Zellic

# Cega

## Smart Contract Security Assessment

March 15, 2023

*Prepared for:*

**Winston Zhang**

Sanic Pte. Ltd.

*Prepared by:*

**Syed Faraz Abrar and Aaron Esau**

Zellic Inc.

# Contents

**About Zellic** 3

**1 Executive Summary** 4

1.1 Goals of the Assessment . . . . . 4

1.2 Non-goals and Limitations . . . . . 4

1.3 Results . . . . . 4

**2 Introduction** 6

2.1 About Cega . . . . . 6

2.2 Methodology . . . . . 6

2.3 Scope . . . . . 7

2.4 Project Overview . . . . . 7

2.5 Project Timeline . . . . . 8

**3 Detailed Findings** 9

3.1 Missing valid vault address check in `processDepositQueue` . . . . . 9

3.2 A malicious or compromised trader admin may lead to locked funds . . 13

3.3 The `vaultAddress` validity check can be bypassed . . . . . 15

3.4 Ability to deposit on other users' behalf . . . . . 18

3.5 Missing status check in `openVaultDeposits` . . . . . 20

3.6 Missing sanity checks for crucial protocol parameters . . . . . 21

3.7 Gas griefing using zero-value deposits and withdrawals . . . . . 23

3.8 Missing checks and some access controls on critical functions . . . . . 24

3.9 Unused or unreachable functionality should be removed . . . . . 26

**4 Discussion** 28

4.1 Centralization risks . . . . . 28
4.2 Using non-ERC20 token standards . . . . . 28
4.3 Impossibility of reentrancy in the queue-processing functions . . . . . 29

**5 Threat Model** **30**
5.1 Module: CegaState.sol . . . . . 30
5.2 Module: FCNProduct.sol . . . . . 31
5.3 Module: Oracle.sol . . . . . 49

**6 Audit Results** **51**
6.1 Disclaimers . . . . . 51

## About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

# 1 Executive Summary

Zellic conducted a security assessment for Sanic Pte. Ltd. from February 13th to February 15th, 2023. During this engagement, Zellic reviewed Cega's code for security vulnerabilities, design issues, and general weaknesses in security posture.

## 1.1 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:

- Are users able to withdraw more than they are entitled?
- Can deposits or withdrawals be manipulated to unfairly reward or penalize users?
- Is it possible for any role to lock user funds in the contract?

## 1.2 Non-goals and Limitations

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

- Problems relating to the front-end components and infrastructure of the project
- Problems due to improper key custody or off-chain access control
- Issues stemming from code or infrastructure outside of the assessment scope

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

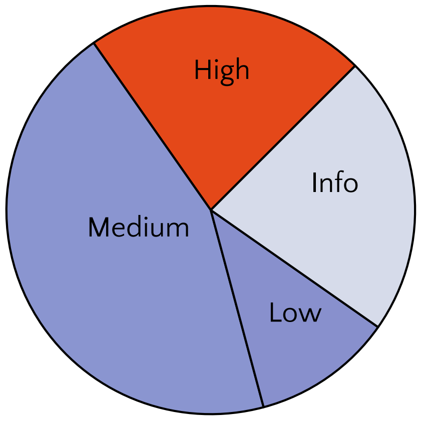
## 1.3 Results

During our assessment on the scoped Cega contracts, we discovered nine findings. No critical issues were found. Of the nine findings, two were of high impact, four were of medium impact, one was of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Sanic Pte. Ltd.'s benefit in the Discussion section (4) at the end of the document.

## Breakdown of Finding Impacts

| Impact Level | Count |
|---|---|
| Critical | 0 |
| High | 2 |
| Medium | 4 |
| Low | 1 |
| Informational | 2 |



# 2 Introduction

## 2.1 About Cega

Cega is a decentralized exotic derivatives protocol. They build exotic options–structured products for retail investors that generate superior yield and offer built-in protection against market downturns. Cega is developing new capabilities in tech, token contracts, and data modeling that will enable the next evolution of De-Fi derivatives.

## 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 the contracts' 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 code base 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, upgradeability 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:

### Cega Contracts

| | |
|---|---|
| **Repository** | https://github.com/cega-fi/cega-eth-v1 |
| **Versions** | `e0ffc80571f97b74a903627cf43e1efe86bbfc29` |
| **Programs** | • FCNProduct<br>• Oracle<br>• FCNVault<br>• CegaState |
| **Type** | Solidity |
| **Platform** | EVM-compatible |

## 2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of six person-days. The assessment was conducted over the course of three calendar days.

### Contact Information

The following project managers were associated with the engagement:

**Chad McDonald**, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

**Syed Faraz Abrar**, Engineer
faith@zellic.io

**Aaron Esau**, Engineer
aaron@zellic.io

## 2.5 Project Timeline

The key dates of the engagement are detailed below.

| | |
|---|---|
| **February 13, 2023** | Kick-off call |
| **February 13, 2023** | Start of primary review period |
| **February 15, 2023** | End of primary review period |

# 3 Detailed Findings

## 3.1 Missing valid vault address check in `processDepositQueue`

- **Target**: FCNProduct
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: High

### Description

Investors are able to deposit assets into an `FCNVault` through the `FCNProduct` contract's `addToDepositQueue()` function. This function pulls funds from the investor's wallet and adds `Deposit` objects into a global `depositQueue` array within the `FCNProduct` contract.

Subsequently, a trader admin is able to call `processDepositQueue()` to process these `Deposit` objects inside the `depositQueue`. On a high level, the `processDepositQueue()` function does the following:

1. Loops over the `depositQueue` a maximum of `maxProcessCount` times, or until it is empty.
2. For each deposit, it tracks the amount being deposited in the vault's metadata storage, accessed using the passed-in `vaultAddress`.
3. It calls the vault's `deposit()` function with the deposit amount and receiver address. This will send share tokens to the receiver.
4. If the `depositQueue` is empty afterwards, it will delete the queue.
5. Otherwise, it will shift over all remaining deposits in the queue to the beginning of the queue.

Now, there are only two checks in `processDepositQueue()` that are used to determine whether the `vaultAddress` that is passed corresponds to a valid, usable vault. They are as follows:

```
function processDepositQueue(address vaultAddress,
    uint256 maxProcessCount) public onlyTraderAdmin {
    FCNVaultMetadata storage vaultMetadata = vaults[vaultAddress];
    require(vaultMetadata.vaultStatus == VaultStatus.DepositsOpen,
    "500:WS");
    FCNVault vault = FCNVault(vaultAddress);
    require(!(vaultMetadata.underlyingAmount == 0 && vault.totalSupply()
    > 0), "500:Z");

    // [...]
}
```

These two checks are not enough. For example, if the trader admin deploys a malicious vault contract, then they can bypass both checks by doing the following:

1. Calling `openVaultDeposits()` with the address of their malicious vault contract.
2. Ensuring that their malicious vault contract contains a `totalSupply()` function that returns a value greater than zero.

```
function openVaultDeposits(address vaultAddress) public onlyTraderAdmin {
    FCNVaultMetadata storage vaultMetadata = vaults[vaultAddress];
    vaultMetadata.vaultStatus = VaultStatus.DepositsOpen;
}
```

After this is done, both of the checks will pass, and the code will treat the malicious vault as a valid `FCNVault` contract.

### Impact

A malicious trader admin can steal investors' funds using the following steps. The funds being stolen here come out of funds that are currently awaiting to be deposited.

1. Set up a fake malicious vault contract as described in the previous section with an empty `deposit()` function and a maliciously crafted `redeem()` function (see further below).
2. Wait for investors to add deposits into the `depositQueue`.
3. Call `processDepositQueue()` with the malicious vault address as many times as needed to process all deposits in the queue. This sets the vault's status to `NotTraded`.
4. Call `setTradeData()` with the `_tradeExpiry` set to a time in the past.

5. Call `sendAssetsToTrade()` to send the deposited assets to the market maker. This sets the vault's status to `Traded`.

6. Call `calculateCurrentYield()` with the malicious vault address. This will set the vault's status to `TradeExpired`.

7. Call `calculateVaultFinalPayoff()` with the malicious vault address. This will set the vault's status to `PayoffCalculated`.

8. Call `collectFees()` with the malicious vault address. This will set the vault's status to `FeesCollected`.

9. Queue a withdrawal to a trader admin–controlled wallet address using the `addToWithdrawalQueue()` function. Any `amountShares` is fine here.

10. Call `processWithdrawalQueue()`. This function ends up calling the vault's `redeem()` function to determine how many assets to return to the receiver of the withdrawal.

11. Since this is the Trader Admin's malicious vault contract, all they need to do is ensure that the malicious `redeem()` function returns `balanceOf(address(fcnProduct))` for themselves and 0 for all other receivers.

After the final step, all asset tokens in the `FCNProduct` contract will be transferred out to the wallet address specified in step 9.

## Recommendations

In `receiveAssetsFromCega()`, we see the following:

```
function receiveAssetsFromCegaState(address vaultAddress, uint256 amount)
    public {
    require(msg.sender == address(cegaState), "403:CS");
    FCNVaultMetadata storage vaultMetadata = vaults[vaultAddress];
    // a valid vaultAddress will never have vaultStart = 0
    require(vaultMetadata.vaultStart ≠ 0, "400:VA");

    IERC20(asset).safeTransferFrom(msg.sender, address(this), amount);
    vaultMetadata.currentAssetAmount += amount;
}
```

This same check should be added to `processDepositQueue()` and all other places where a `vaultAddress` is passed in as an argument. This will prevent invalid vault contract addresses from being used in the contract.

## Remediation

The client has acknowledged and remediated this issue by adding an `onlyValidVault` modifier that guarantees that the `vaultAddress` argument passed to all required functions is valid. This was done in commit f64513a9.

## 3.2 A malicious or compromised trader admin may lead to locked funds

- **Target**: FCNProduct
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: High

### Description

Investors in the Cega protocol use the FCNProduct contract's `addToDepositQueue()` function to deposit their funds. This function uses `safeTransferFrom()` to transfer asset tokens from the investor's address to the FCNProduct contract.

```
function addToDepositQueue(uint256 amount, address receiver) public {
    require(isDepositQueueOpen, "500:NotOpen");
    queuedDepositsCount += 1;
    queuedDepositsTotalAmount += amount;
    require(queuedDepositsTotalAmount + sumVaultUnderlyingAmounts
    ≤ maxDepositAmountLimit, "500:TooBig");

    IERC20(asset).safeTransferFrom(receiver, address(this), amount);
    depositQueue.push(Deposit({ amount: amount, receiver: receiver }));
    emit DepositQueued(receiver, amount);
}
```

Once these funds are deposited, the only way for the funds to leave the contract are through the following functions:

1. `collectFees()` - Only callable by the trader admin. Used to collect fees for the Cega protocol.
2. `processWithdrawalQueue()` - Only callable by the trader admin. Used to process investor withdrawals.
3. `sendAssetsToTrade()` - Only callable by the trader admin. Used to send deposited assets to a market maker.

As there are no other ways to take deposited funds out of the contract, a malicious or compromised trader admin may choose simply to not call any of these functions. If this were to happen, any deposited investor funds (and any other funds in the contract) will become locked in the contract forever.

## Impact

A compromised or malicious trader admin can lead to funds being locked in the `FCNP roduct` contract forever.

## Recommendations

Consider adding a sweep-style function that allows the protocol to transfer out any tokens in the contract to a chosen address. Ideally, this function should only be accessible by the default admin multi-sig role.

```
function sweepTokens(address receiver) external onlyDefaultAdmin {
    IERC20(asset).safeTransfer(receiver,
    IERC20(asset).balanceOf(address(this)));
}
```

## Remediation

The client has acknowledged this issue, and has stated that it is mitigated due to the fact that the `DefaultAdmin` role can assign a new `TraderAdmin` through the `CegaState` contract.

## 3.3 The `vaultAddress` validity check can be bypassed

- **Target**: FCNProduct
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: Medium

### Description

In the `receiveAssetsFromCegaState()` function, the following code is used to determine whether the `vaultAddress` passed to it corresponds to a valid vault:

```
function receiveAssetsFromCegaState(address vaultAddress, uint256 amount)
    public {
    require(msg.sender == address(cegaState), "403:CS");
    FCNVaultMetadata storage vaultMetadata = vaults[vaultAddress];
    // a valid vaultAddress will never have vaultStart = 0
    require(vaultMetadata.vaultStart ≠ 0, "400:VA");

    IERC20(asset).safeTransferFrom(msg.sender, address(this), amount);
    vaultMetadata.currentAssetAmount += amount;
}
```

This check looks correct at first glance because the only way to get a vault metadata's `vaultStart` property set is through the `createVault()` function, which always creates an instance of an `FCNVault` contract:

```
function createVault(
    string memory _tokenName,
    string memory _tokenSymbol,
    uint256 _vaultStart
) public onlyTraderAdmin returns (address vaultAddress) {
    require(_vaultStart ≠ 0, "400:VS");
    FCNVault vault = new FCNVault(asset, _tokenName, _tokenSymbol);
    address newVaultAddress = address(vault);
    vaultAddresses.push(newVaultAddress);

    // vaultMetadata & all of its fields are automatically initialized if
    it doesn't already exist in the mapping
    FCNVaultMetadata storage vaultMetadata = vaults[newVaultAddress];
    vaultMetadata.vaultStart = _vaultStart;
    vaultMetadata.vaultAddress = newVaultAddress;
    emit VaultCreated(newVaultAddress, vaultAddresses.length - 1);
    return newVaultAddress;
}
```

However, the `rolloverVault()` function can allow a malicious or compromised trader admin to bypass this check. The `rolloverVault()` function is missing a check to ensure that the `vaultAddress` passed to it is valid:

```
function rolloverVault(address vaultAddress) public onlyTraderAdmin {
    FCNVaultMetadata storage vaultMetadata = vaults[vaultAddress];
    require(vaultMetadata.vaultStatus
    == VaultStatus.WithdrawalQueueProcessed, "500:WS");
    require(vaultMetadata.tradeExpiry ≠ 0, "400:TE");
    vaultMetadata.vaultStart = vaultMetadata.tradeExpiry;
    // [ ... ]
}
```

This can be used to set an arbitrary address's `vaultStart` metadata property to a non-zero value, which would bypass the `vaultAddress` validity check.

## Impact

A malicious or compromised trader admin can cause the vault metadata of an arbitrary address to look like a valid `FCNVault`.

First, a malicious vault contract must be created. It must contain the following functions:

1. A `totalSupply()` function that returns a value greater than zero.
2. An empty `deposit()` function.
3. An empty `redeem()` function.

Then, the malicious or compromised trader admin can take the following steps to set the malicious vault contract's `vaultStart` metadata property to a non-zero value.

1. Call `openVaultDeposits()` with the malicious vault address. This will set the vault's status to `DepositsOpen`.

2. Call `processDepositQueue()` with the malicious vault address. This will set the vault's status to `NotTraded`.

3. Call `setTradeData()` with the `_tradeExpiry` set to a non-zero value, such that it is set to a time in the past.

4. Call `sendAssetsToTrade()` with the `amount` set to 0. This sets the vault's status to `Traded`.

5. Call `calculateCurrentYield()` with the malicious vault address. This will set the vault's status to `TradeExpired`.

6. Call `calculateVaultFinalPayoff()` with the malicious vault address. This will set the vault's status to `PayoffCalculated`.

7. Call `collectFees()` with the malicious vault address. This will set the vault's status to `FeesCollected`.

8. Call `processWithdrawalQueue()` with the malicious vault address. The withdrawal queue is empty, so this will just set the vault's status to `WithdrawalQueueProcessed`.

9. Call `rolloverVault()` with the malicious vault address. Both the `require` statements in the function will pass, and the `vaultStart` metadata property will be set to the `_tradeExpiry` value from step 3.

## Recommendations

Add the following check to `rolloverVault()`:

```
function rolloverVault(address vaultAddress) public onlyTraderAdmin {
    FCNVaultMetadata storage vaultMetadata = vaults[vaultAddress];
    require(vaultMetadata.vaultStart ≠ 0); // Add this check

    // [ ... ]
}
```

## Remediation

The client has acknowledged and fixed this issue by adding a vault address validity check to `rolloverVault()`. This was fixed in commit f64513a9.

## 3.4 Ability to deposit on other users' behalf

- **Target**: FCNProduct
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: Medium

### Description

When a user calls addToDepositQueue(), they are required to pass the address of a receiver as the second argument. The function pulls an amount of asset tokens from the receiver via the use of safeTransferFrom():

```
function addToDepositQueue(uint256 amount, address receiver) public {
    require(isDepositQueueOpen, "500:NotOpen");
    queuedDepositsCount += 1;
    queuedDepositsTotalAmount += amount;
    require(queuedDepositsTotalAmount + sumVaultUnderlyingAmounts
    ≤ maxDepositAmountLimit, "500:TooBig");

    IERC20(asset).safeTransferFrom(receiver, address(this), amount);
    depositQueue.push(Deposit({ amount: amount, receiver: receiver }));
    emit DepositQueued(receiver, amount);
}
```

This implies that the receiver must preapprove the FCNProduct contract, as the safeTransferFrom() will revert otherwise. Generally, the approval amount is set to the maximum uint256 value. This introduces a vector through which an attacker can deposit more assets on behalf of the receiver at a later point in time.

### Impact

Consider the following scenario:

1. The victim decides they want to invest 1,000 USDC into an FCN product.
2. The victim max approves the FCNProduct contract and uses addToDepositQueue() to invest 1,000 USDC. They have no intention of investing more than this amount.
3. Some amount of time later, the attacker notices that the victim's wallet has been transferred 50,000 USDC from elsewhere.

4. The victim plans to use this USDC for other things, but the attacker now calls `addToDepositQueue()` with `receiver` set to the victim's address.
5. Since the victim has already approved the `FCNProduct` contract, this deposit will go through, and now the victim is at risk of losing a part of this money.

The impact here is that the victim is griefed by the attacker. The attacker may or may not benefit from depositing funds on the victim's behalf, but the victim now stands to lose this money if, for example, the vault experiences a knock in event (a downside protection for user-deposited capital). There is also no way for the victim to cancel their deposit while it is in the deposit queue.

We have noted that `addToWithdrawalQueue()` uses a similar pattern. However, we do not believe a similar attack vector exists there.

### Recommendations

We recommend removing the use of the `receiver` argument and instead pulling funds directly from `msg.sender`.

### Remediation

The client has acknowledged and fixed this issue by removing the `receiver` parameter and using `msg.sender` instead. This was fixed in commit 4a95e773.

## 3.5 Missing status check in `openVaultDeposits`

- **Target**: FCNProduct
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Medium

### Description

Before the `depositQueue` can be processed for a specific vault, that vault's status needs to be set to `DepositsOpen`. This can be achieved using the `openVaultDeposits()` function:

```
function openVaultDeposits(address vaultAddress) public onlyTraderAdmin {
    FCNVaultMetadata storage vaultMetadata = vaults[vaultAddress];
    vaultMetadata.vaultStatus = VaultStatus.DepositsOpen;
}
```

This function does not check to ensure the vault is in the initial `DepositsClosed` status. A trader admin may accidentally, or through malicious intent, modify the status of any vault to `DepositsOpen` at any time from any arbitrary status.

### Impact

The vaults are designed to go through specific states in a certain order. If this order is not followed, the vault may end up in an unintended status, which could lead to any number of problems (e.g., the vault not functioning as intended).

### Recommendations

Add a preconditional status check to `openVaultDeposits()` to ensure that the vault is in a `DepositsClosed` status.

### Remediation

The client has acknowledged and fixed this issue by adding a state check to `openVaultDeposits()`. This was fixed in commit 455ab74c.

## 3.6 Missing sanity checks for crucial protocol parameters

- **Target**: FCNProduct
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: Medium

### Description

The Cega smart contracts rely on a number of protocol parameters to function correctly. There are functions that allow the admins of the protocol to alter most of these parameters.

We found that the majority of these parameters are not checked to be within certain limits before they are set. The majority of these setter functions are only accessible by either the operator admin or the trader admin, both of which are non-multi-sig wallets.

In particular, the following functions are only accessible by either the operator admin or the trader admin. They are missing sanity checks on crucial protocol parameters before they are set:

1. `setManagementFeeBps()` - If set to 100%, it could lead to all investor funds being sent to the Cega fee recipient. Only accessible by the operator admin.
2. `setYieldFeeBps()` - Similar to `setManagementFeeBps()`. Only accessible by the operator admin.
3. `setMaxDepositAmountLimit()` - If set to 0, it will prevent the `FCNProduct` contract from accepting deposits, leading to denial of service. Only accessible by the trader admin.
4. `setTradeData()` - If the `tradeExpiry` parameter is set to hundreds of years in the future, funds would effectively be locked forever in the vault. Furthermore, the `aprBps` parameter can be set to a very high number. The trader admin can become an investor themselves and profit off of the high APR. Only accessible by the trader admin.
5. `updateOptionBarrierOracle()` and `addOracle()` - Allows full control over which oracle is used for a specific option barrier. Only accessible by the operator admin.
6. `addOptionBarrier()` and `updateOptionBarrier()` - If the `strikeAbsoluteValue` is set to 0, then a revert will occur when `calculateVaultFinalPayoff()` is called, as it will result in a division by 0 in `calculateKnockInRatio()`. Only accessible by the trader admin.

## Impact

If the trader admin or operator admin roles are ever compromised or turn malicious, they would be able to set crucial protocol parameters to arbitrary values. This would cause the smart contracts to function incorrectly, and it may lead to loss of protocol or investors' funds in the worst case.

Specifically, consider the `setTradeData()` function, which allows trader admins to modify vault-specific metadata parameters. This function does not check the validity of the parameters. So, if a trader admin were to set the `tradeExpiry` parameter to a non-zero value that is less than the `vaultStart` configured in the `createVault` function, the `collectFees()` function would not be callable (i.e., the trader admin would be locked out of collecting fees).

The `collectFees()` function internally calls the `calculateFees()` function, which has the following subtraction that would underflow:

```
function calculateFees(
    FCNVaultMetadata storage self,
    uint256 managementFeeBps,
    uint256 yieldFeeBps
) public view returns (uint256, uint256, uint256) {
    // [...]
    uint256 numberOfDaysPassed = (self.tradeExpiry - self.vaultStart)
    / SECONDS_TO_DAYS;
    // [...]
}
```

Note that these parameters can be overridden by the default admin role (which is intended to be controlled by a multi-sig) using the `setVaultMetadata()` function, and therefore the impact is partially mitigated.

## Recommendations

Add sanity checks to these functions to ensure the parameters are within sane limits.

## Remediation

The client has acknowledged and fixed this issue by adding the necessary sanity checks. This was fixed in commit a638c874.

## 3.7 Gas griefing using zero-value deposits and withdrawals

- **Target**: FCNProduct
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Low
- **Impact**: Low

### Description

Within the `FCNProduct` contract, users are able to submit deposits and withdrawals using the `addToDepositQueue()` and `addToWithdrawalQueue()` functions, respectively. Both these functions allow for zero-value deposits and withdrawals to be enqueued.

### Impact

The deposits and withdrawals are later processed by the `processDepositQueue()` and `processWithdrawalQueue()` functions, respectively. These functions are intended to be called by a trader admin, and they do not distinguish between zero-value and non-zero-value deposits and withdrawals. This means the same amount of gas will be used in both instances. An attacker that wants to grief the protocol into wasting gas can choose to add a lot of zero-value deposits or withdrawals. The only way to empty the queues are with the aforementioned processing functions; thus, the trader admin will be forced to waste gas on these zero-value deposits and withdrawals.

### Recommendations

Ensure that a user must deposit or withdraw a minimum amount of tokens.

### Remediation

The client has acknowledged and fixed this issue by setting a minimum deposit and withdrawal amount. This was fixed in commit 1944cc8f.

## 3.8 Missing checks and some access controls on critical functions

- **Target**: FCNProduct
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

### Description

As explained by Cega,

> The knock-in (KI) feature provides downside protection for investors' deposited capital. Specifically, investors will receive 100% of their initial investment in the FCN even if crypto asset prices are falling. In this case, unless crypto asset prices fall by 50% or more versus the day the vault started, investors' capital will be protected (unlike vanilla option strategies). If however the FCN does KI, the principal returned at expiry is equal to the lesser of 100% or the fallen asset price percentage of its initial price.

To determine if a knock-in event has occurred, Cega uses option barriers.

It is safe to assume that investors will choose their investments by carefully considering a few factors. One of these factors may be to check what the knock-in barrier level is set to in relation to the price volatility of the option asset tokens. For example, if the token price is highly volatile, and the knock-in barrier level is at 90%, then there is a high chance that a knock-in event will occur, which may cause the investor to decide against investing in that specific vault.

Currently, there exists three functions that the trader admin can use to add, update, and remove knock-in barriers. These are the `addOptionBarrier()`, `updateOptionBarrier()`, and `removeOptionBarrier()` functions, respectively. An important characteristic of these functions is that they do not require the vault to be in any specific state, meaning the trader admin can add or update option barriers at any time, even after the investor's deposits are locked in.

Furthermore, the parameters of a knock-in barrier can be arbitrary, as there are no sanity checks to ensure they are within certain limits.

```
function addOptionBarrier(address vaultAddress,
    OptionBarrier calldata optionBarrier) public onlyTraderAdmin {
    FCNVaultMetadata storage metadata = vaults[vaultAddress];
    metadata.optionBarriers.push(optionBarrier);
    metadata.optionBarriersCount++;
}
```

This will reduce the investor's trust in the protocol because although they might note a very low knock-in barrier level initially (i.e., a low chance of a knock-in event occurring), they will know that the level may be raised at any moment, which makes the investment inherently risky.

There also exists a `setKnockInStatus()` function that the trader admin can use to arbitrarily set a vault's knock-in status to `true`.

Finally, the trader admin can also use the oracle's `updateRoundData()` function to arbitrarily control the option asset token price returned by the oracle. This could also be used to trigger a knock-in event.

### Impact

Investors' trust in the protocol is significantly reduced due to missing checks and incorrect access controls in critical state-modifying functions.

### Recommendations

For the option barrier functionality, consider requiring a vault state of `VaultStatus.NotTraded` to modify any option barriers in the vault.

For the `setKnockInStatus()` function, consider removing it completely. Alternatively, place it behind the `onlyDefaultAdmin` modifier instead.

For the `updateRoundData()` function, consider changing its access control such that only the default admin multi-sig or the operator admin role can call it.

### Remediation

The client has acknowledged and fixed all of the above issues according to our recommendations. This was fixed in commit 834fe7ed.

## 3.9 Unused or unreachable functionality should be removed

- **Target**: FCNVault, FCNProduct
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

### Description

The FCNVault contract contains three `withdraw()` functions that are not used.

The FCNVault contract also contains three `redeem()` functions. Two of the `redeem()` functions are wrappers. These are shown below:

```
function redeem(uint256 shares, address receiver, address owner)
    public onlyOwner returns (uint256) {
    uint256 assets = convertToAssets(shares);
    if (owner ≠ msg.sender) {
        _spendAllowance(owner, msg.sender, shares);
    }
    _burn(owner, shares);
    emit Withdraw(msg.sender, receiver, owner, assets, shares);
    return assets;
}

function redeem(uint256 shares, address receiver)
    external onlyOwner returns (uint256) {
    return redeem(shares, receiver, msg.sender);
}

function redeem(uint256 shares) external onlyOwner returns (uint256) {
    return redeem(shares, msg.sender, msg.sender);
}
```

The last `redeem()` function is the only one that is called by the FCNProduct contract. This will result in a final call to the first `redeem()` function with the `owner` parameter set to `msg.sender`. This makes the `owner` $\neq$ `msg.sender` if block in the first `redeem()` function unnecessary.

The `FCNProduct` contract contains a function called `throwError()`, which is not used. The contract itself also defines an error named `FCNProductError`, which is unused.

## Impact

Unused or unreachable functionality adds to the complexity of the codebase, which might lead to programmer error in the future.

## Recommendations

The `withdraw()` functions should be removed unless there are future plans to use them in the codebase.

The `redeem()` function should have the `owner ≠ msg.sender` if block removed, as the condition will never be true.

The `throwError()` function and the corresponding `FCNProductError` should be removed unless there are future plans to use it.

## Remediation

The client has remediated this issue by removing the use of the unused `FCNProductError` and `throwError()` function in commit 0bb7eb6b.

Unused functionality was removed from FCNVault in commit f4a93124.

# 4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

## 4.1 Centralization risks

There exists four key admin roles in the Cega protocol:

1. Default admin
2. Operator admin
3. Trader admin
4. Service admin

Of these, the default admin role is the only multi-sig.

All core functionality in the `FCNProduct` contract relies on the security and trustworthiness of all the admin roles. However, a majority of this trust is placed on the trader admin, which is a non-multi-sig role. This should be clearly disclosed to the investors to allow them to make an informed decision about the contracts they interact with.

## 4.2 Using non-ERC20 token standards

Note that - while Cega is intended to be used with ERC20 tokens only - it is possible to use Cega with tokens of other standards that share the same ERC20 function selectors (those that Cega use, at least). However, using non-ERC20 tokens is dangerous because Cega is designed to work with ERC20 tokens only.

For example, it is possible to create an `FCNProduct` contract configured to use an ERC721 asset. But `IERC721.transferFrom` triggers a call to the `to` address's `onERC721Received` function, potentially enabling reentrancy exploits. An admin may be able to abuse this behavior if a vault were configured with an ERC721 token to reenter into `collectFees` and drain the vault.

## 4.3 Impossibility of reentrancy in the queue-processing functions

Though the `processDepositQueue()` and `processWithdrawalQueue()` functions appear to allow reentrancy using the `vault.deposit( ... )` and `vault.redeem( ... )` external calls, respectively, the root cause of this issue is that vault addresses are unchecked in many functions — which is already covered in finding 3.1.

Cega added ReentrancyGuard to the FCNProduct contract in commit d73733fb

# 5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

## 5.1 Module: CegaState.sol

### Function: `moveAssetsToProduct(string productName, address vaultAddress, uint256 amount)`

This function moves `amount` of the asset corresponding to the vault identified by `productName` to the vault from this contract.

### Inputs

- `productName`
  - **Control**: Full.
  - **Constraints**: String must exist in `products` mapping.
  - **Impact**: This is the contract the vault is looked up in to transfer tokens to.
- `vaultAddress`
  - **Control**: Full.
  - **Constraints**: In the `fcnProduct.receiveAssetsFromCegaState` external call, the `vaultStart` of the `FCNVaultMetadata` struct corresponding to `vaultAddress` is checked to be greater than zero; that is, the `vaultAddress` must be an existing vault in the `fcnProduct`.
  - **Impact**: This is the address that receives the tokens.
- `amount`
  - **Control**: Full.
  - **Constraints**: The token balance of this contract must be greater than or equal to the `amount` value or the approval fails.
  - **Impact**: This is the amount of token to transfer to the `vaultAddress`.

### Branches and code coverage (including function calls)

**Intended branches**

- Assets are properly transferred to the `vaultAddress`.
  - ☑ Test coverage

**Negative behaviour**

- Address `productAddress` cannot be zero (i.e., `productName` must be valid).
  - ☑ Negative test
- Contract `CegaState` must have enough funds.
  - ☑ Negative test
- Address `vaultAddress` must be valid, as determined by the `fcnProduct` in the external call to `fcnProduct.receiveAssetsFromCegaState`.
  - ☑ Negative test

### Function call analysis

- `moveAssetsToProduct` → `receiveAssetsFromCegaState(vaultAddress, amount)`
  - **What is controllable?**: vaultAddress, amount.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Nothing. The function follows the CEI pattern.

## 5.2 Module: FCNProduct.sol

### Function: `addOptionBarrier(address vaultAddress, OptionBarrier optionBarrier)`

Adds a new option barrier to the vault metadata's `optionBarriers` array.

### Inputs

- `vaultAddress`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: Updates this vault's metadata's `optionBarriers` array.
- `optionBarrier`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: Inserted into the vault metadata's `optionBarriers` array.

### Branches and code coverage (including function calls)

**Intended branches**

- Should update the vault's metadata correctly.
  ☑ Test coverage

**Negative behaviour**

- Should revert if called by a non–trader-admin role.
  ☑ Negative test

### Function: `addToDepositQueue(uint256 amount, address receiver)`

Adds a deposit to the `depositQueue` storage array.

### Inputs

- `amount`
  - **Control**: Fully controlled.
  - **Constraints**: The `receiver` must own at least this amount of `asset` tokens.
  - **Impact**: This amount of tokens are transferred from the `receiver` to this contract.
- `receiver`
  - **Control**: Fully controlled.
  - **Constraints**: Must have preapproved this contract.
  - **Impact**: Tokens are transferred from this address to this contract.

### Branches and code coverage (including function calls)

**Intended branches**

- Should update the count of queued deposits.
  ☑ Test coverage
- Should update the total queued deposits amount.
  ☑ Test coverage
- Should add a new deposit into the `depositQueue` storage array.
  ☑ Test coverage
- Should emit a `DepositQueued` event.
  ☑ Test coverage
- Should decrease the asset balance of the `receiver`.
  ☑ Test coverage
- Should increase the asset balance of this contract.
  ☑ Test coverage

**Negative behaviour**

- Should revert if the deposit queue is not open.
  - ☑ Negative test
- Should revert if this deposit would cause the product to go over its maximum deposit amount limit.
  - ☑ Negative test
- Should revert if the `receiver` has not approved this contract to spend the required amount.
  - ☑ Negative test

### Function call analysis

- `IERC20(asset).safeTransferFrom(receiver, address(this), amount)`
  - **What is controllable?** receiver, amount.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the deposit will fail. No other side effects as it follows the CEI pattern.

## Function: `addToWithdrawalQueue(address vaultAddress, uint256 amountShares, address receiver)`

Adds a vault withdrawal action to the `withdrawalQueue` storage array.

### Inputs

- `vaultAddress`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The withdrawal is queued for this specific vault.
- `amountShares`
  - **Control**: Fully controlled.
  - **Constraints**: User must own at least this amount of shares.
  - **Impact**: This amount of shares are transferred out of the receiver's wallet.
- `receiver`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The shares are transferred out of this address.

### Branches and code coverage (including function calls)

**Intended branches**

- Should update the vault's metadata.
  ☑ Test coverage
- Should increase this contract's share token balance.
  ☑ Test coverage

**Negative behaviour**

- Should revert if the user has not approved this contract.
  ☑ Negative test

### Function call analysis

- `IERC20(asset).safeTransferFrom(receiver, address(this), amountShares)`
  - **What is controllable?** receiver, amountShares.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the share transfer will fail. No other side effects as it follows the CEI pattern.

## Function: `calculateCurrentYield(address vaultAddress)`

Updates the metadata of the vault specified by `vaultAddress` to account for any yield accumulated up until the current day. The accumulated amount is stored in the metadata's `totalCouponPayoff` property. If the trade has already expired, then the vault's status is set to `TradeExpired`.

### Inputs

- `vaultAddress`
  - **Control**: Fully controlled.
  - **Constraints**: Vault's status must be set to `Traded`.
  - **Impact**: Updates this vault's metadata.

### Branches and code coverage (including function calls)

**Intended branches**

- Should set `vaultStatus` to `TradeExpired` if time is past trade expiry.
  ☑ Test coverage
- Should calculate the `totalCouponPayoff` correctly.
  ☑ Test coverage
- Should not calculate `totalCouponPayoff` if the trade has already expired.
  ☑ Test coverage

**Negative behaviour**

- Must revert if the `vaultStatus` is not set to `Traded`.
  ☑ Negative test

### Function: `calculateVaultFinalPayoff(address vaultAddress)`

Calculates the final payoff for a given vault.

### Inputs

- `vaultAddress`
  - **Control**: Fully controlled.
  - **Constraints**: The vault must have a state of `TradeExpired` or `PayoffCalculated`.
  - **Impact**: The vault's state and metadata may potentially be updated.

### Branches and code coverage (including function calls)

**Intended branches**

- Should calculate the correct payoff when a knock-in event has not occurred.
  ☑ Test coverage
- Should calculate the correct payoff when a knock-in event has occurred.
  ☑ Test coverage

**Negative behaviour**

- Should revert if the vault is in an invalid state.
  ☑ Negative test

### Function: `checkBarriers(address vaultAddress)`

Checks if a knock-in event occurred due to a drop in an option barrier token's price.

### Inputs

- `vaultAddress`
  - **Control**: Fully controlled.
  - **Constraints**: Vault must have a status of `Traded`.
  - **Impact**: Vault metadata might potentially be updated to account for knock-in status.

### Branches and code coverage (including function calls)

**Intended branches**

- Should account for knock-in state correctly.
  ☑ Test coverage

**Negative behaviour**

- Should revert if the vault state is not `Traded`.
  ☑ Negative test
- Should fail if an oracle does not exist for this option barrier's token.
  ☑ Negative test

### Function: `createVault(string _tokenName, string _tokenSymbol, uint256 _vaultStart)`

Creates a new instance of an FCNVault and stores associated metadata for it inside the `vaults[]` mapping. The mapping is keyed by the new FCNVault contract's address.

### Inputs

- `_tokenName`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: Passed to the constructor of the new FCNVault contract.
- `_tokenSymbol`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: Passed to the constructor of the new FCNVault contract.
- `_vaultStart`
  - **Control**: Fully controlled.
  - **Constraints**: Must not be equal to 0.
  - **Impact**: Creates a new FCNVault and sets the vault's start time to this value.

### Branches and code coverage (including function calls)

**Intended branches**

- Should add the newly created vault's address to the `vaults[]` mapping.
  ☑ Test coverage
- Should create a new FCNVault and track its address in storage.
  ☑ Test coverage

- Should emit the `VaultCreated` event.
  - ☑ Test coverage

**Negative behaviour**

- Should revert if `_vaultStart` is set to 0.
  - ☑ Negative test
- Should revert if called by a non–trader-admin role.
  - ☑ Negative test

### Function: `openVaultDeposits(address vaultAddress)`

Opens a vault for deposits.

### Inputs

- `vaultAddress`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This vault's status is set to `DepositsOpened`.

### Branches and code coverage (including function calls)

**Intended branches**

- Should update the vault's status to `DepositsOpened`.
  - ☑ Test coverage

**Negative behaviour**

- Should revert if called by a non–trader-admin role.
  - ☑ Negative test

### Function: `processDepositQueue(address vaultAddress, uint256 maxProcessCount)`

Processes deposits that are currently in the deposit queue.

### Inputs

- `vaultAddress`
  - **Control**: Fully controlled.
  - **Constraints**: This vault's status must be set to `DepositsOpen`.
  - **Impact**: Deposits are processed for this vault.

- `maxProcessCount`
  - **Control**: Fully controlled.
  - **Constraints**: Used as a loop counter, therefore must not cause the call to exceed gas limits.
  - **Impact**: Used to constrain the amount of gas used by this function.

## Branches and code coverage (including function calls)

### Intended branches

- Should update relevant `FCNProduct` storage variables correctly.
  - ☑ Test coverage
- Should update relevant vault metadata properties correctly.
  - ☑ Test coverage
- Should still allow for more deposits to be processed if the entire deposit queue is not processed after one call.
  - ☑ Test coverage
- Should set the vault's status to `NotTraded` if the entire deposit queue is processed.
  - ☑ Test coverage
- Should emit a `DepositQueueProcessed` event.
  - ☑ Test coverage

### Negative behaviour

- Should revert if the vault's status it not set to `DepositsOpen`.
  - ☑ Negative test
- Should revert if called by a non-trader-admin role.
  - ☑ Negative test
- Should revert if the vault is in the `Zombie` state.
  - ☑ Negative test

## Function call analysis

- `vault.deposit(deposit.amount, deposit.receiver)`
  - **What is controllable?** deposit.amount, deposit.receiver.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Denial of service on revert, as processing the deposit queue is crucial to the functionality of this contract. Deposits may be accounted for twice on reentry, and some deposits will not be accounted for at all in that scenario.

### Function: `processWithdrawalQueue(address vaultAddress, uint256 maxProcessCount)`

Processes all the queued withdrawals in the withdrawal queue.

#### Inputs

- `vaultAddress`
  - **Control**: Fully controlled.
  - **Constraints**: Vault must have a status of `FeesCollected` or `Zombie`.
  - **Impact**: This vault's metadata is updated.
- `maxProcessCount`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: Used to constrain the amount of gas used by this function.

#### Branches and code coverage (including function calls)

**Intended branches**

- Should update relevant `FCNProduct` storage variables correctly.
  ☑ Test coverage
- Should update relevant vault metadata properties correctly.
  ☑ Test coverage
- Should set the vault's status to `WithdrawalQueueProcessed` if the entire deposit queue is processed.
  ☑ Test coverage
- Should set the vault's status to `Zombie` if the specific preconditions are met.
  ☑ Test coverage
- Should emit a `WithdrawalQueueProcessed` event.
  ☑ Test coverage

**Negative behaviour**

- Should revert if the vault's status is not set to `FeesCollected` or `Zombie`.
  ☑ Negative test
- Should revert if called by a non-trader-admin role.
  ☑ Negative test

#### Function call analysis

- `vault.redeem(withdrawal.amountShares, withdrawal.receiver)`
  - **What is controllable?** withdrawal.amountShares, withdrawal.receiver.

  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Denial of service on revert, as processing the withdrawal queue is crucial to the functionality of this contract. Withdrawals may be accounted for twice on reentry, and some withdrawals will not be accounted for at all in that scenario.

### Function: `receiveAssetsFromCegaState(address vaultAddress, uint256 amount)`

Receive assets and allocate the underlying asset to the specified vault's balance.

#### Inputs

- `vaultAddress`
  - **Control**: Fully controlled.
  - **Constraints**: The CegaState contract that is calling this function must have the required funds.
  - **Impact**: The funds are transferred to this contract.
- `amount`
  - **Control**: Fully controlled.
  - **Constraints**: The CegaState contract must own at least this amount of funds.
  - **Impact**: This amount is transferred over to this contract.

#### Branches and code coverage (including function calls)

**Intended branches**

- The balance of this contract is updated successfully.
  ☑ Test coverage
- The vault's metadata is updated to account for the new funds.
  ☑ Test coverage

**Negative behaviour**

- Should revert if not called by the CegaState contract.
  ☑ Negative test
- Should revert if an invalid vault address is used.
  ☑ Negative test
- Should revert if the caller does not have the required funds.
  ☑ Negative test

### Function: `removeOptionBarrier(address vaultAddress, uint256 index, string _asset)`

Removes an option barrier from a vault metadata's `optionBarriers` array and shifts all other elements down.

#### Inputs

- `vaultAddress`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This vault's metadata is accessed and updated.
- `index`
  - **Control**: Fully controlled.
  - **Constraints**: Used to index into the vault metadata's `optionBarriers` array.
  - **Impact**: N/A.
- `_asset`
  - **Control**: Fully controlled.
  - **Constraints**: Must match the `asset` of the option barrier being removed.
  - **Impact**: N/A.

#### Branches and code coverage (including function calls)

**Intended branches**

- Should remove the indexed option barrier and shift all other option barriers down.
  - ☑ Test coverage

**Negative behaviour**

- Should revert if called by a non–trader-admin role.
  - ☑ Negative test
- Should revert if the `index` argument is out of bounds of the vault metadata's `optionBarriers` array.
  - ☑ Negative test
- Should revert if the `_asset` argument does not match the `asset` stored in the indexed option barrier.
  - ☑ Negative test

### Function: `removeVault(address vaultAddress)`

Removes a vault from the `vaultAddresses` array and deletes its metadata.

### Inputs

- `vaultAddress`
  - **Control**: Fully controlled.
  - **Constraints**: Must exist in the `vaultAddresses` array.
  - **Impact**: This address is removed from the `vaultAddresses` array.

### Branches and code coverage (including function calls)

**Intended branches**

- Should remove the `vaultAddress` from the `vaultAddresses` storage array.
  - ☑ Test coverage
- Should emit the `VaultRemoved` event.
  - ☑ Test coverage

**Negative behaviour**

- Should revert if called by a non-default-admin role.
  - ☑ Negative test
- Should revert if the `vaultAddress` does not exist in the `vaultAddresses` storage array.
  - ☑ Negative test

### Function: `rolloverVault(address vaultAddress)`

Resets the vault to the default state after the trade is settled.

### Inputs

- `vaultAddress`
  - **Control**: Fully controlled.
  - **Constraints**: The vault metadata's `vaultStart` property must not be 0, and it must have a state of `WithdrawalQueueProcessed`.
  - **Impact**: This vault's metadata is reset.

### Branches and code coverage (including function calls)

**Intended branches**

- Should successfully roll over the vault.
  - ☑ Test coverage
- Should emit a `RolloverVault` event.
  - ☑ Test coverage

**Negative behaviour**

- Should revert if not called by a trader admin.
  - ☑ Negative test
- Should revert if the vault status is not set to `WithdrawalQueueProcessed`.
  - ☑ Negative test
- Should revert if the vault's `tradeExpiry` is set to 0.
  - ☐ Negative test

### Function: `setIsDepositQueueOpen(bool _isDepositQueueOpen)`

Opens or closes the deposit queue. Only callable by the operator admin role.

### Inputs

- `_isDepositQueueOpen`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: Modifies the `isDepositQueueOpen` storage variable.

### Branches and code coverage (including function calls)

**Intended branches**

- Should update the `isDepositQueueOpen` storage variable correctly.
  - ☑ Test coverage

**Negative behaviour**

- Should revert if called by a non–operator-admin role.
  - ☑ Negative test

### Function: `setKnockInStatus(address vaultAddress, bool newState)`

Sets a vault's knock-in status.

### Inputs

- `vaultAddress`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This vault's knock-in status is updated.
- `newState`
  - **Control**: Fully controlled.

  - **Constraints**: N/A.
  - **Impact**: This is the new knock-in status.

### Branches and code coverage (including function calls)

**Intended branches**

- Should set the knock-in status correctly.
  ☑ Test coverage

**Negative behaviour**

- Should revert if called by a non–trader-admin role.
  ☑ Negative test

### Function: `setManagementFeeBps(uint256 _managementFeeBps)`

This function sets the global management fee percentage for this FCNProduct. Only callable by the operator admin role.

### Inputs

- `_managementFeeBps`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: Modifies the `managementFeeBps` storage variable.

### Branches and code coverage (including function calls)

**Intended branches**

- Modifies the `managementFeeBps` storage variable correctly.
  ☑ Test coverage

**Negative behaviour**

- Should revert if called by a non–operator-admin role.
  ☑ Negative test

### Function: `setMaxDepositAmountLimit(uint256 _maxDepositAmountLimit)`

Sets the `maxDepositAmountLimit` storage variable. Only callable by the trader admin role.

### Inputs

- `_maxDepositAmountLimit`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: Updates the `maxDepositAmountLimit` storage variable.

### Branches and code coverage (including function calls)

**Intended branches**

- Should update the `maxDepositAmountLimit` storage variable correctly.
  ☑ Test coverage

**Negative behaviour**

- Should revert if called by a non–trader-admin role.
  ☑ Negative test

### Function: `setTradeData(address vaultAddress, uint256 _tradeDate, uint256 _tradeExpiry, uint256 _aprBps, uint256 _tenorInDays)`

Sets a vault's trade-related metadata.

### Inputs

- `vaultAddress`
  - **Control**: Fully controlled.
  - **Constraints**: The vault's status must be set to `NotTraded`.
  - **Impact**: This vault's metadata is updated.
- `_tradeDate`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: Used to update the vault's metadata.
- `_tradeExpiry`
  - **Control**: Fully controlled.
  - **Constraints**: Must not be set to 0.
  - **Impact**: Used to update the vault's metadata.
- `_aprBps`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: Used to update the vault's metadata.

- `_tenorInDays`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: Used to update the vault's metadata.

### Branches and code coverage (including function calls)

#### Intended branches

- Should update the vault's metadata correctly.
  ☑ Test coverage

#### Negative behaviour

- Should revert if the vault's status it not set to `NotTraded`.
  ☑ Negative test
- Should revert if the `_tradeExpiry` argument is set to 0.
  ☑ Negative test
- Should revert if called by a non-trader-admin role.
  ☑ Negative test

## Function: `setVaultMetadata(address vaultAddress, FCNVaultMetadata metadata)`

Sets a vault's metadata.

### Inputs

- `vaultAddress`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The metadata for this vault is updated.
- `metadata`
  - **Control**: Fully controlled.
  - **Constraints**: The `metadata.vaultStart` property must not be 0.
  - **Impact**: The metadata for the vault is updated to this.

### Branches and code coverage (including function calls)

#### Intended branches

- Should set the correct vault's metadata using the `vaultAddress`.
  ☑ Test coverage

#### Negative behaviour

- Should revert if `metadata.vaultStart` is set to 0.
  - ☑ Negative test
- Should revert if called by a non–default-admin role.
  - ☑ Negative test

## Function: `setVaultStatus(address vaultAddress, VaultStatus _vaultStatus)`

Sets a vault's status.

### Inputs

- `vaultAddress`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This vault's status is updated.
- `_vaultStatus`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the new status that will be set.

### Branches and code coverage (including function calls)

#### Intended branches

- Should update the vault status correctly.
  - ☑ Test coverage

#### Negative behaviour

- Should revert if called by a non–operator-admin role.
  - ☑ Negative test

## Function: `setYieldFeeBps(uint256 _yieldFeeBps)`

This function sets the global yield fee percentage for this FCNProduct. Only callable by the operator admin role.

### Inputs

- `_yieldFeeBps`
  - **Control**: Fully controlled.

  - **Constraints**: N/A.
  - **Impact**: Modifies the `yieldFeeBps` storage variable.

### Branches and code coverage (including function calls)

**Intended branches**

- Modifies the `yieldFeeBps` storage variable correctly.
  - ☑ Test coverage

**Negative behaviour**

- Should revert if called by a non-operator-admin role.
  - ☑ Negative test

### Function: `updateOptionBarrierOracle(address vaultAddress, uint256 index, string _asset, string newOracleName)`

Updates the oracle being used by a specific option barrier.

### Inputs

- `vaultAddress`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This vault's metadata is updated.
- `index`
  - **Control**: Fully controlled.
  - **Constraints**: Must be less than the vault metadata's `optionBarrierCount`.
  - **Impact**: Used to index into the vault metadata's `optionBarriers` array.
- `_asset`
  - **Control**: Fully controlled.
  - **Constraints**: Must match the asset of the option barrier being updated.
  - **Impact**: N/A.
- `newOracleName`
  - **Control**: Fully controlled.
  - **Constraints**: Must exist in the `CegaState` contract's `oracleAddresses` mapping.
  - **Impact**: Used to update the option barrier.

### Branches and code coverage (including function calls)

**Intended branches**

- Should correctly update the oracle of the indexed option barrier.
  ☑ Test coverage

**Negative behaviour**

- Should revert if called by a non-operator-admin role.
  ☑ Negative test
- Should revert if the `_asset` argument does not match the `asset` of the indexed option barrier.
  ☑ Negative test
- Should revert if the `newOracleName` argument does not exist in the CegaState contract's `oracleAddresses` mapping.
  ☑ Negative test

## 5.3 Module: Oracle.sol

### Function: `addNextRoundData(RoundData _roundData)`

The `addNextRoundData` function pushes a pricing data struct (`RoundData`) to storage (`oracleData`). Its caller must be a service admin.

### Inputs

- `_roundData`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The `RoundData` struct to store. Note that the service admin could force a knock-in or knock-out event.

### Branches and code coverage (including function calls)

**Intended branches**

- Function pushes `RoundData` struct.
  ☑ Test coverage

**Negative behaviour**

- Function is called by non-service admin.
  ☑ Negative test

### Function: `updateRoundData(uint80 roundId, RoundData _roundData)`

This function changes the pricing data struct (`RoundData`) for a given round number (`roundId`).

#### Inputs

- `roundId`
  - **Control**: Full.
  - **Constraints**: Value must be less than the size of the `oracleData` array or the function call will revert with an index-out-of-bounds error.
  - **Impact**: The round ID to update in the `oracleData` array.
- `_roundData`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The new `RoundData` struct to store.

#### Branches and code coverage (including function calls)

**Intended branches**

- Updates the pricing round data stored at the `roundId` index of the `oracleData` array.
  - [x] Test coverage

**Negative behaviour**

- Function is called by non-service admin.
  - [x] Negative test
- Function is called with a `roundId` that is not a valid index in the `oracleData` array.
  - [ ] Negative test

# 6 Audit Results

At the time of our audit, the code was not deployed to mainnet.

During our audit, we discovered nine findings. Of these, two were high risk, four were medium risk, one was low risk, and two were suggestions (informational). Sanic Pte. Ltd. acknowledged all findings and implemented fixes.

## 6.1 Disclaimers

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 additional code added to the assessed project after the audit version of 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 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.

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.