, address _asset, address _receiver) external {
    if (_listed($, _asset)) revert AssetNotRescuable(_asset);
    IERC20(_asset).safeTransfer(_receiver, IERC20(_asset).balanceOf(address(this)));
    emit RescueERC20(_asset, _receiver);
}
```

This function checks if the asset is listed in the vault's supported assets using the `_listed()` check. If it is listed, the function reverts with an `AssetNotRescuable` error.

However, when assets are invested in external ERC4626 vaults through `FractionalReserveLogic.invest()`, the vault receives share tokens representing its share in the external vault. These share tokens are not added to the list of supported assets, making them eligible for "rescue" via `rescueERC20()`.

This oversight allows an admin to call `rescueERC20()` with the address of the ERC4626 share tokens and transfer them to any address, effectively draining all funds deposited by liquidity providers.

Recommendation: Update `FractionalReserveLogic.invest()` to track external vaults in which funds are invested. Then, add checks in `VaultLogic.rescueERC20()` to ensure that the token to rescue is not one of these external vaults.

CAP Labs: Fixed in PR 132 by integrating `EnumerableSet` to efficiently track vaults.

CAP Security Cartel: Verified fix.

7.2.4 Burn fee underestimated due to incorrect denominator

Description: The `_applyFeeSlopes()` function implements an incorrect calculation in the burn fee rate formula when the ratio is below the burn kink ratio. The current implementation uses `(RAY_PRECISION - fees.burnKinkRatio)` as the denominator, but this doesn't align with the expected behavior for a fee curve calculation.

The issue lies in the following calculation:

```
if (params.ratio < fees.optimalRatio) {
    if (params.ratio < fees.burnKinkRatio) {
        uint256 excessRatio = fees.burnKinkRatio - params.ratio;
        rate = fees.slope0 + (fees.slope1 * excessRatio / (RAY_PRECISION - fees.burnKinkRatio));
```

For a kinked fee curve design, the denominator should be the `burnKinkRatio` itself, not `(RAY_PRECISION - fees.burnKinkRatio)`. The current formula generates a much smaller fee rate than intended when the ratio is below the kink.

Recommendation: Modify the calculation to use the correct denominator:

```
if (params.ratio < fees.optimalRatio) {
    if (params.ratio < fees.burnKinkRatio) {
        uint256 excessRatio = fees.burnKinkRatio - params.ratio;
-       rate = fees.slope0 + (fees.slope1 * excessRatio / (RAY_PRECISION - fees.burnKinkRatio));
+       rate = fees.slope0 + (fees.slope1 * excessRatio / fees.burnKinkRatio);
```

CAP Labs: Fixed in PR 131 and implemented as suggested in the recommendation.

CAP Security Cartel: Verified fix.

7.2.5 `ERC4626` rounding issues may cause system-wide failures in fractional reserve logic

Summary: The `FractionalReserveLogic.divest()` function reverts when `redeemedAssets < loanedAssets`, which can cause system-wide failures due to `ERC4626` rounding issues. This vulnerability can break critical operations including borrows, repayments, and liquidations when the vault is nearly depleted.

Description: In the `FractionalReserveLogic` contract, the `divest()` function contains conditional logic that transfers excess yield to the fee auction when `redeemedAssets > loanedAssets`, but reverts with `LossFromFractionalReserve` when `redeemedAssets < loanedAssets`:

```
if (redeemedAssets > loanedAssets) {
    IERC20(_asset).safeTransfer($.feeAuction, redeemedAssets - loanedAssets);
} else if (redeemedAssets < loanedAssets) {
    revert LossFromFractionalReserve(_asset, $.vault[_asset], loanedAssets - redeemedAssets);
}
```

This creates the following issues:

1. `ERC4626` deposit operations inherently round down when calculating shares, which means that redeeming all shares may return slightly fewer assets than initially deposited.
2. The system skims interest to the fee auction, further exacerbating the issue as it reduces the assets available in the vault.
3. When `BorrowLogic.realizeRestakerInterest()` attempts to borrow from the vault, it relies on `IVault(reserve.vault).availableBalance(_asset)`, which may report more assets than can actually be redeemed due to the rounding issues.
4. When the vault is nearly depleted (most assets are borrowed), these rounding discrepancies can trigger the `LossFromFractionalReserve` revert, blocking critical system functions.

Impact Explanation: The impact is medium. When triggered, this issue causes system-wide failures affecting core protocol functionality:

1. Borrows become impossible to execute
2. Repayments cannot be processed
3. Liquidations fail to complete

These failures persist until new deposits are made into the vault, potentially creating significant risk during market stress when liquidations are most needed.

This can be mitigated by batching transactions to make a small vault deposit.

Likelihood Explanation: The likelihood is medium. The issue emerges in specific edge cases:

1. When the `ERC4626` vault is nearly depleted
2. When most assets are being borrowed

3. When `realizeRestakerInterest()` is called in this state
4. When rounding errors from `ERC4626` share calculations accumulate

These conditions are likely to occur in normal protocol operation, especially during periods of high utilization.

Recommendation: Consider one of the following changes to address this issue:

1. Modify the `divest()` function to add a small tolerance for rounding errors:

```
if (redeemedAssets > loanedAssets) {
    IERC20(_asset).safeTransfer($.feeAuction, redeemedAssets - loanedAssets);
- } else if (redeemedAssets < loanedAssets) {
+ } else if (redeemedAssets < loanedAssets && loanedAssets - redeemedAssets > ROUNDING_TOLERANCE) {
    revert LossFromFractionalReserve(_asset, $.vault[_asset], loanedAssets - redeemedAssets);
}
```

Then, adjust the `invest()` function to account for potential rounding losses when depositing into `ERC4626` vaults:

```
- uint256 shares = IERC4626($.vault[_asset]).deposit(investAmount, address(this));
+ uint256 preBalance = IERC20(_asset).balanceOf($.vault[_asset]);
+ uint256 shares = IERC4626($.vault[_asset]).deposit(investAmount, address(this));
+ uint256 postBalance = IERC20(_asset).balanceOf($.vault[_asset]);
+ uint256 actualDeposit = postBalance - preBalance;
+ $.loaned[_asset] = actualDeposit;  // Adjust loaned amount to actual deposit
```

2. Alternatively, remove the strict requirement to match redeemed and loaned values exactly:

```
if (redeemedAssets > loanedAssets) {
    IERC20(_asset).safeTransfer($.feeAuction, redeemedAssets - loanedAssets);
- } else if (redeemedAssets < loanedAssets) {
-    revert LossFromFractionalReserve(_asset, $.vault[_asset], loanedAssets - redeemedAssets);
+ } else if (redeemedAssets < loanedAssets) {
+    // Accept small losses from rounding
+    $.loaned[_asset] = redeemedAssets;  // Adjust loaned tracking to match reality
}
```

CAP Labs: Fixed in PR 146.

CAP Security Cartel: Verified fix. After lengthy discussions, because the CAP vault is the only depositor in the ERC4626 vault, any share related rounding will re-coup assets on the last redeem and hence it is only necessary to make a redemption for the remaining share balance on the final divest.

7.2.6 Mint/burn kink curve overestimate fees due to incorrect formula

Description: The fee calculation for the mint curve when the ratio is greater than the optimal ratio is incorrect. The current implementation always accounts for `fees.minMintFee` before applying `slope0` on top of it.

```
function _applyFeeSlopes(IMinter.FeeData memory fees, IMinter.FeeSlopeParams memory params) {
    if (params.mint) {
        rate = fees.minMintFee; // @audit rate always account for minMintFee
        if (params.ratio > fees.optimalRatio) {
            if (params.ratio > fees.mintKinkRatio) {
                uint256 excessRatio = params.ratio - fees.mintKinkRatio;
                rate += fees.slope0 + (fees.slope1 * excessRatio / (RAY_PRECISION -
↪   fees.mintKinkRatio));
            } else {
                rate += fees.slope0 * params.ratio / fees.mintKinkRatio; // @audit add slope0
            }
        }
```

This results in a mint fee curve with the following shape:

Figure 1: Image

However, the following shape should be expected instead:

Figure 2: Image

Recommendation: The `slope0` should only be calculated on the `[fees.optimalRatio, fees.mintKinkRatio]` range to ensure correct shape of the mint fee curve. Similarly, this should also be somewhat true for burns where `slope0` should be calculated on the `[fees.burnKinkRatio, fees.optimalRatio]` range.

CAP Labs: Fixed in PR 130. The fix was also applied to the burn side as well.

CAP Security Cartel: Verified fix.

7.2.7 Interest rate manipulation for new borrowers

Summary: A vulnerability exists in the `VaultAdapter` contract where new borrowers can manipulate interest rates to benefit themselves at the expense of existing borrowers. When an agent makes their first borrow, they increase utilization and affect interest rates for all users, but may not bear the cost of this increase themselves due to the sequence of operations.

Description: The issue occurs in the `rate()` function of the `VaultAdapter` contract. When calculating interest rates, the function updates utilization data based on time elapsed since the last update:

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

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

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

The vulnerability arises from the sequence of operations in the broader protocol:

1. A new borrower borrows funds from the vault, increasing spot utilization.
2. Upon repayment, the borrower bears no cost for the borrow as tokens were initially minted on the updated index.
3. After repayment, the debt token's `$.interestRate` remains the same as when the borrow was made, keeping utilization elevated when this does not accurately reflect the vault's utilized asset amount.

This sequence means that the interest rate increase caused by the new borrower's actions is charged to existing debt token holders. This creates a scenario where:

1. A new agent could borrow to maximum utilization.
2. Repay in the same block.
3. Pay zero interest themselves.

4. But cause other borrowers to pay increased interest rates due to the temporary spike in utilization on future blocks.
5. This persists until the debt token's index is updated again, however, the attacker could maintain full asset utilization by being the first user to borrow/repay on the vault in each block.

Recommendation: Consider adjusting the sequence of operations so that debt tokens are minted before the utilization index is updated:

```
- // Current flow: Repay  Burn debt tokens  Update index  Transfer assets
+ // Proposed flow: Repay  Transfer assets  Burn debt tokens  Update index
```

CAP Labs: Fixed in PR 129. Debt tokens are only burned after assets have been transferred out to track the correct vault utilization.

CAP Security Cartel: Verified fix.

7.2.8 First-depositor share price inflation attack

Summary: The `StakedCap` contract is vulnerable to a first-depositor inflation attack, where an attacker can manipulate the share price to steal funds from subsequent depositors. This occurs because the contract lacks protection against share price manipulation and does not enforce a minimum deposit amount or use any share price normalization technique.

Description: The `StakedCap` contract implements the `ERC4626` standard for tokenized vaults without additional protections against share price manipulation. The attack works as follows:

1. An attacker becomes the first depositor by depositing a minimal amount (e.g., 1 wei) to get 1 share.
2. The attacker then transfers additional tokens directly to the contract and calls `notify()` to signal a yield distribution.
3. After some time, as the locked profit begins to vest, the `shares : assets` exchange rate will grow significantly.
4. With already some divergence in the exchange rate, the depositor can begin to abuse the issue by rounding down their own deposits.
5. When a legitimate user makes a deposit, if their deposit isn't large enough to mint at least 1 share with the inflated exchange rate, they will receive 0 shares due to rounding down.
6. This effectively transfers the depositor's assets to existing shareholders (the attacker).

The vulnerability exists because:

- The `totalAssets()` function returns `storedTotal - lockedProfit()`, which means locked profit gradually becomes part of the total assets used for share price calculation.
- There is no minimum deposit requirement or share price normalization.
- The `ERC4626` implementation rounds down when calculating shares to mint, which benefits existing shareholders.

This allows an attacker to steal funds from subsequent depositors. Any user depositing an amount that results in less than 1 share (after the inflation) will lose their entire deposit, with the value accruing to the attacker. The attack can be executed repeatedly, and the attacker can continue to inflate the share price to maximize the stolen amount.

Recommendation: Several strategies can be implemented to mitigate this vulnerability:

1. Implement a minimum deposit amount that is significantly higher than 1 wei.

```
function _deposit(address _caller, address _receiver, uint256 _assets, uint256 _shares) internal
↪  override {
    if (totalSupply() == 0 && _assets < MINIMUM_DEPOSIT) revert DepositTooSmall();
    super._deposit(_caller, _receiver, _assets, _shares);
```

```
    getStakedCapStorage().storedTotal += _assets;
}
```

2. Enforce a minimum share price during the first deposit:

```
function _deposit(address _caller, address _receiver, uint256 _assets, uint256 _shares) internal
↪   override {
    if (totalSupply() == 0) {
        // First deposit - enforce minimum initial shares
        _shares = _assets * INITIAL_EXCHANGE_RATE;
    }
    super._deposit(_caller, _receiver, _assets, _shares);
    getStakedCapStorage().storedTotal += _assets;
}
```

3. Add a virtual offset to both assets and shares to normalize the initial exchange rate:

```
// Add these constants
uint256 private constant VIRTUAL_SHARES = 1e6;
uint256 private constant VIRTUAL_ASSETS = 1e6;

// Override convertToShares
function convertToShares(uint256 assets) public view override returns (uint256) {
    uint256 supply = totalSupply() + VIRTUAL_SHARES;
    uint256 totalAssets = totalAssets() + VIRTUAL_ASSETS;
    return supply == 0 ? assets : assets.mulDiv(supply, totalAssets, Math.Rounding.Down);
}

// Override convertToAssets
function convertToAssets(uint256 shares) public view override returns (uint256) {
    uint256 supply = totalSupply() + VIRTUAL_SHARES;
    uint256 totalAssets = totalAssets() + VIRTUAL_ASSETS;
    return supply == 0 ? shares : shares.mulDiv(totalAssets, supply, Math.Rounding.Down);
}
```

4. Consider seeding the vault with a significant initial deposit (e.g., $10k worth) at deployment time to establish a reasonable initial exchange rate.

CAP Labs: We will seed the initial deposit on contract deployment to establish an initial exchange rate.

CAP Security Cartel: Acknowledged that the team will handle this on deployment. Deposit front-runners take on some risk if they deposit and someone else deposits before any additional assets have started vesting, hence this seems unlikely in the first place.

7.3 Low

7.3.1 Incorrect `AccessControl` address causes access control bypass

Description: In the `Access` contract, the `_checkAccess()` function makes an external call to the `accessControl.checkAccess()` function. The issue is that if this address is uninitialized or points to an EOA, the call will not revert but instead succeed silently, effectively bypassing the access control check.

In Solidity, low-level calls to an empty address don't revert when no return value is expected. In this case, since the function doesn't check any return value from `checkAccess()`, a call to any zero code address would silently succeed, allowing unauthorized access.

This vulnerability could occur if:

- The `accessControl` address is never set correctly during initialization
- The contract at the `accessControl` address is self-destructed
- A deployment script error causes the address to be set incorrectly

Recommendation: Make the following changes to properly handle access checks:

1. Modify the AccessControl contract's `checkAccess()` function to return a boolean instead of reverting:

```
- function checkAccess(bytes4 _selector, address _target, address _caller) external view;
+ function checkAccess(bytes4 _selector, address _target, address _caller) external view returns (bool);
```

2. Update the `_checkAccess()` function to check the return value and explicitly revert if access is not granted:

```
- function _checkAccess(bytes4 _selector) internal view {
-     IAccessControl(getAccessStorage().accessControl).checkAccess(_selector, address(this),
↪  msg.sender);
+ function _checkAccess(bytes4 _selector) internal view {
+     address accessControl = getAccessStorage().accessControl;
+     if (accessControl == address(0)) revert AccessControlNotSet();
+     bool hasAccess = IAccessControl(accessControl).checkAccess(_selector, address(this), msg.sender);
+     if (!hasAccess) revert AccessDenied();
}
```

3. Add relevant error definitions to the IAccess interface:

```
error AccessControlNotSet();
error AccessDenied();
```

This approach explicitly handles the case where the accessControl address is not set and ensures that the access check cannot be bypassed when the contract is deployed.

CAP Labs: Fixed in PR 120.

CAP Security Cartel: Verified fix. A boolean is now returned, ensuring that calling an EOA for access control will not grant access control.

7.3.2 Repayment and liquidation may leave debt below minimum borrow threshold

Description: The protocol enforces a minimum borrow amount when users initially borrow funds through the `validateBorrow()` function:

```
if (params.amount < $.reservesData[params.asset].minBorrow) revert MinBorrowAmount();
```

However, no similar validation exists in the `repay()` or `liquidate()` functions to ensure that partial repayments or liquidations don't leave the remaining debt below this minimum threshold. This could result in small-sized loans that are:

1. Potentially uneconomical to liquidate due to gas costs versus liquidation rewards
2. Vulnerable to various small-loan attack vectors, including Sybil attacks where an attacker creates many small positions
3. Less efficient from a gas and accounting perspective

Recommendation: Add a check in both the `repay()` and `liquidate()` functions to ensure that if the remaining debt after the operation is not zero, it must remain above the minimum borrow threshold. This approach ensures that:

1. Normal repayments either fully clear the debt or maintain the minimum required amount
2. Liquidations can still occur for small positions but generally avoid leaving dust amounts
3. The protocol maintains efficiency by preventing a proliferation of small positions

```
function repay(ILender.LenderStorage storage $, ILender.RepayParams memory params)
    external
    returns (uint256 repaid)
{
    /// Realize restaker interest before repaying
    realizeRestakerInterest($, params.agent, params.asset);

    ILender.ReserveData storage reserve = $.reservesData[params.asset];

    /// Can only repay up to the amount owed
    repaid = Math.min(params.amount, IERC20(reserve.debtToken).balanceOf(params.agent));

+   uint256 remainingDebt = IERC20(reserve.debtToken).balanceOf(params.agent) - repaid;
+   if (remainingDebt > 0 && remainingDebt < $.reservesData[params.asset].minBorrow) {
+       // Either repay full amount or limit repayment to maintain minimum debt
+       if (IERC20(reserve.debtToken).balanceOf(params.agent) <=
↪   $.reservesData[params.asset].minBorrow) {
+           repaid = IERC20(reserve.debtToken).balanceOf(params.agent); // Full repayment
+       } else {
+           repaid = IERC20(reserve.debtToken).balanceOf(params.agent) -
↪   $.reservesData[params.asset].minBorrow;
+       }
+   }

    IDebtToken(reserve.debtToken).burn(params.agent, repaid);
    IERC20(params.asset).safeTransferFrom(params.caller, address(this), repaid);

    // Rest of the function remains unchanged
}
```

A similar check should be added to the `liquidate()` function, with the additional consideration that liquidations should still be possible for very small positions that fall below the minimum threshold:

```
function liquidate(ILender.LenderStorage storage $, ILender.RepayParams memory params)
    external
    returns (uint256 liquidatedValue)
```

```
{
    // Existing validation and calculations

    uint256 maxLiquidation = ViewLogic.maxLiquidatable($, params.agent, params.asset);
    uint256 liquidated = params.amount > maxLiquidation ? maxLiquidation : params.amount;

+   uint256 currentDebt = IERC20($.reservesData[params.asset].debtToken).balanceOf(params.agent);
+   uint256 remainingDebt = currentDebt - liquidated;
+   if (remainingDebt > 0 && remainingDebt < $.reservesData[params.asset].minBorrow) {
+       // For liquidations, prioritize full liquidation over maintaining minimum
+       if (currentDebt <= $.reservesData[params.asset].minBorrow * 2) {
+           liquidated = currentDebt; // Full liquidation
+       } else {
+           liquidated = currentDebt - $.reservesData[params.asset].minBorrow;
+       }
+   }

    liquidated = BorrowLogic.repay(
        $,
        ILender.RepayParams({ agent: params.agent, asset: params.asset, amount: liquidated, caller:
↪   params.caller })
    );

    // Rest of the function remains unchanged
}
```

CAP Labs: Fixed in PR 144.

CAP Security Cartel: Verified fix ensures remaining debt is zero or above threshold.

7.3.3 Zero amount rejection prevents redemption of small balances

Description: In the `redeem()` function of the `VaultLogic` library, there's a check that prevents redeeming when `amountsOut[i]` equals zero.

```
if (params.amountsOut[i] == 0) revert InvalidAmount();
```

This validation creates an issue when the total supply for a given asset approaches zero (e.g., 1 wei). In such cases, the amount of tokens calculated through the redemption formula might round down to zero due to precision loss, causing the transaction to revert with `InvalidAmount`. This effectively prevents users from redeeming very small remaining balances.

The check is redundant since the slippage protection on the previous line already ensures that users receive at least their minimum expected amount:

```
if (params.amountsOut[i] < params.minAmountsOut[i]) {
    revert Slippage(asset, params.amountsOut[i], params.minAmountsOut[i]);
}
```

Recommendation: Remove the zero amount check to allow redemption of very small balances:

```
- if (params.amountsOut[i] == 0) revert InvalidAmount();
```

CAP Labs: Fixed in PR 121.

CAP Security Cartel: Verified fix. The problematic check has been removed.

7.3.4 Inconsistent amount calculation for initial mint

Description: In the `MinterLogic._amountOutBeforeFee()` function, there's an inconsistency in how the output amount is calculated when minting the first tokens (i.e. `capSupply == 0`). The current implementation directly converts the input amount based on decimal differences rather than using the asset's value, which is inconsistent with how amounts are calculated for subsequent mints.

```
if (params.mint) {
    assetValue = params.amount * assetPrice / assetDecimalsPow;
    if (capSupply == 0) {
        newRatio = RAY_PRECISION;
        amount = params.amount * capDecimalsPow / assetDecimalsPow; // Direct decimal conversion
    } else {
        newRatio = (allocationValue + assetValue) * RAY_PRECISION / (capValue + assetValue);
        amount = assetValue * capDecimalsPow / capPrice; // Value-based conversion
    }
}
```

For non-initial mints, the calculation uses `assetValue * capDecimalsPow / capPrice`, which accounts for the price of both the input asset and the cap token. However, for the initial mint, it only adjusts for decimal differences without considering the asset price.

Recommendation: For consistency and to properly account for asset prices during the initial mint, consider updating the calculation to use the same value-based approach:

```
if (params.mint) {
    assetValue = params.amount * assetPrice / assetDecimalsPow;
    if (capSupply == 0) {
        newRatio = RAY_PRECISION;
-       amount = params.amount * capDecimalsPow / assetDecimalsPow;
+       amount = assetValue * capDecimalsPow / assetPrice;
    } else {
        newRatio = (allocationValue + assetValue) * RAY_PRECISION / (capValue + assetValue);
        amount = assetValue * capDecimalsPow / capPrice;
    }
}
```

CAP Labs: Fixed in PR 122 by implementing the suggested recommendation.

CAP Security Cartel: Verified fix.

7.3.5 Insufficient balance verification in vault `burn` and `redeem` functions

Summary: The `VaultLogic` library has a vulnerability in the `burn()` and `redeem()` functions where the balance verification does not account for fees, leading to potential underflows and accounting inconsistencies. While the functions deduct both the principal amount and fees from total supplies, they only verify availability of the principal amount.

Description: The `burn()` and `redeem()` functions in the `VaultLogic` library deduct both the output amount and fees from `totalSupplies`, but only verify that the output amount is available using the `_verifyBalance()` function. This creates a discrepancy between what is verified and what is actually deducted.

In the `burn()` function:

```
_verifyBalance($, params.asset, params.amountOut);
$.totalSupplies[params.asset] -= params.amountOut + params.fee;
```

Similarly, in the `redeem()` function:

```
_verifyBalance($, asset, params.amountsOut[i]);
$.totalSupplies[asset] -= params.amountsOut[i] + params.fees[i];
```

The `_verifyBalance()` function checks if there's enough available balance (total supplies minus total borrows) for a given amount, but it's not accounting for the additional fee that will be deducted.

Recommendation: Update the `burn()` and `redeem()` functions to verify the total amount being deducted, including fees:

```
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);
    }
    if (params.amountOut == 0) revert InvalidAmount();

-   _verifyBalance($, params.asset, params.amountOut);
+   _verifyBalance($, params.asset, params.amountOut + params.fee);

    $.totalSupplies[params.asset] -= params.amountOut + params.fee;

    IERC20(params.asset).safeTransfer(params.receiver, params.amountOut);
    IERC20(params.asset).safeTransfer($.insuranceFund, params.fee);

    emit Burn(msg.sender, params.receiver, params.asset, params.amountIn, params.amountOut, params.fee);
}
```

Similarly for the `redeem()` function:

```
for (uint256 i; i < length; ++i) {
    address asset = $.assets.at(i);
    if (params.amountsOut[i] < params.minAmountsOut[i]) {
        revert Slippage(asset, params.amountsOut[i], params.minAmountsOut[i]);
    }
    if (params.amountsOut[i] == 0) revert InvalidAmount();
-   _verifyBalance($, asset, params.amountsOut[i]);
+   _verifyBalance($, asset, params.amountsOut[i] + params.fees[i]);
    _updateIndex($, asset);
    $.totalSupplies[asset] -= params.amountsOut[i] + params.fees[i];
    IERC20(asset).safeTransfer(params.receiver, params.amountsOut[i]);
    IERC20(asset).safeTransfer($.insuranceFund, params.fees[i]);
}
```

CAP Labs: Fixed in PR 123 by implementing the suggested recommendation.

CAP Security Cartel: Verified fix.

7.3.6 Insufficient divestment in burn function excludes fees

Description: In the `burn()` function, users can withdraw an asset by burning their cap tokens. The function calculates both the withdrawal amount (`amountOut`) and a fee to be paid:

```
function burn(address _asset, uint256 _amountIn, uint256 _minAmountOut, address _receiver, uint256
↪   _deadline)
    external
    returns (uint256 amountOut)
{
    uint256 fee;
    (amountOut, fee) = getBurnAmount(_asset, _amountIn);
    divest(_asset, amountOut);  // <-- Only divests the amountOut, not including fee
    VaultLogic.burn(
        getVaultStorage(),
```

```
        MintBurnParams({
            asset: _asset,
            amountIn: _amountIn,
            amountOut: amountOut,
            minAmountOut: _minAmountOut,
            receiver: _receiver,
            deadline: _deadline,
            fee: fee
        })
    );
    _burn(msg.sender, _amountIn);
}
```

The issue is that the `divest(_asset, amountOut)` call only ensures there's enough liquidity for the base withdrawal amount, but not for the additional fee. As the `VaultLogic.burn()` implementation transfers both `amountOut` and `fee`, this could lead to insufficient funds being available when the fee is greater than the reserve amount.

Recommendation: Update the `burn()` function to divest the total amount needed for the withdrawal, including both the base amount and the fee:

```
function burn(address _asset, uint256 _amountIn, uint256 _minAmountOut, address _receiver, uint256
↪   _deadline)
    external
    returns (uint256 amountOut)
{
    uint256 fee;
    (amountOut, fee) = getBurnAmount(_asset, _amountIn);
-   divest(_asset, amountOut);
+   divest(_asset, amountOut + fee);
    VaultLogic.burn(
        getVaultStorage(),
        MintBurnParams({
            asset: _asset,
            amountIn: _amountIn,
            amountOut: amountOut,
            minAmountOut: _minAmountOut,
            receiver: _receiver,
            deadline: _deadline,
            fee: fee
        })
    );
    _burn(msg.sender, _amountIn);
}
```

CAP Labs: Fixed in PR 124.

CAP Security Cartel: Verified fix.

7.3.7 Incorrect ratio calculation causes excessive fees on first mint

Description: In the `MinterLogic._amountOutBeforeFee()` function, the first mint operation (when `capSupply == 0`) sets `newRatio = RAY_PRECISION` (representing 100%). This ratio is used in the fee calculation logic and can lead to excessive fees being charged to users during the first mint operation.

The issue occurs because:

1. When `capSupply == 0` (first mint), `newRatio` is set to `RAY_PRECISION` (100%)
2. This ratio is passed to `_applyFeeSlopes()` which calculates fees based on how far the ratio deviates from the optimal ratio
3. If `RAY_PRECISION > fees.optimalRatio`, this will trigger additional fees from the slope calculations
4. The first mint should logically not be penalized with excess fees as it's establishing the initial state

Recommendation: Modify the ratio calculation for the first mint case to prevent excessive fees:

```
if (params.mint) {
    assetValue = params.amount * assetPrice / assetDecimalsPow;
    if (capSupply == 0) {
-       newRatio = RAY_PRECISION;
+       newRatio = 0; // Set to 0 to avoid excess fees on first mint
        amount = params.amount * capDecimalsPow / assetDecimalsPow;
    }
    // ...
}
```

CAP Labs: Fixed in PR 125.

CAP Security Cartel: Verified fix. The new ratio is zero on the first mint.

7.3.8 Interest rate inaccuracy due to averaged utilization calculations

Description: The `VaultAdapter` contract calculates interest rates based on utilization ratios, but uses a simplified averaging approach that may lead to inaccurate interest rate determinations. Specifically, in the `rate()` function, utilization is calculated as:

```
if (elapsed != block.timestamp) {
    utilization = (index - utilizationData.index) / elapsed;
} else {
    utilization = IVault(_vault).utilization(_asset);
}
```

This approach averages utilization over the entire elapsed period. However, the `_applySlopes()` function applies different interest rate calculations depending on whether utilization is above or below a "kink" threshold:

```
if (_utilization > slopes.kink) {
    // Apply higher slope1 rate with increasing multiplier
} else {
    // Apply lower slope0 rate with decreasing multiplier
}
```

When utilization fluctuates around the kink point during the elapsed period, the average utilization might fall on either side of the kink while the actual utilization may have spent significant time on both sides. This leads to inaccurate interest rate calculations that don't properly reflect the true utilization pattern.

Additionally, the multiplier adjustments are cumulative and depend on how far utilization is from the kink, which compounds the inaccuracy when using averaged values.

Recommendation: Consider updating the interest rate calculation more frequently, ideally on each vault interaction that affects utilization (including deposits and withdrawals), to ensure that rates accurately reflect actual utilization patterns. This would prevent the need for averaging over long periods.

If frequent updates aren't practical, consider implementing a more sophisticated tracking mechanism that records significant utilization changes, especially those that cross the kink threshold, and adjusts rates accordingly rather than using a simple time-weighted average.

CAP Labs: I think at this time we will acknowledge the issue but won't fix. We will keep the recommendations in mind for any changes down the line.

In order to keep the vault and lender separate, we wouldn't want to notify on every utilization change. A more sophisticated interest rate system would take more time than we have to implement and test and get reviewed.

CAP Security Cartel: Acknowledged.

7.4 Informational

7.4.1 Redundant assignment of `lastUpdated` variable

Description: In `CapTokenAdapter.price()`, the `lastUpdated` variable is assigned the value of `block.timestamp` twice. This doesn't cause any functional issues but is an unnecessary operation that could be removed to improve code clarity and slightly reduce gas consumption.

```
lastUpdated = block.timestamp;  // First assignment

uint256 totalUsdValue;
lastUpdated = block.timestamp;  // Redundant second assignment
```

Recommendation: Remove the redundant assignment to improve code clarity and gas efficiency:

```
function price(address _asset) external view returns (uint256 latestAnswer, uint256 lastUpdated) {
    uint256 capTokenSupply = IERC20Metadata(_asset).totalSupply();
    if (capTokenSupply == 0) return (1e8, block.timestamp);

    address[] memory assets = IVault(_asset).assets();
    lastUpdated = block.timestamp;

    uint256 totalUsdValue;
-   lastUpdated = block.timestamp;

    for (uint256 i; i < assets.length; ++i) {
        // ...existing code...
    }

    // ...remaining code...
}
```

CAP Labs: Fixed in PR 137.

CAP Security Cartel: Verified fix.

7.4.2 Unsafe casting of potentially negative Chainlink price data

Description: The `ChainlinkAdapter.price()` function performs an unsafe type conversion from `int256` to `uint256` when processing price data from Chainlink oracles. While Chainlink price feeds are not expected to return negative values under normal circumstances, there is no validation to ensure the returned price is positive before the conversion.

```
function price(address _source) external view returns (uint256 latestAnswer, uint256 lastUpdated) {
    uint8 decimals = IChainlink(_source).decimals();
    int256 intLatestAnswer;
    (, intLatestAnswer,, lastUpdated,) = IChainlink(_source).latestRoundData();
    latestAnswer = uint256(intLatestAnswer);
    if (decimals < 8) latestAnswer *= 10 ** (8 - decimals);
    if (decimals > 8) latestAnswer /= 10 ** (decimals - 8);
}
```

If a Chainlink oracle were to return a negative value (potentially due to a malfunction, implementation error, or manipulation), the unchecked conversion to `uint256` would result in an extremely large positive value due to how two's complement encoding works. This could lead to incorrect price information being used in downstream calculations, potentially affecting protocol solvency or user funds.

Recommendation: Add a validation check to ensure the returned price is positive before performing the conversion.

```
function price(address _source) external view returns (uint256 latestAnswer, uint256 lastUpdated) {
    uint8 decimals = IChainlink(_source).decimals();
```

```
    int256 intLatestAnswer;
    (, intLatestAnswer,, lastUpdated,) = IChainlink(_source).latestRoundData();
+   require(intLatestAnswer > 0, "ChainlinkAdapter: Negative price");
    latestAnswer = uint256(intLatestAnswer);
    if (decimals < 8) latestAnswer *= 10 ** (8 - decimals);
    if (decimals > 8) latestAnswer /= 10 ** (decimals - 8);
}
```

CAP Labs: Fixed in PR 138.

CAP Security Cartel: Verified fix. A zero price is returned instead to use the backup oracle.

7.4.3 Missing vault existence check in `invest` function

Description: In the `FractionalReserveLogic.invest()` function, there is no explicit check to verify that a vault exists for an asset before attempting to deposit funds. Currently, if no vault is configured for an asset (i.e., `$.vault[_asset] == address(0)`), the function will continue execution until it fails at the `forceApprove()` call due to the `SafeERC20` library's protections against approving a zero address.

While the current implementation will revert safely, this approach relies on an implicit safeguard rather than an explicit validation. This makes the code less readable and could lead to confusion about the intended behavior.

```
function invest(IFractionalReserve.FractionalReserveStorage storage $, address _asset) external {
    uint256 assetBalance = IERC20(_asset).balanceOf(address(this));
    uint256 reserveBalance = $.reserve[_asset];

    if (assetBalance > reserveBalance) {
        uint256 investAmount = assetBalance - reserveBalance;
        $.loaned[_asset] += investAmount;
        IERC20(_asset).forceApprove($.vault[_asset], investAmount);
        IERC4626($.vault[_asset]).deposit(investAmount, address(this));
    }
}
```

It's worth noting that the `divest()` functions include explicit checks for vault existence (`if ($.vault[_asset] != address(0)) {`), showing an inconsistency in validation patterns across the library.

Recommendation: Add an explicit check to verify that a vault exists for the asset before proceeding with the investment:

```
function invest(IFractionalReserve.FractionalReserveStorage storage $, address _asset) external {
    uint256 assetBalance = IERC20(_asset).balanceOf(address(this));
    uint256 reserveBalance = $.reserve[_asset];

-   if (assetBalance > reserveBalance) {
+   if (assetBalance > reserveBalance && $.vault[_asset] != address(0)) {
        uint256 investAmount = assetBalance - reserveBalance;
        $.loaned[_asset] += investAmount;
        IERC20(_asset).forceApprove($.vault[_asset], investAmount);
        IERC4626($.vault[_asset]).deposit(investAmount, address(this));
    }
}
```

CAP Labs: Fixed in PR 139 by implementing the suggested recommendation.

CAP Security Cartel: Verified fix.

7.4.4 Borrows can be griefed when token reserves are low

Description: An agent can monitor pending borrow transactions and execute their own borrows through front-running to reduce the token reserves. This will reduce the borrowing capacity returned from `maxBorrowable()` in `validateBorrow()`, causing the victim's transaction to revert with `CollateralCannotCoverNewBorrow()`.

```
function validateBorrow(ILender.LenderStorage storage $, ILender.BorrowParams memory params) external {
    // ...

    uint256 borrowCapacity = ViewLogic.maxBorrowable($, params.agent, params.asset); // @audit: capped
↪   to the token reserves

    if (params.amount > borrowCapacity) revert CollateralCannotCoverNewBorrow(); // @audit: reverts if
↪   not enough borrow capacitiy

    // ...
}
```

Recommendation: Implement a "magic value" option that allows users to borrow the maximum borrow capacity.

```
function validateBorrow(ILender.LenderStorage storage $, ILender.BorrowParams memory params) external {
-   if (params.amount < $.reservesData[params.asset].minBorrow) revert MinBorrowAmount();
    if (params.receiver == address(0) || params.asset == address(0)) revert ZeroAddressNotValid();
    if ($.reservesData[params.asset].paused) revert ReservePaused();
    uint256 borrowCapacity = ViewLogic.maxBorrowable($, params.agent, params.asset);
-   if (params.amount > borrowCapacity) revert CollateralCannotCoverNewBorrow();
+
+   // Special case: If amount equals MAX_UINT, set to max borrowable
+   if (params.amount == type(uint256).max) {
+       params.amount = borrowCapacity;
+   } else {
+       if (params.amount < $.reservesData[params.asset].minBorrow) revert MinBorrowAmount();
+       if (params.amount > borrowCapacity) revert CollateralCannotCoverNewBorrow();
+   }
+
    IDelegation($.delegation).setLastBorrow(params.agent);
}
```

This allows users to specify `type(uint256).max` to borrow their maximum capacity, automatically adjusting to the actual borrowing capacity and preventing front-running attacks.

CAP Labs: Fixed in PR 140.

CAP Security Cartel: Verified fix. A boolean has been added to borrow the maximum amount, it is set to `true` when the amount is `uint256.max`.

7.4.5 Emergency liquidation bonus depends on liquidation start timestamp

Description: The `getBonus()` function calculates the bonus amount based on the liquidation start timestamp. When an emergency liquidation occurs without a prior call to `initiateLiquidation()`, the `$.liquidationStart[agent]` value is zero, leading to a maximum bonus calculation.

However, the same liquidation with a prior call to `initiateLiquidation()` at the current timestamp results in a zero bonus amount.

```
function getBonus(
    // ...
) internal view returns (uint256 bonus) {
    if (totalDelegation > totalDebt) {
        uint256 elapsed;
```

```
        // This would only happen if liquidation is called when its emergencyLiquidationThreshold is
↪   below 1e27 and before the grace period ends
        if (block.timestamp < $.liquidationStart[agent] + $.grace) {
            elapsed = block.timestamp - $.liquidationStart[agent]; // @audit elapsed = 0 when a prior
↪   call to `initiateLiquidation` is made in the same block
        } else {
            elapsed = block.timestamp - ($.liquidationStart[agent] + $.grace);
        }

        uint256 duration = $.expiry - $.grace;
        if (elapsed > duration) elapsed = duration;  // @audit elapsed is capped to duration when no
↪   prior call to `initiateLiquidation`

        uint256 bonusPercentage = $.bonusCap * elapsed / duration;
        // ...
    }
}
```

This inconsistency leads to not incentivizing emergency liquidations when a prior call to `initiateLiquidation()` has been made.

Recommendation: The maximum bonus could be applied to any emergency liquidation for incentivizing liquidating highly unhealthy positions.

CAP Labs: Fixed in PR 141.

CAP Security Cartel: Verified fix. The full bonus is given for emergency liquidation.

7.4.6 Inconsistent reserve pausing implementation

Description: The `ValidationLogic` library implements a pausing mechanism for reserves via the `validateBorrow()` function which checks if the reserve is paused before allowing a borrow:

```
function validateBorrow(ILender.LenderStorage storage $, ILender.BorrowParams memory params) external {
    // ...
    if ($.reservesData[params.asset].paused) revert ReservePaused();
    // ...
}
```

However, this validation is not consistently applied across all functions that allow borrowing. Two specific functions bypass this validation:

1. `realizeInterest()`
2. `realizeRestakerInterest()`

These functions can execute borrows directly without checking the paused status of the reserve, which creates an inconsistent security boundary. If a reserve is paused due to security concerns or market conditions, these functions could still issue new borrows, potentially exposing the protocol to risk.

Recommendation: Add the pause validation check to all functions that can issue borrows. For both `realizeInterest()` and `realizeRestakerInterest()`, add a check similar to:

```
function realizeInterest(...) {
    // ...
+   if ($.reservesData[asset].paused) revert ReservePaused();
    // continue with borrow logic
    // ...
}

function realizeRestakerInterest(...) {
    // ...
+   if ($.reservesData[asset].paused) revert ReservePaused();
```

```
    // continue with borrow logic
    // ...
}
```

Note: This fix will also make `repay` unusable when the contract is paused as `realizeRestakerInterest` would revert.

CAP Labs: Fixed in PR 142.

CAP Security Cartel: Verified fix. Instead of reverting, `maxRealization()` and `maxRestakerRealization()` have been modified to return zero realized interest (converting this to unrealized for restaker interest), allowing agents to make repayments even when the reserve has been paused.

7.4.7 Missing parameter validation

Description: The `Lender` contract initialization process lacks proper validation for parameters. Specifically, the `targetHealth` and `bonusCap` parameters do not have boundary checks to ensure they are set within reasonable limits.

Key instances where validation is missing:

1. The `targetHealth` parameter should have a minimum threshold to prevent unsafe health targets:

```
$.targetHealth = _targetHealth;
```

2. The `bonusCap` parameter lacks an upper bound check to prevent excessive liquidation bonuses:

```
$.bonusCap = _bonusCap;
```

Without these validations, the contract could be initialized with unsafe values, potentially leading to economic vulnerabilities or unexpected system behavior.

Recommendation: Add appropriate parameter validation during initialization to ensure all parameters are within safe operating ranges:

```
// For targetHealth
+ if (_targetHealth < 1e27) revert InvalidTargetHealth();
$.targetHealth = _targetHealth;

// For bonusCap
+ if (_bonusCap > 1e27) revert InvalidBonusCap();
$.bonusCap = _bonusCap;
```

CAP Labs: Fixed in PR 126.

CAP Security Cartel: Verified fix.

7.5 Gas Optimization

7.5.1 Redundant contains check in agent addition

Description: In the `addAgent()` function, there's an unnecessary gas cost due to redundant checks using `EnumerableSet.contains()`. The current implementation performs a `contains()` check to verify if an agent already exists, and then performs an `add()` operation. However, the `add()` function in EnumerableSet also internally performs a `contains()` check and returns a boolean indicating whether the element was actually added.

The current implementation:

```
if ($.agents.contains(_agent)) revert DuplicateAgent();
if (!$.networks.contains(_network)) revert NetworkDoesntExist();
$.agents.add(_agent);
```

This causes duplicate `contains()` checks to be performed for the agent address, resulting in unnecessary gas costs.

Recommendation: Modify the code to use the return value from `add()` to determine if the agent already exists, eliminating the redundant `contains()` check:

```
- if ($.agents.contains(_agent)) revert DuplicateAgent();
  if (!$.networks.contains(_network)) revert NetworkDoesntExist();
- $.agents.add(_agent);
+ if (!$.agents.add(_agent)) revert DuplicateAgent();
```

CAP Labs: Fixed in PR 136.

CAP Security Cartel: Verified fix.

7.5.2 Avoid redundant contains check before add in `EnumerableSet`

Description: In the `registerNetwork()` function, there is a redundant check for duplicate networks that can be optimized to save gas. The function currently makes two internal calls to `EnumerableSet.contains()` - one directly in the code and another implicitly inside the `EnumerableSet.add()` method.

```
function registerNetwork(address _network) external checkAccess(this.registerNetwork.selector) {
    DelegationStorage storage $ = getDelegationStorage();
    if (_network == address(0)) revert InvalidNetwork();

    // Check for duplicates
    if ($.networks.contains(_network)) revert DuplicateNetwork();

    $.networks.add(_network);
    emit RegisterNetwork(_network);
}
```

The `EnumerableSet.add()` function already checks for existence internally and returns a boolean indicating whether the element was added (true) or already existed (false).

Recommendation: Modify the `registerNetwork()` function to use the return value from `EnumerableSet.add()` instead of making a separate `contains()` check:

```
function registerNetwork(address _network) external checkAccess(this.registerNetwork.selector) {
    DelegationStorage storage $ = getDelegationStorage();
    if (_network == address(0)) revert InvalidNetwork();

-   // Check for duplicates
-   if ($.networks.contains(_network)) revert DuplicateNetwork();
-
-   $.networks.add(_network);
+   // Add returns false if the element already exists
```

```
+   if (!$.networks.add(_network)) revert DuplicateNetwork();

    emit RegisterNetwork(_network);
}
```

CAP Labs: Fixed in PR 136.

CAP Security Cartel: Verified fix.