SHERLOCK

Security Review For
Cap

Public Bug Bounty Audit Contest Prepared For:	**Cap**
Lead Security Expert:	**0x73696d616f**
Date Audited:	**July 10 - July 24, 2025**
Final Commit:	**403326d**

Introduction

Cap is a stablecoin protocol on Ethereum with two products: cUSD and stcUSD. cUSD is a digital dollar, backed by blue chip dollar assets like BENJI and USDC. stcUSD is a savings product that generates yield via an autonomous layer of operators.

Scope

Repository: cap-labs-dev/cap-contracts

Audited Commit: 0a57fbfdba7f54e516b5ed412548b7e415f3739d

Final Commit: 403326df39f6987359bba648b7c09f81bd53da62

Files:

- contracts/access/AccessControl.sol
- contracts/access/Access.sol
- contracts/delegation/Delegation.sol
- contracts/delegation/providers/symbiotic/SymbioticNetworkMiddleware.sol
- contracts/delegation/providers/symbiotic/SymbioticNetwork.sol
- contracts/feeAuction/FeeAuction.sol
- contracts/lendingPool/Lender.sol
- contracts/lendingPool/libraries/BorrowLogic.sol
- contracts/lendingPool/libraries/configuration/AgentConfiguration.sol
- contracts/lendingPool/libraries/LiquidationLogic.sol
- contracts/lendingPool/libraries/math/MathUtils.sol
- contracts/lendingPool/libraries/math/PercentageMath.sol
- contracts/lendingPool/libraries/math/WadRayMath.sol
- contracts/lendingPool/libraries/ReserveLogic.sol
- contracts/lendingPool/libraries/ValidationLogic.sol
- contracts/lendingPool/libraries/ViewLogic.sol
- contracts/lendingPool/tokens/base/MintableERC20.sol
- contracts/lendingPool/tokens/base/ScaledToken.sol
- contracts/lendingPool/tokens/DebtToken.sol
- contracts/oracle/libraries/AaveAdapter.sol
- contracts/oracle/libraries/CapTokenAdapter.sol
- contracts/oracle/libraries/ChainlinkAdapter.sol

- contracts/oracle/libraries/VaultAdapter.sol
- contracts/oracle/Oracle.sol
- contracts/oracle/PriceOracle.sol
- contracts/oracle/RateOracle.sol
- contracts/storage/AccessStorageUtils.sol
- contracts/storage/DebtTokenStorageUtils.sol
- contracts/storage/DelegationStorageUtils.sol
- contracts/storage/FeeAuctionStorageUtils.sol
- contracts/storage/FractionalReserveStorageUtils.sol
- contracts/storage/LenderStorageUtils.sol
- contracts/storage/MintableERC20StorageUtils.sol
- contracts/storage/MinterStorageUtils.sol
- contracts/storage/PriceOracleStorageUtils.sol
- contracts/storage/RateOracleStorageUtils.sol
- contracts/storage/ScaledTokenStorageUtils.sol
- contracts/storage/StakedCapStorageUtils.sol
- contracts/storage/SymbioticNetworkMiddlewareStorageUtils.sol
- contracts/storage/SymbioticNetworkStorageUtils.sol
- contracts/storage/VaultAdapterStorageUtils.sol
- contracts/storage/VaultStorageUtils.sol
- contracts/token/CapToken.sol
- contracts/token/StakedCap.sol
- contracts/vault/FractionalReserve.sol
- contracts/vault/libraries/FractionalReserveLogic.sol
- contracts/vault/libraries/MinterLogic.sol
- contracts/vault/libraries/VaultLogic.sol
- contracts/vault/Minter.sol
- contracts/vault/Vault.sol

Final Commit Hash

403326df39f6987359bba648b7c09f81bd53da62

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues Found

High	Medium
0	10

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

0x73696d616f
0xsh
0xzey
Albert_Mei
Bigsam
Drynooo
HeckerTrieuTien
Knight1
KrisRenZo
Matin
Tigerfrake
Uddercover
anirruth_
attacker_code
dobrevaleri
farismaulana
frndz0ne
kangaroo
lanrebayode77
magiccentaur
maxim371
montecristo
moray5554
pashap9990
roshark
silver_eth
swarun
ustas
valuevalk

Issue M-1: Attacker/partial liquidator can extend Liquidation action by resetting $.liquidationStart[_-agent] to 0.

Source: https://github.com/sherlock-audit/2025-07-cap-judging/issues/145

Found by

Bigsam, dobrevaleri, roshark, silver_eth, swarun

Summary

Attackers can extend Liquidation by Action, by partially liquidating a position with a few wei to manipulate health to >=1e27 before taking the collateral, since the account is still liquidatable after the delegator is slashed of this amount, a new liquidator's call to liquidate this position will revert incorrectly. This is because the health check to close the liquidation uses debt amount after repayment with delegation still yet unslashed.

Root Cause

```
    /// @notice Liquidate an agent when their health is below 1
    /// @dev Liquidation must be opened first and the grace period must have
    ↪  passed. Liquidation
    /// bonus linearly increases, once grace period has ended, up to the cap at
    ↪  expiry.
    /// All health factors, LTV ratios, and thresholds are in ray (1e27)
    /// @param $ Lender storage
    /// @param params Parameters to liquidate an agent
    /// @return liquidatedValue Value of the liquidation returned to the liquidator
    function liquidate(ILender.LenderStorage storage $, ILender.RepayParams memory
    ↪  params)
        external
        returns (uint256 liquidatedValue)
    {
        (uint256 totalDelegation, uint256 totalSlashableCollateral, uint256
        ↪  totalDebt,,, uint256 health) =
            ViewLogic.agent($, params.agent);

        ValidationLogic.validateLiquidation(
            health,
            totalDelegation * $.emergencyLiquidationThreshold / totalDebt,
            $.liquidationStart[params.agent],
            $.grace,
```

```
            $.expiry
        );

        (uint256 assetPrice,) = IOracle($.oracle).getPrice(params.asset);
        uint256 bonus = ViewLogic.bonus($, params.agent);
        uint256 maxLiquidation = ViewLogic.maxLiquidatable($, params.agent,
        ↪  params.asset);
        uint256 liquidated = params.amount > maxLiquidation ? maxLiquidation :
        ↪  params.amount;

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

@audit>>          (,,,,, health) = ViewLogic.agent($, params.agent);
@audit>>           if (health >= 1e27) _closeLiquidation($, params.agent);   //
↪  premature close ............. health is not health.....

        liquidatedValue =
            (liquidated + (liquidated * bonus / 1e27)) * assetPrice / (10 **
            ↪  $.reservesData[params.asset].decimals);
        if (totalSlashableCollateral < liquidatedValue) liquidatedValue =
        ↪  totalSlashableCollateral;

        if (liquidatedValue > 0) IDelegation($.delegation).slash(params.agent,
        ↪  params.caller, liquidatedValue);

        emit Liquidate(params.agent, params.caller, params.asset, liquidated,
        ↪  liquidatedValue);
    }
```

Liquidation is closed before delegations are slashed, hence the health factor returned is not the ending health factor.....

```
    /// @dev Cancel further liquidations with no checks
    /// @param $ Lender storage
    /// @param _agent Agent address
    function _closeLiquidation(ILender.LenderStorage storage $, address _agent)
    ↪  internal {

@audit>>        $.liquidationStart[_agent] = 0;

         emit CloseLiquidation(_agent);
    }
```

This will cause the the next liquidator or actual liquidator that was front run , the liquidator's call will revert here.

```
    function liquidate(ILender.LenderStorage storage $, ILender.RepayParams memory
    ↪  params)
        external
        returns (uint256 liquidatedValue)
    {
        (uint256 totalDelegation, uint256 totalSlashableCollateral, uint256
        ↪  totalDebt,,, uint256 health) =
            ViewLogic.agent($, params.agent);

@audit>>          ValidationLogic.validateLiquidation(
            health,
            totalDelegation * $.emergencyLiquidationThreshold / totalDebt,
            $.liquidationStart[params.agent],
            $.grace,
            $.expiry
        );
```

```
    /// @notice Validate the liquidation of an agent
    /// @dev Health of above 1e27 is healthy, below is liquidatable
    /// @param health Health of an agent's position
    /// @param emergencyHealth Emergency health below which the grace period is
    ↪  voided
    /// @param start Last liquidation start time
    /// @param grace Grace period duration
    /// @param expiry Liquidation duration after which it expires
    function validateLiquidation(uint256 health, uint256 emergencyHealth, uint256
    ↪  start, uint256 grace, uint256 expiry)
        external
        view
    {
        if (health >= 1e27) revert HealthFactorNotBelowThreshold();
        if (emergencyHealth >= 1e27) {

@audit>>              if (block.timestamp <= start + grace) revert
↪  GracePeriodNotOver();
            if (block.timestamp >= start + expiry) revert LiquidationExpired();
        }
    }
```

When we repay we burn debt tokens this will reduce the debt of the agent, but this reduced debt is used to get the new health before delegation is reduced.

Because the health factor will return a value above >= 1e27 which is incorrect because delegation has not been reduced by the liquidated value. hence the check is using a higher than the actual health factor.

This means the liquidator will be force to Open another liquidation with a new Grace

Period.

The attacker is incentivized to carry out this attack has they will receive the liquidated value plus fee, making this attack profitable.

Internal Pre-conditions

1. Account health drops below 1e27
2. Liquidation is open giving the liquidator a grace period to respond.

External Pre-conditions

https://github.com/sherlock-audit/2025-07-cap/blob/main/cap-contracts/contracts/lendingPool/libraries/LiquidationLogic.sol#L81-L87

https://github.com/sherlock-audit/2025-07-cap/blob/main/cap-contracts/contracts/lendingPool/libraries/ViewLogic.sol#L108-L129

https://github.com/sherlock-audit/2025-07-cap/blob/main/cap-contracts/contracts/lendingPool/libraries/BorrowLogic.sol#L158

Attack Path

1. Account health drops below 1e27
2. Liquidation is open giving the liquidator a grace period to respond.
3. Attacker liquidates immediately grace period passes with a partial liquidation
4. Account is still liquidatable because the health factor wasn't returned back to health
5. Attacker successfully updates the mapping with the health still below 1e27.

Impact

Health factor below 1e27 will have the Open liquidation call close prematurely, preventing other liquidators from liquidating early.

PoC

No response

Mitigation

Close the liquidation after the delegation Amount has been slashed to ensure that the health factor is indeed the Positions current health factor.

```
  ValidationLogic.validateLiquidation(
            health,
            totalDelegation * $.emergencyLiquidationThreshold / totalDebt,
            $.liquidationStart[params.agent],
            $.grace,
            $.expiry
        );

        (uint256 assetPrice,) = IOracle($.oracle).getPrice(params.asset);
        uint256 bonus = ViewLogic.bonus($, params.agent);
        uint256 maxLiquidation = ViewLogic.maxLiquidatable($, params.agent,
        ↪  params.asset);
        uint256 liquidated = params.amount > maxLiquidation ? maxLiquidation :
        ↪  params.amount;

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

--        (,,,,, health) = ViewLogic.agent($, params.agent);
--        if (health >= 1e27) _closeLiquidation($, params.agent);   // premature
↪  close ............. health is not health.....

        liquidatedValue =
            (liquidated + (liquidated * bonus / 1e27)) * assetPrice / (10 **
            ↪  $.reservesData[params.asset].decimals);
        if (totalSlashableCollateral < liquidatedValue) liquidatedValue =
        ↪  totalSlashableCollateral;

        if (liquidatedValue > 0) IDelegation($.delegation).slash(params.agent,
        ↪  params.caller, liquidatedValue);


++        (,,,,, health) = ViewLogic.agent($, params.agent);
++       if (health >= 1e27) _closeLiquidation($, params.agent);   // premature
↪  close ............. health is not health.....

        emit Liquidate(params.agent, params.caller, params.asset, liquidated,
        ↪  liquidatedValue);
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/cap-labs-dev/cap-contracts/pull/185

Issue M-2: Utilization rate multiplier will not shift if oracle is consulted frequently

Source: https://github.com/sherlock-audit/2025-07-cap-judging/issues/148

Found by

Matin, kangaroo, montecristo, valuevalk

Summary

Due to round down issue in VaultAdapter's multiplier shift calculation, utilization rate will not increase/decrease if oracle is consulted frequently.

Root Cause

Root cause resides in multiplier calculation in `VaultAdapter.sol`:

File: cap-contracts/contracts/oracle/libraries/VaultAdapter.sol

```
86: if (_utilization > slopes.kink) {
87:             uint256 excess = _utilization - slopes.kink;
88:@>           utilizationData.multiplier = utilizationData.multiplier
89:                 * (1e27 + (1e27 * excess / (1e27 - slopes.kink)) * (_elapsed *
 ↪  $.rate / 1e27)) / 1e27;
90:
91:             if (utilizationData.multiplier > $.maxMultiplier) {
92:                 utilizationData.multiplier = $.maxMultiplier;
93:             }
94:
95:             interestRate = (slopes.slope0 + (slopes.slope1 * excess / 1e27)) *
 ↪  utilizationData.multiplier / 1e27;
96:         } else {
97:@>           utilizationData.multiplier = utilizationData.multiplier * 1e27
98:                 / (1e27 + (1e27 * (slopes.kink - _utilization) / slopes.kink) *
 ↪  (_elapsed * $.rate / 1e27));
99:
100:             if (utilizationData.multiplier < $.minMultiplier) {
101:                 utilizationData.multiplier = $.minMultiplier;
102:             }
103:
104:             interestRate = (slopes.slope0 * _utilization / slopes.kink) *
 ↪  utilizationData.multiplier / 1e27;
105:         }
```

More specifically, rounding issue happens in the following calculation:

`_elapsed` is diff between `block.timestamp` and `utilizationData.lastUpdate`. This means one can consult rate oracle frequently to make this term 0.

As a result, `utilizationData.multiplier` will not shift in L88 and L97.

Internal Pre-conditions

n/a

External Pre-conditions

n/a

Attack Path

Attacker calls `Oracle::utilizationRate` function frequently.

The interval depends on rate parameter of `VaultAdapter` contract.

For example, if the rate is set so that multiplier is shifted fully in 24 hours window, the attacker can consult the oracle every 23 hours to prevent multiplier shifts.

Impact

DebtToken's interest rate calculation will be affected. The attacker can prevent it from increasing during high LTV, or vice versa during low LTV.

PoC

```
pragma solidity ^0.8.28;

import { IOracle } from "../contracts/interfaces/IOracle.sol";
import { IVaultAdapter } from "../contracts/interfaces/IVaultAdapter.sol";
import { VaultAdapter } from "../contracts/oracle/libraries/VaultAdapter.sol";
import { TestDeployer } from "./deploy/TestDeployer.sol";
import { console } from "forge-std/console.sol";

contract POC is TestDeployer {
    VaultAdapter adapter;
    address user_agent;

    function setUp() public {
        _deployCapTestEnvironment();

        _initTestVaultLiquidity(usdVault);
        _initSymbioticVaultsLiquidity(env);
```

```
        user_agent = _getRandomAgent();

        vm.startPrank(env.symbiotic.users.vault_admin);
        _symbioticVaultDelegateToAgent(symbioticWethVault,
        ↪  env.symbiotic.networkAdapter, user_agent, 2.385e18);
        vm.stopPrank();

        vm.startPrank(env.users.lender_admin);

        uint256 reservesCount = lender.reservesCount();
        console.log("Reserves Count", reservesCount);

        vm.stopPrank();

        // setup vault adapter
        vm.startPrank(env.users.access_control_admin);
        VaultAdapter adapterImpl = new VaultAdapter();
        adapter = VaultAdapter(_proxy(address(adapterImpl)));
        adapter.initialize(address(accessControl));
        accessControl.grantAccess(adapter.setSlopes.selector, address(adapter),
        ↪  env.users.access_control_admin);
        accessControl.grantAccess(adapter.setLimits.selector, address(adapter),
        ↪  env.users.access_control_admin);
        adapter.setSlopes(address(usdc), IVaultAdapter.SlopeData({ kink: 2e26,
        ↪  slope0: 0.01e27, slope1: 0.03e27 }));
        adapter.setLimits({
            _maxMultiplier: 1e27, // 100%
            _minMultiplier: 1e25, // 1%
            _rate: uint(1e27) / 1 days // 100% / day
        });
        vm.stopPrank();
    }

    function test_submissionValidity() public {
        vm.startPrank(user_agent);
        // utilization rate is set below than the kink ratio
        lender.borrow(address(usdc), 100e6, user_agent);
        assertLt(cUSD.utilization(address(usdc)), 2e26);
        adapter.rate(env.usdVault.capToken, address(usdc));
        // utilization rate is set above than the kink ratio
        lender.borrow(address(usdc), 2600e6, user_agent);
        assertGt(cUSD.utilization(address(usdc)), 2e26);

        // due to rounding issue, utilization rate will not grow because multiplier
        ↪  is not shifted
        _timeTravel(1 days / 2);
        uint256 utilizationRateFirst = adapter.rate(env.usdVault.capToken,
        ↪  address(usdc));
        _timeTravel(1 days / 2);
```

```
        uint256 utilizationRateSecond = adapter.rate(env.usdVault.capToken,
        ↪  address(usdc));
        assertEq(utilizationRateFirst, utilizationRateSecond);

        // rounding issue happens for 1 day as well, so we need to advance 1 second
        ↪  further
        _timeTravel(1 days + 1);
        // now the utilization rate is increased
        uint256 utilizationRateThird = adapter.rate(env.usdVault.capToken,
        ↪  address(usdc));
        assertGt(utilizationRateThird, utilizationRateSecond);
    }
}
```

Mitigation

```
diff --git a/cap-contracts/contracts/oracle/libraries/VaultAdapter.sol
↪  b/cap-contracts/contracts/oracle/libraries/VaultAdapter.sol
index 0fff0f0..7bdf86b 100644
--- a/cap-contracts/contracts/oracle/libraries/VaultAdapter.sol
+++ b/cap-contracts/contracts/oracle/libraries/VaultAdapter.sol
@@ -86,7 +86,7 @@ contract VaultAdapter is IVaultAdapter, UUPSUpgradeable, Access,
↪  VaultAdapterSto
         if (_utilization > slopes.kink) {
             uint256 excess = _utilization - slopes.kink;
             utilizationData.multiplier = utilizationData.multiplier
-                * (1e27 + (1e27 * excess / (1e27 - slopes.kink)) * (_elapsed *
↪  $.rate / 1e27)) / 1e27;
+                * (1e27 + (1e27 * excess / (1e27 - slopes.kink)) * _elapsed *
↪  $.rate / 1e27) / 1e27;

             if (utilizationData.multiplier > $.maxMultiplier) {
                 utilizationData.multiplier = $.maxMultiplier;
@@ -95,7 +95,7 @@ contract VaultAdapter is IVaultAdapter, UUPSUpgradeable, Access,
↪  VaultAdapterSto
             interestRate = (slopes.slope0 + (slopes.slope1 * excess / 1e27)) *
             ↪  utilizationData.multiplier / 1e27;
         } else {
             utilizationData.multiplier = utilizationData.multiplier * 1e27
-                / (1e27 + (1e27 * (slopes.kink - _utilization) / slopes.kink) *
↪  (_elapsed * $.rate / 1e27));
+                / (1e27 + (1e27 * (slopes.kink - _utilization) / slopes.kink) *
↪  _elapsed * $.rate / 1e27);

             if (utilizationData.multiplier < $.minMultiplier) {
                 utilizationData.multiplier = $.minMultiplier;
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/cap-labs-dev/cap-contracts/pull/187

Issue M-3: Lender DoS if all asset is borrowed or realized

Source: https://github.com/sherlock-audit/2025-07-cap-judging/issues/150

Found by

Bigsam, Drynooo, farismaulana, frndz0ne, lanrebayode77, magiccentaur, montecristo

Summary

If all asset is borrowed (or already realized to restaker interests), `realizedInterest` is 0. Thus, there is no rewards to distribute to restakers. However, `BorrowLogic` is missing zero check, thus it will try to distribute rewards anway, which will revert due to `InsufficientReward` error from Symbiotic `DefaultStakerRewards` contract.

Root Cause

When debt is repaid, restakers' interests will be realized.

File: cap-contracts/contracts/lendingPool/libraries/BorrowLogic.sol

```
201: function realizeRestakerInterest(ILender.LenderStorage storage $, address
↪  _agent, address _asset)
202:         public
203:         returns (uint256 realizedInterest)
204:     {
205:         ILender.ReserveData storage reserve = $.reservesData[_asset];
206:         uint256 unrealizedInterest;
207:         (realizedInterest, unrealizedInterest) = maxRestakerRealization($,
↪  _agent, _asset);
208:         reserve.lastRealizationTime[_agent] = block.timestamp;
209:
210:         if (realizedInterest == 0 && unrealizedInterest == 0) return 0;
211:
212:         reserve.debt += realizedInterest;
213:         reserve.unrealizedInterest[_agent] += unrealizedInterest;
214:         reserve.totalUnrealizedInterest += unrealizedInterest;
215:
216:         IDebtToken(reserve.debtToken).mint(_agent, realizedInterest +
↪  unrealizedInterest);
217:         IVault(reserve.vault).borrow(_asset, realizedInterest, $.delegation);
218:@>       IDelegation($.delegation).distributeRewards(_agent, _asset);
219:         emit RealizeInterest(_asset, realizedInterest, $.delegation);
```

In L207, `realizedInterest` equals to 0 if all asset is borrowed.

So in L217, lender contract will borrow 0 asset from cUSD contract.

In 218, delegation distributes rewards but it doesn't hold any reward. Thus, transaction will revert with `InsufficientReward` error due to this check.

Internal Pre-conditions

1. All asset is borrowed

External Pre-conditions

n/a

Attack Path

n/a

Impact

Lender contract will suffer from DoS, as restaker rewards are realized prior to all major features (borrow, repay, liquidation etc).

Although DoS can be easily mitigated by minting 1 wei of cUSD (thus, depositing affected asset), this vulnerability can be exploited to brick certain time-sensitive operations.

For example:

- One agent can brick other agent from repaying, or brick liquidations (which is very time-sensitive).
- If vault does not have enough balance to cover the current restaker reward, an attacker can frontrun incoming operations (borrow, repay, liquidation) by `Lender::realizeRestakerInterest` to drain vault balance and brick the operation.

Additional Note

This bug also affects repayment on paused asset. As `realizedInterest` is set to 0 for paused asset:

File: cap-contracts/contracts/lendingPool/libraries/BorrowLogic.sol

```
262: if (reserve.paused) {
263:             unrealizedInterest = realization;
264:@>             realization = 0;
```

But paused asset repayment is blocked by another bug which I'll describe in another report.

PoC

```
pragma solidity ^0.8.28;

import { IOracle } from "../contracts/interfaces/IOracle.sol";
import { IVaultAdapter } from "../contracts/interfaces/IVaultAdapter.sol";
import { VaultAdapter } from "../contracts/oracle/libraries/VaultAdapter.sol";
import { TestDeployer } from "./deploy/TestDeployer.sol";
import { console } from "forge-std/console.sol";

contract POC is TestDeployer {
    VaultAdapter adapter;
    address user_agent;

    function setUp() public {
        _deployCapTestEnvironment();

        _initTestVaultLiquidity(usdVault);
        _initSymbioticVaultsLiquidity(env);

        user_agent = _getRandomAgent();

        vm.startPrank(env.symbiotic.users.vault_admin);
        _symbioticVaultDelegateToAgent(symbioticWethVault,
        ↪  env.symbiotic.networkAdapter, user_agent, 100e18);
        vm.stopPrank();
    }

    function test_submissionValidity() public {
        _setAssetOraclePrice(address(weth), 2000e8);
        vm.startPrank(user_agent);
        // borrows all USDC
        lender.borrow(address(usdc), 12000e6, user_agent);

        _timeTravel(90 days);
        usdc.approve(address(lender), 5000e6);
        // repay is DoSed
        vm.expectRevert(abi.encodeWithSignature("InsufficientReward()"));
        lender.repay(address(usdc), 5000e6, user_agent);
        vm.stopPrank();
    }
}
```

Mitigation

```
diff --git a/cap-contracts/contracts/lendingPool/libraries/BorrowLogic.sol
 ↪  b/cap-contracts/contracts/lendingPool/libraries/BorrowLogic.sol
index 3c2b60a..e36f043 100644
--- a/cap-contracts/contracts/lendingPool/libraries/BorrowLogic.sol
+++ b/cap-contracts/contracts/lendingPool/libraries/BorrowLogic.sol
@@ -214,8 +214,10 @@ library BorrowLogic {
         reserve.totalUnrealizedInterest += unrealizedInterest;

         IDebtToken(reserve.debtToken).mint(_agent, realizedInterest +
          ↪  unrealizedInterest);
-        IVault(reserve.vault).borrow(_asset, realizedInterest, $.delegation);
-        IDelegation($.delegation).distributeRewards(_agent, _asset);
+        if (realizedInterest > 0) {
+            IVault(reserve.vault).borrow(_asset, realizedInterest, $.delegation);
+            IDelegation($.delegation).distributeRewards(_agent, _asset);
+        }
         emit RealizeInterest(_asset, realizedInterest, $.delegation);
     }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/cap-labs-dev/cap-contracts/pull/189

Issue M-4: Cannot repay or liquidate on paused asset

Source: https://github.com/sherlock-audit/2025-07-cap-judging/issues/180

Found by

0x73696d616f, Bigsam, Knight1, Tigerfrake, dobrevaleri, farismaulana, montecristo, pashap9990

Summary

Although the protocol allows repayment on paused asset, it cannot be done because Lender contract tries an unnecessary (and prohibited due to pause) borrow from Vault contract.

Root Cause

First, let's see why repayment on paused asset is an intended protocol design.

- In Lender contract: Asset's paused state is checked on borrow, but **not** checked on liquidation
- In Vault contract: `whenNotPaused` modifier is applied to mint and borrow functions, but **not** applied to burn and repay functions
- In `BorrowLogic` library, realized interests are specially handled if the asset is paused.

So for the paused assets, the protocol wants to:

- Disallow minting or borrowing
- Allow repaying or liquidating
- Avoid/delay collecting interest

However, repayment on paused assets (and consequently liquidations) will not be processed because Lender contract unncessarily tries to borrow from Vault contract:

- Prior to repayment, lender contract realizes restaker interest
- Since the asset is paused, realizedInterest = 0
- Since there is no realization, lender contract doesn't need to borrow from Vault contract to repay the interest. However, it tries to borrow 0 from Vault contract:
- Because the asset is paused, Vault contract will revert with AssetPaused error

File: cap-contracts/contracts/lendingPool/libraries/BorrowLogic.sol

```
201:     function realizeRestakerInterest(ILender.LenderStorage storage $, address
 ↪  _agent, address _asset)
202:         public
203:         returns (uint256 realizedInterest)
204:     {
205:         ILender.ReserveData storage reserve = $.reservesData[_asset];
206:         uint256 unrealizedInterest;
207:         (realizedInterest, unrealizedInterest) = maxRestakerRealization($,
 ↪  _agent, _asset);
208:         reserve.lastRealizationTime[_agent] = block.timestamp;
209:
210:         if (realizedInterest == 0 && unrealizedInterest == 0) return 0;
211:
212:         reserve.debt += realizedInterest;
213:         reserve.unrealizedInterest[_agent] += unrealizedInterest;
214:         reserve.totalUnrealizedInterest += unrealizedInterest;
215:
216:         IDebtToken(reserve.debtToken).mint(_agent, realizedInterest +
 ↪  unrealizedInterest);
217:@>       IVault(reserve.vault).borrow(_asset, realizedInterest, $.delegation);
218:         IDelegation($.delegation).distributeRewards(_agent, _asset);
219:         emit RealizeInterest(_asset, realizedInterest, $.delegation);
220:     }
```

Internal Pre-conditions

An asset is paused

External Pre-conditions

n/a

Attack Path

n/a

Impact

The protocol cannot get debt repaid from agents on paused assets.

PoC

```
pragma solidity ^0.8.28;

import { IOracle } from "../contracts/interfaces/IOracle.sol";
import { TestDeployer } from "./deploy/TestDeployer.sol";
import { console } from "forge-std/console.sol";

contract POC is TestDeployer {
    address user_agent;

    function setUp() public {
        _deployCapTestEnvironment();

        _initTestVaultLiquidity(usdVault);
        _initSymbioticVaultsLiquidity(env);

        user_agent = _getRandomAgent();

        vm.startPrank(env.symbiotic.users.vault_admin);
        _symbioticVaultDelegateToAgent(symbioticWethVault,
        ↪  env.symbiotic.networkAdapter, user_agent, 100e18);
        vm.stopPrank();
    }

    function test_submissionValidity() public {
        _setAssetOraclePrice(address(weth), 2000e8);
        vm.startPrank(user_agent);
        lender.borrow(address(usdc), 11000e6, user_agent);
        vm.stopPrank();

        vm.startPrank(env.users.access_control_admin);
        accessControl.grantAccess(lender.pauseAsset.selector, address(lender),
        ↪  env.users.vault_config_admin);
        vm.stopPrank();

        vm.startPrank(env.users.vault_config_admin);
        cUSD.pauseAsset(address(usdc));
        lender.pauseAsset(address(usdc), true);
        vm.stopPrank();

        vm.startPrank(user_agent);
        _timeTravel(90 days);
        usdc.approve(address(lender), 5000e6);

        vm.expectRevert(abi.encodeWithSignature("AssetPaused(address)",
        ↪  address(usdc)));
        lender.repay(address(usdc), 5000e6, user_agent);
        vm.stopPrank();
```

```
    }
}
```

Mitigation

```
diff --git a/cap-contracts/contracts/lendingPool/libraries/BorrowLogic.sol
 ↪  b/cap-contracts/contracts/lendingPool/libraries/BorrowLogic.sol
index 3c2b60a..e36f043 100644
--- a/cap-contracts/contracts/lendingPool/libraries/BorrowLogic.sol
+++ b/cap-contracts/contracts/lendingPool/libraries/BorrowLogic.sol
@@ -214,8 +214,10 @@ library BorrowLogic {
         reserve.totalUnrealizedInterest += unrealizedInterest;

         IDebtToken(reserve.debtToken).mint(_agent, realizedInterest +
          ↪  unrealizedInterest);
-        IVault(reserve.vault).borrow(_asset, realizedInterest, $.delegation);
-        IDelegation($.delegation).distributeRewards(_agent, _asset);
+        if (realizedInterest > 0) {
+            IVault(reserve.vault).borrow(_asset, realizedInterest, $.delegation);
+            IDelegation($.delegation).distributeRewards(_agent, _asset);
+        }
         emit RealizeInterest(_asset, realizedInterest, $.delegation);
     }
```

After applying the patch, run the POC again to see if repayment on paused asset is possible.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/cap-labs-dev/cap-contracts/pull/189

Issue M-5: Agents can evade the full extent of a slash

Source: https://github.com/sherlock-audit/2025-07-cap-judging/issues/316

Found by

Uddercover

Summary

The Cap Delegation.sol::slashTimestamp function returns a timestamp for historically determination of an agent's slashable stake, but this returned timestamp can make it possible for an operator to evade the full extent of a slash.

The `slashTimestamp` is chosen as either the agent's last borrow timestamp or the last delegation epoch start timestamp, whichever is later. But this then creates only a small window for the agent to be slashable. An agent's slashable collateral can fall in the same block as when they borrow, for example if their operator shares is reduced in the symbiotic delegator and they frontrun the reduction transaction with a borrow. Since the `slashTimestamp` will always be the last borrow time or later, the agent's slashable stake would always return less than it should be. The readme states that the cap delegation epoch duration will always be shorter than the symbiotic vault epoch duration, so even if the agent can have a greater penalty by the symbiotic vault standards, they will be able to get off with less.

Root Cause

Cap protocol's slashTimestamp() function returns a timestamp that allows an agent to avoid the full penalty of a slash

Internal Pre-conditions

NIL

External Pre-conditions

The agent's slashable collateral falls in the same block as they borrow from cap

Attack Path

1. Agent borrows amount from `Lender.sol`

2. Agent frontruns symbiotic `OPERATOR_NETWORK_SHARES_SET_ROLE` holder transaction to reduce agent vault shares to 0, with a borrow in the same block
3. Any subsequent attempts to slash the agent becomes impossible as `slashTimestamp` always returns slashable collateral as 0

For example, take a symbiotic vault epoch duration of 60 seconds, a current block.timestamp of 0 and a cap epoch duration of 10 seconds.

- At block.timestamp = 0 seconds: Agent borrows from cap vaults
- At block.timestamp = 30 seconds: Agent frontruns shares reduction transaction with minimum amount borrow.
- At block.timestamp = 31 seconds: `cap epoch = 3`, `slash timestamp = 30` i.e the last borrow timestamp, and agent slashable stake = 0
- At block.timestamp = 50 seconds: `cap epoch = 5` and 'slash timestamp = (5 - 1) * 10 = 40
- Agent's slashable stake and coverage at 40 = 0,
- Agent evades slashing even as the symbiotic epoch still considers that agent as slashable

Impact

Agents are not fully penalized, even potentially evading penalty completely

PoC

Add the following test function to Lender.borrow.t.sol:

```
function testAgentEvadeSlash() public {
        //agent borrows
        vm.startPrank(user_agent);
        assertEq(usdc.balanceOf(user_agent), 0);
        lender.borrow(address(usdc), 500e6, user_agent);
        assertGt(usdc.balanceOf(user_agent), 0);
        vm.stopPrank();

        //some time later, agent front runs share update with another borrow
        vm.warp(block.timestamp + 5 days);
        vm.startPrank(user_agent);
        lender.borrow(address(usdc), type(uint256).max, user_agent);
        vm.stopPrank();

        vm.startPrank(env.symbiotic.users.vault_admin);
        _symbioticVaultDelegateToAgent(symbioticWethVault,
        ↪  env.symbiotic.networkAdapter, user_agent, 0);
        vm.stopPrank();
```

```
        vm.warp(block.timestamp + 1);

        uint256 slashableCollateral = delegation.slashableCollateral(user_agent);
        uint256 coverage = delegation.coverage(user_agent);
        assertEq(slashableCollateral, 0);
        assertEq(coverage, 0);

        (,,,,, uint256 health) = lender.agent(user_agent);
        assertEq(health, 0);
    }
```

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/cap-labs-dev/cap-contracts/pull/198

Issue M-6: VaultAdapter::multiplier not initialized can lead first borrows to have `utilizationRate` = 0

Source: https://github.com/sherlock-audit/2025-07-cap-judging/issues/396

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

HeckerTrieuTien, magiccentaur, montecristo

Summary

InterestRate calculted in `DebtToken::_nextInterestRate()` is composed of 2 value: rate (marketRate or benchmarkRate) plus an `utilizationRate`. For an issue in `VaultAdapter` the utilizationRate for the first borrowers can be 0, leading to a lower cost for the users and a loss of yield for protocol.

Root Cause

https://github.com/sherlock-audit/2025-07-cap/blob/main/cap-contracts/contracts/oracle/libraries/VaultAdapter.sol#L27-L50

Rate is a function that changes the storage. `utilization` value is retrieved by `rate` under different cases.

https://github.com/sherlock-audit/2025-07-cap/blob/main/cap-contracts/contracts/oracle/libraries/VaultAdapter.sol#L79-L95

`multiplier` is a storage variable auto-initialized to 0. In `_applySlopes` if the first user borrows a large amount of tokens such that `_utilization > slopes.kink` we have this formula:

```
utilizationData.multiplier = utilizationData.multiplier
                * (1e27 + (1e27 * excess / (1e27 - slopes.kink)) * (_elapsed *
                ↪  $.rate / 1e27)) / 1e27;
```

utilizationData.multiplier will be multiplied for a number, but is 0 so final `multiplier` will be 0:

```
utilizationData.multiplier= 0 * (1e27 + (1e27 * excess / (1e27 - slopes.kink)) *
↪  (_elapsed * $.rate / 1e27)) / 1e27 = 0
```

So:

```
interestRate = (slopes.slope0 + (slopes.slope1 * excess / 1e27)) *
↪  utilizationData.multiplier / 1e27;
```

```
interestRate = (slopes.slope0 + (slopes.slope1 * excess / 1e27)) * 0 / 1e27 = 0
```

The final `interestRate` will be 0 too. New `utilizationData.multiplier` is stored as 0. For a second borrower (if remains `utilization > slopes.kink`) will have the same situation, and so on.

Series will be interrupted when, after some operations, we will have `utilization < slopes.kink`. Entering the `else` will set at least the multiplier to `minMultiplier` for:

```
 if (utilizationData.multiplier < $.minMultiplier) {
utilizationData.multiplier = $.minMultiplier;
}
```

At that point, the storage will contain a number != 0.

Internal Pre-conditions

amount borrowed by first user must lead to `utilization > slopes.kink`.

External Pre-conditions

--

Attack Path

1. First agent borrows a large amount of tokens such that: `utilization > slopes.kink` .
2. New value stored as multiplier: 0.
3. InterestRate (from utilization) is 0.
4. As long as the condition `utilization > slopes.kink` remains valid other agents can borrow at the basic condition interest because the new multiplier that should be used is always multiplied by the old one which is 0 (and the new value will be too).

Impact

Under the conditions, early borrowers borrow with a interest rate lower than expected. The protocol will receive less interests than it should. The higher the usage, the higher

the interest to be paid should be, instead it is 0 (only the base interest will be paid).

PoC

--

Mitigation

Consider the initialization of `multiplier` at the neutral value.

Issue M-7: post burn state fee calculation model can be gamed by burning small amounts

Source: https://github.com/sherlock-audit/2025-07-cap-judging/issues/449

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

kangaroo, lanrebayode77, moray5554

Summary

Protocol suppose to incentives users for burning CAP for popular assets by 0 fee and penalised if asset we want to burn for is ”rare” with fee up to 100%. There are 2 main functions we need to pay attention to understand the issue MinterLogic.sol

```
function _amountOutBeforeFee(address _oracle, IMinter.AmountOutParams memory params)
    internal
    view
    returns (uint256 amount, uint256 newRatio)
{
    (uint256 assetPrice,) = IOracle(_oracle).getPrice(params.asset);
    (uint256 capPrice,) = IOracle(_oracle).getPrice(address(this));

    uint256 assetDecimalsPow = 10 ** IERC20Metadata(params.asset).decimals();
    uint256 capDecimalsPow = 10 ** IERC20Metadata(address(this)).decimals();

    uint256 capSupply = IERC20(address(this)).totalSupply();
    uint256 capValue = capSupply * capPrice / capDecimalsPow;
    uint256 allocationValue = IVault(address(this)).totalSupplies(params.asset) *
    ↪  assetPrice / assetDecimalsPow;

    uint256 assetValue;
    if (params.mint) {
        assetValue = params.amount * assetPrice / assetDecimalsPow;
        if (capSupply == 0) {
            newRatio = 0;
            amount = assetValue * capDecimalsPow / assetPrice;
        } else {
            newRatio = (allocationValue + assetValue) * RAY_PRECISION / (capValue +
            ↪  assetValue);
            amount = assetValue * capDecimalsPow / capPrice;
        }
    } else {
        assetValue = params.amount * capPrice / capDecimalsPow;
        if (params.amount == capSupply) {
```

```
            newRatio = RAY_PRECISION;
            amount = assetValue * assetDecimalsPow / assetPrice;
        } else {
            if (allocationValue < assetValue || capValue <= assetValue) {
                newRatio = 0;
            } else {
                newRatio = (allocationValue - assetValue) * RAY_PRECISION /
                ↪  (capValue - assetValue);
            }
            amount = assetValue * assetDecimalsPow / assetPrice;
        }
    }
}

function _applyFeeSlopes(IMinter.FeeData memory fees, IMinter.FeeSlopeParams memory
↪  params)
    internal
    pure
    returns (uint256 amount, uint256 fee)
{
    uint256 rate;
    if (params.mint) {
        rate = fees.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.optimalRatio) /
                ↪  (fees.mintKinkRatio - fees.optimalRatio);
            }
        }
    } else {
        if (params.ratio < fees.optimalRatio) {
            if (params.ratio < fees.burnKinkRatio) {
                uint256 excessRatio = fees.burnKinkRatio - params.ratio;
                rate = fees.slope0 + (fees.slope1 * excessRatio /
                ↪  fees.burnKinkRatio);
            } else {
                rate = fees.slope0 * (fees.optimalRatio - params.ratio) /
                ↪  (fees.optimalRatio - fees.burnKinkRatio);
            }
        }
    }

    if (rate > RAY_PRECISION) rate = RAY_PRECISION;
    fee = params.amount * rate / RAY_PRECISION;
    amount = params.amount - fee;
```

```
}
```

`_amountOutBeforeFee` calculates amount and the ratio (share of the asset in the pool) and `_applyFeeSlopes` applies slope depends on the ratio we got. The problem here, the ratio we calculating essentially is a post burn state ratio.Means, the fee will be applied to the whole amount like as whole amount was burned at specific point in the slope. In reality, say, 30% of total amount can be burned above optimal ratio, another 20 bellow it, and another 50 bellow burn kink ratio - resulting lower fee than if we burn it one tx.

Let's illustrate it with an example:

```
fees = IMinter.FeeData({
    ...
    optimalRatio: 0.5e27,        // 50%
    mintKinkRatio: 0.8e27,       // 80%
    burnKinkRatio: 0.2e27,       // 20%
    slope0: 0.2e27,              // 20% fee slope before kink
    slope1: 0.8e27               // 80% fee slope after kink
});
```

I willing to burn 1000e18 tokens and the ratio (we got from `_amountOutBeforeFee()`) will be 0.05e27(5% allocation) If i do it one burn:

```
excessRatio = burnKinkRatio - ratio = 0.15e27
rate = slope0 + (slope1 * excessRatio / burnKinkRatio) = 0.8e27 (80%) -> fee
↪  800e18, receive 200e18
```

If i split it into 2 500e18 calls : 1st) rate = 0.2 + (0.8 * 0.08 / 0.2) = 0.2 + 0.32 = 0.52 (52%) fee = 500 * 0.52 = 260 receive = 240 2nd) rate = 0.2 + (0.8 * 0.15 / 0.2) = 0.2 + 0.6 = 0.8 (80%) fee = 500 * 0.8 = 400 receive = 100

In our example splitting let us moving ratio 0.2 to 0.12 and pays fee on this ratio first, and then 0.12 to 0.05 (the final ratio in our one tx burn()) In reality, attacker will burn everything before optimal ratio and then start splitting as more as close we go to ”fee cliffs”.

Root Cause

_amountOutBeforeFee(), _applyFeeSlopes() in MinterLogic.sol

Internal Pre-conditions

None, but at the fee cliff state or low volume basket it will drastically reduce the fee

External Pre-conditions

none

Attack Path

attacker split burn() into multiple calls in order to save on fees

Impact

User avoids fees while burn()

PoC

No response

Mitigation

No response

Issue M-8: Restaker rewards on zero coverage agent will be stolen by subsequent restaker interest realization

Source: https://github.com/sherlock-audit/2025-07-cap-judging/issues/467

Found by

Uddercover, attacker_code, montecristo, silver_eth

Summary

Lender contract will transfer realized restaker interest to Delegation contract. Delegation contract is supposed to transfer the received amount to network middleware and delegate rewards distribution.

However, in an edge case where agent's coverage is 0, Delegation contract skips such flow. Thus, received amount will be stuck in Delegation contract until the next restaker interest realization.

In this case, subsequent restaker interest realization will sweep stuck amount and target restakers receive more rewards than deserved.

Root Cause

Root cause lies in the fact that Delegation contract skips network reward distribution if agent's current coverage is 0:

File: cap-contracts/contracts/delegation/Delegation.sol

```
55: function distributeRewards(address _agent, address _asset) external {
56:         DelegationStorage storage $ = getDelegationStorage();
57:         uint256 _amount = IERC20(_asset).balanceOf(address(this));
58:
59:         uint256 totalCoverage = coverage(_agent);
60:@>       if (totalCoverage == 0) return;
61:
62:         address network = $.agentData[_agent].network;
63:         IERC20(_asset).safeTransfer(network, _amount);
64:         ISymbioticNetworkMiddleware(network).distributeRewards(_agent, _asset);
```

However, Delegation contract already has received restaker interest from Lender contract:

File: cap-contracts/contracts/lendingPool/libraries/BorrowLogic.sol

```
217:@>         IVault(reserve.vault).borrow(_asset, realizedInterest, $.delegation);
218:@>       IDelegation($.delegation).distributeRewards(_agent, _asset);
```

Thus, restaker interest will be stuck at Delegation contract.

Due to L57, next restaker interest realization will sweep this amount to another subnetwork.

Internal Pre-conditions

1. Debt repay or liquidation happens on zero-coverage agent's position.

Zero coverage can happen in one of the following scenarios:

- All restaked amount are scheduled to be withdrawn
- Agent's network limit is set to 0
- Protocol team repays on bad debt position. More specifically, previous liquidation already set agent's delegation to zero.

External Pre-conditions

n/a

Attack Path

n/a

Impact

Since `realizeRestakerInterest` is public, restaker rewards can be stolen by any interested party (e.g. agent or restaker)

PoC

No response

Mitigation

Don't send restaker interest to Delegation contract if coverage is 0.

Instead, send it to FeeAuction contract so that the interest is distributed among scUSD holders.

Alternatively, one can fix `maxRestakerRealization` so that realized interest amount is 0 when agent's coverage is 0.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/cap-labs-dev/cap-contracts/pull/204

Issue M-9: Missing slippage protection in liquidation allows unexpected collateral loss

Source: https://github.com/sherlock-audit/2025-07-cap-judging/issues/542

Found by

0xsh, Drynooo, Tigerfrake, dobrevaleri, farismaulana, silver_eth, valuevalk

Summary

Liquidators cannot specify a minimum amount of collateral to receive during liquidation, exposing them to potential losses when slashable collateral is less than expected.

Root Cause

In LiquidationLogic::liquidate(), the function lacks a parameter for liquidators to specify a minimum acceptable amount of collateral to receive. The function contains a safety check that caps the liquidation value to the total slashable collateral:

```
    function liquidate(ILender.LenderStorage storage $, ILender.RepayParams memory
    ↪  params)
        external
        returns (uint256 liquidatedValue)
    {
        (uint256 totalDelegation, uint256 totalSlashableCollateral, uint256
        ↪  totalDebt,,, uint256 health) =
            ViewLogic.agent($, params.agent);


        ValidationLogic.validateLiquidation(
            health,
            totalDelegation * $.emergencyLiquidationThreshold / totalDebt,
            $.liquidationStart[params.agent],
            $.grace,
            $.expiry
        );


        (uint256 assetPrice,) = IOracle($.oracle).getPrice(params.asset);
        uint256 bonus = ViewLogic.bonus($, params.agent);
        uint256 maxLiquidation = ViewLogic.maxLiquidatable($, params.agent,
        ↪  params.asset);
        uint256 liquidated = params.amount > maxLiquidation ? maxLiquidation :
        ↪  params.amount;
```

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


        (,,,,, health) = ViewLogic.agent($, params.agent);
        if (health >= 1e27) _closeLiquidation($, params.agent);


@>      liquidatedValue =
            (liquidated + (liquidated * bonus / 1e27)) * assetPrice / (10 **
            ↪  $.reservesData[params.asset].decimals);
@>      if (totalSlashableCollateral < liquidatedValue) liquidatedValue =
↪   totalSlashableCollateral;


        if (liquidatedValue > 0) IDelegation($.delegation).slash(params.agent,
        ↪  params.caller, liquidatedValue);


        emit Liquidate(params.agent, params.caller, params.asset, liquidated,
        ↪  liquidatedValue);
    }
```

However, there's no corresponding check to ensure liquidators receive at least a specified minimum amount. This means liquidators are forced to accept whatever collateral is available, which could be significantly less than what they paid to repay the debt.

The function calculates the expected liquidation value based on the repaid debt amount, asset price, and bonus.

But when `totalSlashableCollateral < liquidatedValue`, the liquidator receives less value than calculated, with no option to revert the transaction.

The issue is viable, because the `totalSlashableCollateral` is calcualted based on the epoch, which is calculated based on the timestamp. So the epoch can change between the off-chain calculation and the transaction execution, which could lead to lesser collateral available for seizing. Moreover, another liquidation (for another asset) could be executed first, which will reduce the `totalSlashableCollateral` and there is no mechanism for protecting agains such scenarios.

Internal Pre-conditions

1. Agent's position must have less slashable collateral than what would be expected based on their debt
2. Liquidator needs to call `liquidate()` with an amount parameter for an asset the agent has borrowed

External Pre-conditions

1. Collateral value must have decreased since the agent's position was opened

Attack Path

1. An agent's position becomes unhealthy
2. Liquidator opens liquidation via `Lender::openLiquidation(agent)`
3. After grace period, liquidator calls `Lender::liquidate(agent, asset, amount)`
4. System calculates expected collateral to receive: `liquidatedValue`
5. System checks if `totalSlashableCollateral < liquidatedValue`
6. If true, liquidator receives `totalSlashableCollateral` instead of `liquidatedValue`
7. Transaction completes successfully, but liquidator receives less collateral than expected
8. Liquidator suffers unexpected losses

Impact

The liquidators can receive less than expected and there is no way to protect against such scenarios.

PoC

No response

Mitigation

Add a minimum collateral received parameter to the liquidate function, which will act as a guard.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/cap-labs-dev/cap-contracts/pull/205

Issue M-10: `ViewLogic::maxLiquidatable()` doesn't take the bonus into account, making the agent liquidatable again

Source: https://github.com/sherlock-audit/2025-07-cap-judging/issues/638

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

0x73696d616f, 0xzey, Albert_Mei, Drynooo, KrisRenZo, anirruth_, maxim371, montecristo, ustas

Summary

ViewLogic::maxLiquidatable() computes the maximum debt to liquidate such that the resulting target health is exactly the desired:

```
maxLiquidatableAmount = (($.targetHealth * totalDebt) - (totalDelegation *
↪  liquidationThreshold)) * decPow
    / (($.targetHealth - liquidationThreshold) * assetPrice);
```

However, this is inaccurate due to the bonus, which affects the final health as it slashes more than the debt is repays.

Thus, it will trigger more than 1 liquidations in certain conditions and lead to heavy losses for the agent.

Root Cause

In `ViewLogic.sol:91`, it doesn't include the bonus.

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1. Agent is liquidated twice in a row due to the `ViewLogic::maxLiquidatable()` calculation not taking into account the bonus.

Impact

Agent suffers more losses than supposed.

PoC

No response

Mitigation

Include the bonus in the debt to repay so the health factor equals the target.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.