Zellic

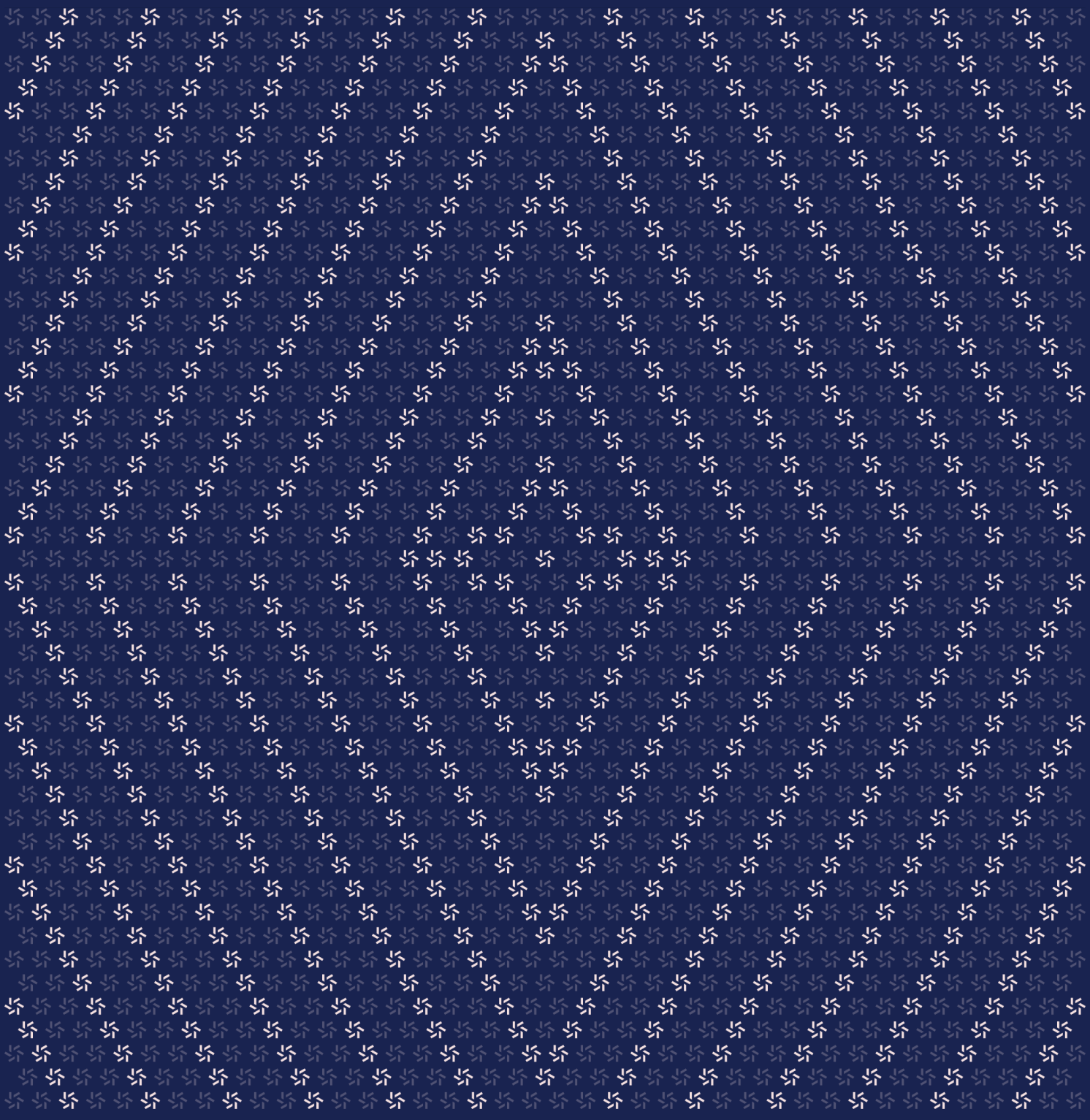
Prepared for
CAP team

Prepared by
Chongyu Lv
Mohit Sharma
Zellic

March 17, 2025

CAP Labs Core Contracts

Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for the CAP team from February 17th to March 7th, 2025. During this engagement, Zellic reviewed CAP Labs Core contracts' code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Does the lending functionality operate correctly and securely?
- Are there any vulnerabilities in the rounding mechanisms that could be exploited?
- Are there initialization issues that could compromise contract security?
- Is the overall lending logic sound and resistant to manipulation?
- Are there any vectors that would allow infinite minting of tokens?
- Does the liquidation logic function as intended without exploitable vulnerabilities?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

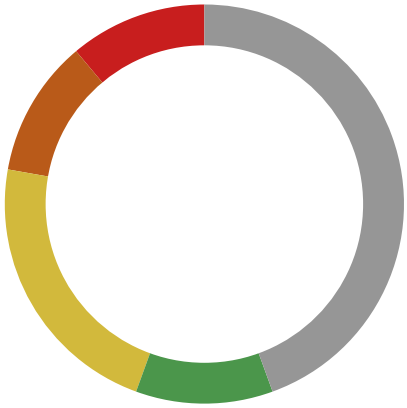
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped CAP Labs Core contracts, we discovered nine findings. One critical issue was found. One was of high impact, two were of medium impact, one was of low impact, and the remaining findings were informational in nature.

Breakdown of Finding Impacts

Impact Level	Count
Critical	1
High	1
Medium	2
Low	1
Informational	4



2. Introduction

2.1. About CAP Labs Core Contracts

The CAP team contributed the following description of CAP Labs Core contracts:

> CAP is a redeemable stablecoin protocol with certain financial guarantees. Functioning as a shared security network on both EigenLayer and Symbiotic, CAP tasks operators with generating yield on the collateral assets of its stablecoins. Restakers are tasked with underwriting the risk of operator activity. Stablecoin holders earn yield by staking their stablecoins.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case

basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3. Scope

The engagement involved a review of the following targets:

CAP Labs Core Contracts

Type	Solidity
Platform	EVM-compatible
Target	cap-contracts
Repository	https://github.com/cap-labs-dev/cap-contracts ↗
Version	`de71faf8368c0632cb8d51cd4ff44670bb18fbe2`
Programs	`contracts/*`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 3.8 person-weeks. The assessment was conducted by two consultants over the course of three calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Chongyu Lv
Engineer
chongyu@zellic.io

Mohit Sharma
Engineer
mohit@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

February 17, 2025	Kick-off call
February 17, 2025	Start of primary review period
March 4, 2025	End of primary review period

3. Detailed Findings

3.1. Calculation error in `ValidationLogic::validateBorrow` resulting in a loss of funds

Target	ValidationLogic		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The `Lender::Borrow` function will call the `BorrowLogic::borrow` function to perform some borrow-related validation. The main validation logic is in the `ValidationLogic::validateBorrow` function:

```
function validateBorrow(ILender.LenderStorage storage $,
    ILender.BorrowParams memory params) external {
    // ...
    uint256 borrowCapacity = totalDelegation * ltv;

    if (newTotalDebt > borrowCapacity)
    revert CollateralCannotCoverNewBorrow();

    IDelegation($.delegation).setLastBorrow(params.agent);
}
```

Normally, this function will first calculate the maximum amount that the agent can borrow (i.e., `borrowCapacity`). When the total amount that the agent actually borrows (i.e., `newTotalDebt`) exceeds its `borrowCapacity`, it will `revert CollateralCannotCoverNewBorrow`.

However, this function does not divide by 1e27 when calculating `borrowCapacity`. This causes the total amount that the agent actually borrows to increase by 1e27 times the original amount.

Impact

A malicious agent can simply borrow more than the collateral value and liquidate itself to profit.

Assume that a malicious agent pledges $6,200 worth of collateral in the protocol, the protocol's loan-to-value (LTV) ratio is 50%, and the remaining borrowable USDC in the Vault is $12,000. In this case, if the `ValidationLogic::validateBorrow` function can correctly calculate the agent's `borrowCapacity`, then the agent can borrow $3,100 worth of USDC through the `Lender::borrow` function.

However, due to a calculation error, the agent can actually borrow $3,100e27 worth of USDC, far exceeding the value of the collateral, which is $6,200.

The malicious agent can even directly clear all the USDC in the Vault. Here is a proof of concept:

```
function test_audit_lender_borrow_1() public {
    vm.startPrank(user_agent);

    uint256 agent_coverage = delegation.coverage(user_agent);
    uint256 agent_ltv = delegation.ltv(user_agent);
    uint256 agent_LT = delegation.liquidationThreshold(user_agent);
    console2.log("-------------");
    console2.log("agent_coverage :", agent_coverage);
    console2.log("agent_LTV      :", agent_ltv);
    console2.log("agent_LT       :", agent_LT);

    uint256 backingBefore = usdc.balanceOf(address(cUSD));
    console2.log("-------------");
    console2.log("before Lender::borrow ,,, backingBefore              :",
    backingBefore);
    console2.log("before Lender::borrow ,,, usdc.balanceOf(user_agent) :",
    usdc.balanceOf(user_agent));
    console2.log("-------------");

    lender.borrow(address(usdc), backingBefore, user_agent);

    console2.log("after  Lender::borrow ,,, backingAfter               :",
    usdc.balanceOf(address(cUSD)));
    console2.log("after  Lender::borrow ,,, usdc.balanceOf(user_agent) :",
    usdc.balanceOf(user_agent));
    vm.stopPrank(); // user_agent
}
```

And here is its result:

```
-------------
agent_coverage :  620000000000
agent_LTV      : 500000000000000000000000000
agent_LT       : 700000000000000000000000000
-------------
before Lender::borrow ,,, backingBefore              : 12000000000
before Lender::borrow ,,, usdc.balanceOf(user_agent) : 0
-------------
in ValidationLogic::validateBorrow ,,,
    borrowCapacity:  3100000000000000000000000000000000000000
after  Lender::borrow ,,, backingAfter               : 0
```

```
after  Lender::borrow ,,, usdc.balanceOf(user_agent) : 12000000000
```

Recommendations

Correct the calculation error in `ValidationLogic::validateBorrow` like this:

```
diff --git a/contracts/lendingPool/libraries/ValidationLogic.sol
    b/contracts/lendingPool/libraries/ValidationLogic.sol
index c134813..13ff8a4 100644
-- a/contracts/lendingPool/libraries/ValidationLogic.sol
++ b/contracts/lendingPool/libraries/ValidationLogic.sol
@@ -69,7 +69,7 @@ library ValidationLogic {
         uint256 ltv = IDelegation($.delegation).ltv(params.agent);
         uint256 assetPrice = IOracle($.oracle).getPrice(params.asset);
         uint256 newTotalDebt = totalDebt + (params.amount * assetPrice
    / (10 ** reserve.decimals));
-        uint256 borrowCapacity = totalDelegation * ltv;
+        uint256 borrowCapacity = totalDelegation * ltv / 1e27;

         if (newTotalDebt > borrowCapacity)
    revert CollateralCannotCoverNewBorrow();
```

Remediation

This issue has been acknowledged by CAP team, and a fix was implemented in commit 70893b96 ↗.

3.2. Missing safety checks for LTV and liquidation threshold

Target	Delegation		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

In `Delegation::addAgent`, the agent's liquidation threshold is enforced to be less than 100%.

```
if (_liquidationThreshold > 1e27) revert InvalidLiquidationThreshold();
```

But this check is missing in `Delegation::modifyAgent`. Hence, an agent can be created and later modified to have a liquidation threshold of over 100%.

Additionally, no checks are enforced on the agent's LTV in either `Delegation::addAgent` or `Delegation::modifyAgent`. We also note the lack of a safety buffer between liquidation threshold and LTV to prevent unfair liquidations.

Impact

A liquidation threshold over 100% can allow for insolvent positions to remain liquid, causing bad debt. Similarly, an LTV over 100% can lead to undercollateralized loans. Both these values should be constrained to be under 100% for all agents.

The liquidation threshold for an agent should always be more than the corresponding LTV by some safety buffer. This is crucial as without a safety buffer, positions at the edge of liquidation can be penalized from minor market movements.

Recommendations

We recommend adding the following checks in `Delegation::addAgent` and `Delegation::modifyAgent`:

- LTV is under 100%
- Liquidation threshold is under 100%
- Liquidation threshold > LTV + safety buffer

The value of the safety buffer must be determined by the protocol developers as per practical tokenomic considerations.

Remediation

This issue has been acknowledged by CAP team, and a fix was implemented in commit `81a1a9e6` ↗.

3.3. Potential denial of service in looping Delegation contract's agents and networks array

Target	Delegation		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `Delegation::addAgent` function can add a new agent to the `$.agents` array:

```
/// @notice Add agent to be delegated to
/// @param _agent Agent address
/// @param _ltv Loan to value ratio
/// @param _liquidationThreshold Liquidation threshold
function addAgent(address _agent, uint256 _ltv, uint256 _liquidationThreshold)
    external
    checkAccess(this.addAgent.selector)
{
    // if liquidation threshold is greater than 100%, agent
    // could borrow more than they are collateralized for
    if (_liquidationThreshold > 1e27) revert InvalidLiquidationThreshold();

    DelegationStorage storage $ = getDelegationStorage();

    // If the agent already exists, we revert
    for (uint i; i < $.agents.length; ++i) {
        if ($.agents[i] == _agent) revert DuplicateAgent();
    }

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

This function uses a for loop to check whether the new agent already exists in the `$.agents` array. If not, the new agent is pushed to the `$.agents` array.

However, in the Delegation contract, there is no `removeAgent` logic corresponding to `addAgent` to delete the agent, thereby reducing the length of the `$.agents` array. Therefore, the length of the

`$.agents` array can only increase gradually and cannot be reduced.

Impact

Both the `Delegation::addAgent` function and `Delegation::modifyAgent` function need to traverse the `$.agents` array to check for duplicates — time complexity O(n). It may revert due to being out of gas, resulting in denial of service.

Take `Delegation::modifyAgent` as an example; if the `Delegation::modifyAgent` function is called to modify the LTV of an agent at a later position in the `$.agents` array, it may revert due to being out of gas:

```
/// @notice Modify an agents config only callable by the operator
/// @param _agent the agent to modify
/// @param _ltv Loan to value ratio
/// @param _liquidationThreshold Liquidation threshold
function modifyAgent(address _agent, uint256 _ltv,
    uint256 _liquidationThreshold)
    external
    checkAccess(this.modifyAgent.selector)
{
    DelegationStorage storage $ = getDelegationStorage();

    // Check that the agent exists
    for (uint i; i < $.agents.length; ++i) {
        if ($.agents[i] == _agent) {
            $.agentData[_agent].ltv = _ltv;
            $.agentData[_agent].liquidationThreshold = _liquidationThreshold;
            emit ModifyAgent(_agent, _ltv, _liquidationThreshold);
            return;
        }
    }
    revert AgentDoesNotExist();
}
```

Here is a proof of concept:

```
function test_audit_poc_DOS() public {
    vm.startPrank(env.users.delegation_admin);
    address tmp_agents;
    // create 1000 agents
    uint256 numberOfAgents = 100;
    for (uint256 i = 0; i < numberOfAgents; i++) {
        tmp_agents = address(uint160(0xdeadbeef0000 + i));
        delegation.addAgent(tmp_agents, 0.9e18, 0.8e18);
```

```
        delegation.registerNetwork(tmp_agents,
    env.symbiotic.networkAdapter.networkMiddleware);
    }
    address lastAgent = address(0x11223344aabbccdd);
    delegation.addAgent(lastAgent, 11223344, 0.8e18);
    console2.log("before modifyAgent function ,,, tvl of lastAgent: ",
    delegation.ltv(lastAgent));

    uint256 gasBefore = gasleft();
    delegation.modifyAgent(lastAgent, 22222222, 0.8e18);
    uint256 gasAfter = gasleft();

    console2.log("after  modifyAgent function ,,, tvl of lastAgent: ",
    delegation.ltv(lastAgent));
    console2.log("gas used: ", gasBefore - gasAfter);

    vm.stopPrank();
}
```

When `numberOfAgents = 100`, the result is as follows:

```
before modifyAgent function ,,, tvl of lastAgent:  11223344
after  modifyAgent function ,,, tvl of lastAgent:  22222222
gas used:  58564
```

When `numberOfAgents = 1000`, the result is as follows:

```
before modifyAgent function ,,, tvl of lastAgent:  11223344
after  modifyAgent function ,,, tvl of lastAgent:  22222222
gas used:  501364
```

Recommendations

Consider adding a `removeAgent` function in the Delegation contract to delete redundant agents. Or use a mapping to store agent information to reduce a large number of array-traversal operations.

Remediation

This issue has been acknowledged by CAP team, and a fix was implemented in commit `fab7f93d` ↗.

3.4. Missing staleness check in `PriceOracle::getPrice`

Target	PriceOracle		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `PriceOracle::getPrice` function only gets the `price`, but it does not get the `updatedAt` time of the `price`, so it is impossible to check whether the price is expired.

```
/// @notice Fetch the price for an asset
/// @dev If initial price fetch fails then a backup source is used, never
    reverts
/// @param _asset Asset address
/// @return price Price of the asset
function getPrice(address _asset) external view returns (uint256 price) {
    PriceOracleStorage storage $ = getPriceOracleStorage();
    IOracle.OracleData memory data = $.oracleData[_asset];

    price = _getPrice(data.adapter, data.payload);

    if (price == 0) {
        data = $.backupOracleData[_asset];
        price = _getPrice(data.adapter, data.payload);
    }
}
```

Impact

In this case, code will execute with prices that do not reflect the current pricing, resulting in a potential loss of funds for users.

Recommendations

When getting the Chainlink oracle feed price, get the price itself and the `updatedAt` corresponding to the price. Also, add a staleness check to ensure that the oracle price has not expired.

Remediation

This issue has been acknowledged by CAP team, and a fix was implemented in commit `f2be6e04` ↗.

Note that the `_getPrice` function itself makes a static call to the underlying adapter, which is configured at deploy time. The team should ensure that the adapter returns `lastUpdated`

3.5. Centralization risk from lack of safety checks in `removeAsset`

Target	Vault		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The vault allows assets to be added and removed dynamically. The `removeAsset` method implements no constraints on the removed asset.

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

    if (!removed) revert AssetNotSupported(_asset);

    emit RemoveAsset(_asset);
}
```

This means a compromised admin account can unlist any asset regardless of the available user balance.

Impact

The contract also implements the `rescueERC20` function, which allows an admin to drain any unlisted assets mistakenly transferred to the vault.

This, combined with unconstrained asset removal, allows a compromised admin account to freely call `removeAsset` on any listed asset and drain the vault of that asset.

Recommendations

One naive fix for this issue is to disallow the removal of any assets with a nonzero balance. This can however be abused as a malicious actor can transfer some tokens to the vault contract to block an asset from being removed. Another potential fix can be to implement a buffer period for `rescueERC20` if it is called on a previously listed asset.

Remediation

This issue has been acknowledged by CAP team, and a fix was implemented in commit 40869760 ↗.

3.6. Utilization rate can exceed 100%

Target	Vault		
Category	Business Logic	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The vault keeps track of an asset's utilization rate as the ratio of borrowed tokens to the total supply of that asset present within the vault. It is calculated as such:

```
function utilization(IVault.VaultStorage storage $, address _asset)
    public view returns (uint256 ratio) {
    ratio = $.totalSupplies[_asset] != 0 ? $.totalBorrows[_asset]
    * 1e27 / $.totalSupplies[_asset] : 0;
}
```

This value is expected to be lower than 1e27. The `$.totalSupplies` map is mutated only during mints and burns, while the `$.totalBorrows` map is updated in the `borrow` method. However, this method uses `IERC20(params.asset).balanceOf(address(this));` to determine the maximum borrowable amount.

```
function borrow(IVault.VaultStorage storage $,
    IVault.BorrowParams memory params)
    external
    whenNotPaused($, params.asset)
    updateIndex($, params.asset)
{
    uint256 balanceBefore = IERC20(params.asset).balanceOf(address(this));
    if (balanceBefore < params.amount)
    revert InsufficientReserves(params.asset, balanceBefore, params.amount);

    $.totalBorrows[params.asset] += params.amount;
    IERC20(params.asset).safeTransfer(params.receiver, params.amount);

    emit Borrow(msg.sender, params.asset, params.amount);
}
```

Since an attacker can transfer ERC-20 tokens to the Vault without depositing them, this means the vault's asset balance can be artificially inflated without changing `$.totalSupplies`. These tokens can then also be used in a borrow, leading to a case where the value of `$.totalBorrows[_asset]`

can exceed `$.totalSupplies[_asset]`, making the utilization rate higher than 100%.

Impact

During our assessment, we did not find demonstrable impact on the protocol for this behavior, but we recommend fixing it in order to avoid violating downstream assumptions.

Recommendations

One straightforward fix for this is to use the `IVault.availableBalance` function to calculate the max borrowable amount, which only takes into account the deposited tokens.

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

Remediation

This issue has been acknowledged by CAP team, and a fix was implemented in commit a2c6d9dd ↗.

3.7. The `FeeAuction::setMinStartPrice` function is missing a check

Target	FeeAuction		
Category	Business Logic	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The `FeeAuction::setMinStartPrice` function is used to set the minimum starting price in FeeAuction:

```
/// @notice Set minimum start price
/// @param _minStartPrice New minimum start price
function setMinStartPrice(uint256 _minStartPrice)
    external checkAccess(this.setMinStartPrice.selector) {
    FeeAuctionStorage storage $ = get();
    $.minStartPrice = _minStartPrice;
    emit SetMinStartPrice(_minStartPrice);
}
```

However, this function does not guarantee that `$.minStartPrice` is greater than zero.

Impact

If `$.minStartPrice` is accidentally set to zero, there is a possibility that the `FeeAuction::buy` function can continuously purchase for free with a price of 0.

Recommendations

Consider adding a check in `FeeAuction::setMinStartPrice` to ensure that `$.minStartPrice` is always greater than zero.

Remediation

This issue has been acknowledged by CAP team, and a fix was implemented in commit 363bb236 ↗.

3.8. Rounding error in `MinterLogic::redeemAmountOut`

Target	MinterLogic		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The `Vault::redeem` function is used to redeem the whitelisted assets that the user has previously minted to the Vault. The `Vault::redeem` function calls the `MinterLogic::redeemAmountOut` function to calculate how much USDX can be redeemed based on the amount of cUSD paid at the time.

However, there may be rounding errors in the `MinterLogic::redeemAmountOut` function:

```
uint256 shares = params.amount * 1e27 / IERC20(address(this)).totalSupply();
```

Impact

If the amount of cUSD redeemed by the user is less than `IERC20(address(this)).totalSupply()/1e27`, rounding errors may occur, although users can set slippage to prevent capital loss.

Recommendations

Consider adding a check to ensure that the asset redeemed by the redeem function is always greater than zero.

Remediation

This issue has been acknowledged by CAP team, and a fix was implemented in commit 8aa35708 ↗.

3.9. Incorrect length check in `VaultLogic::redeem`

Target	VaultLogic		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the `VaultLogic::redeem` function, the code should check that `params.minAmountsOut.length` is equal to `params.amountsOut.length`. However, the code incorrectly checks that `params.amountsOut.length` is equal to `$.assets.length`.

```
if (params.amountsOut.length != $.assets.length)
    revert InvalidMinAmountsOut();
```

Impact

This is a meaningless check because in the `MinterLogic::redeemAmountOut` function, when the `params.amountsOut` array is created, its length is `$.assets.length`, so `params.amountsOut.length` and `$.assets.length` are always equal.

```
function redeemAmountOut(IMinter.MinterStorage storage $,
    IMinter.RedeemAmountOutParams memory params)
    external
    view
    returns (uint256[] memory amounts)
{
    // ...
    address[] memory assets = IVault(address(this)).assets();
    uint256 assetLength = assets.length;
    amounts = new uint256[](assetLength); // @amounts is params.amountsOut
    // ...
}
```

Recommendations

Consider changing the array-length check in the `VaultLogic::redeem` function to the following:

```
if (params.amountsOut.length != params.minAmountsOut.length)
    revert InvalidMinAmountsOut();
```

Remediation

This issue has been acknowledged by CAP team, and a fix was implemented in commit fba778e1 ↗.

Function: `buy(address[] _assets, address _receiver, bytes _callback)`

Buy fees in exchange for the payment token

Inputs

- `_assets`
  - **Control**: Arbitrary
  - **Constraints**: None
  - **Impact**: Assets to buy
- `_receiver`
  - **Control**: Arbitrary
  - **Constraints**: None
  - **Impact**: Receiver address for the assets
- `_callback`
  - **Control**: Arbitrary
  - **Constraints**: None
  - **Impact**: Data for caller's callback

Branches and code coverage

Intended branches

- Auction timestamp is reset to current time
  - ☑ Test coverage
- Auction price doubles after buy
  - ☑ Test coverage

Function: `liquidate(address _agent, address _asset, uint256 _amount)`

Liquidate an agent when the health is below 1

Inputs

- `_agent`
  - **Control**: Arbitrary
  - **Constraints**: Cannot be 0 address.
  - **Impact**: Adddres of the agent to liquidate
- `_asset`
  - **Control**: Arbitrary
  - **Constraints**: Cannot be 0 address.
  - **Impact**: Asset to repay
- `_amount`
  - **Control**: Arbitrary
  - **Constraints**: None
  - **Impact**: Amount of asset to repay on behalf of the agent

Branches and code coverage

Intended branches

- Anyone can liquidate when agent is below liquidation threshold
  - ☑ Test coverage
- Interest is repaid after liquidation
  - ☑ Test coverage

Negative behavior

- Revert if agent health is over 1e27
  - ☐ Negative test
- Revert if agent emergency health over 1e27 and grace period is not over
  - ☐ Negative test
- Revert if agent emergency health over 1e27 and liquidation window has expired
  - ☐ Negative test

Function: `repay(address _asset, uint256 _amount, address _agent)`

Repay an asset on behalf of an agent

Inputs

- `_asset`
  - **Control**: Arbitrary
  - **Constraints**: Cannot be 0 address
  - **Impact**: Asset to repay
- `_amount`
  - **Control**: Arbitrary
  - **Constraints**: None
  - **Impact**: Amount to repay
- `_agent`
  - **Control**: Arbitrary
  - **Constraints**: Cannot be 0 address
  - **Impact**: Agent on whose behalf the asset is to be repaid

Branches and code coverage

Intended branches

- Principal debt tokens are burned first, followed by interest tokens and then restaker tokens
  - ☐ Test coverage
- If restaker tokens are repaid, the interest receiver distributes agent rewards
  - ☐ Test coverage

4. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped CAP Labs Core contracts, we discovered nine findings. One critical issue was found. One was of high impact, two were of medium impact, one was of low impact, and the remaining findings were informational in nature.

4.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.