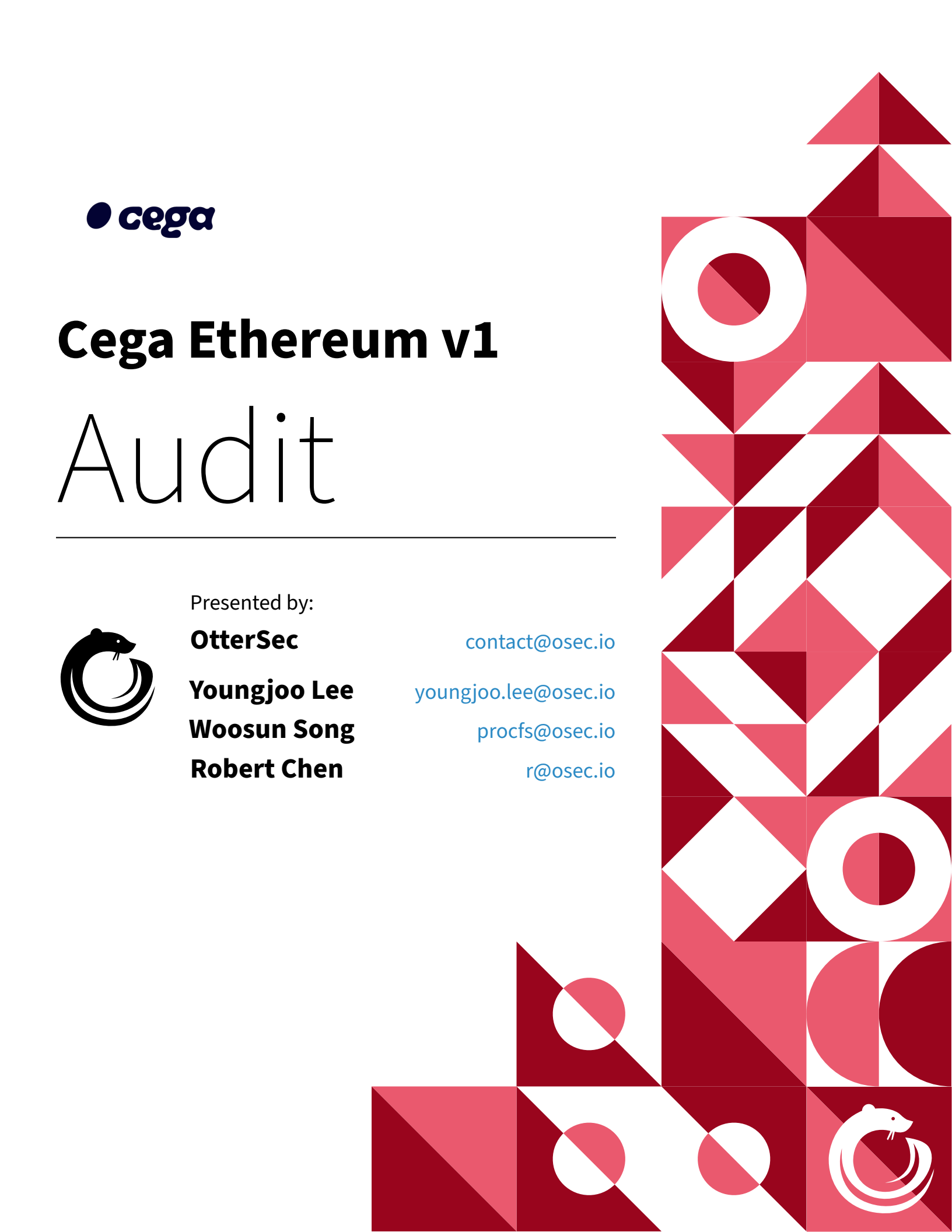


Cega Ethereum v1

Audit

Presented by:

OtterSec contact@osec.io

Youngjoo Lee youngjoo.lee@osec.io

Woosun Song procfs@osec.io

Robert Chen r@osec.io

Contents

Appendices

01 | Executive Summary

Overview

Cega Finance engaged OtterSec to perform an assessment of the `cega-eth-v1` program. This assessment was conducted between February 13th and March 6th, 2023. For more information on our auditing methodology, see Appendix B.

Critical vulnerabilities were communicated to the team prior to the delivery of the report to speed up remediation. After delivering our audit report, we worked closely with the team to streamline patches and confirm remediation. We delivered final confirmation of the patches March 11th, 2023.

Key Findings

Over the course of this audit engagement, we produced 12 findings total.

In particular, we found an issue with `vaultAddress` validation that could result in the loss of user's funds under limited circumstances (OS-CEG-ADV-00). Additionally, we found an issue that leads to a denial of service regarding deposit and withdrawal queues (OS-CEG-ADV-01).

We also made recommendations around potential reentrancy risks, which although not currently exploitable, should be addressed to avoid potential vulnerabilities in the future (OS-CEG-SUG-00). Furthermore, we suggested adding validations for crucial market parameters to mitigate against a malicious trader admin (OS-CEG-SUG-03, OS-CEG-SUG-04).

Overall, we commend the Cega Finance team for being responsive and knowledgeable throughout the audit.

02 | Scope

The source code was delivered to us in a git repository at github.com/cega-fi/cega-fi-cega-eth-v1. This audit was performed against commit `e0ffc80`.

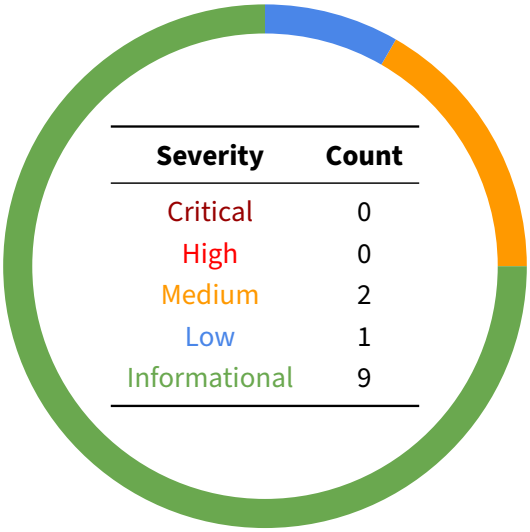
A brief description of the programs is as follows.

Name	Description
cega-eth-v1	`cega-eth-v1` is a Solidity-based version of the existing Solana-based program `cega`, which is designed to facilitate a fixed coupon note with knock-in barrier options. The contract consists of two components, the product and the vault: • The product holds the financial parameters, such as the reward percentage and trade duration, and is responsible for creating and managing multiple vaults • The vaults receive assets and mint shares in return, and they are opened, traded, and closed in a round-robin fashion on a periodic basis.

03 | Findings

Overall, we reported 12 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings don't have an immediate impact but will help mitigate future vulnerabilities.



Severity	Count
Critical	0
High	0
Medium	2
Low	1
Informational	9

04 | Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-CEG-ADV-00	Medium	Resolved	`addToWithdrawalQueue` and `addOptionBarrier` do not perform validation on the `vaultAddress` argument.
OS-CEG-ADV-01	Medium	Resolved	In `FCNProduct`, the number of iterations performed in queue processor functions may be increased indefinitely by abusing queuing functions, resulting in a denial of service.
OS-CEG-ADV-02	Low	Resolved	Ignoring the timestamp of oracle data when computing knock-in conditions and ratios may lead to incorrect behaviour due to the use of outdated price data.

OS-CEG-ADV-00 [med] | Unchecked Vault Address

Description

In `addToWithdrawalQueue` and `addOptionBarrier`, the `vaultAddress` argument is not validated.

contracts/FCNProduct.sol SOLIDITY

```
function addToWithdrawalQueue(
        address vaultAddress,
        uint256 amountShares,
        address receiver
) public {
    FCNVaultMetadata storage vaultMetadata = vaults[vaultAddress];

    IERC20(vaultAddress).safeTransferFrom(receiver, address(this),
    ↪  amountShares);
    Withdrawal[] storage withdrawalQueue =
    ↪  withdrawalQueues[vaultAddress];
    withdrawalQueue.push(Withdrawal({ amountShares: amountShares,
    ↪  receiver: receiver }));
    vaultMetadata.queuedWithdrawalsCount += 1;
    vaultMetadata.queuedWithdrawalsSharesAmount += amountShares;

    emit WithdrawalQueued(vaultAddress, receiver, amountShares);
}
```

In the case of `addToWithdrawalQueue`, the absence of such checks may result in a loss of funds. By abusing this function, it is possible to perform `ERC20.transferFrom` on a token address and receiver address of any choice. Thus, if a user has given approval for the product, they may lose their funds.

Remediation

All functions that take `vaultAddress` as an argument must perform checks to ensure that the vault is initialized.

Patch

Fixed in #72 by adding the `validVault` modifier, which checks if `vaultMetadata.tradeStart` is nonzero.

OS-CEG-ADV-01 [med] | Unbounded Iterations

Description

The functions `processDepositQueue` and `processWithdrawalQueue` contain a loop that iterates for a number of times equal to the length of the queue.

contracts/FCNProduct.sol SOLIDITY

```
function processWithdrawalQueue(address vaultAddress, uint256
    ↪  maxProcessCount) public onlyTraderAdmin {
   /* abridged */
    for (i = processCount; i < vaultMetadata.queuedWithdrawalsCount; i++)
    ↪  {
        withdrawal = withdrawalQueue[i];
        withdrawalQueue[i - processCount] = withdrawal;
    }
    /* abridged */
}
```

An adversary may abuse this property by adding numerous requests to the queue, causing the processor function to exceed the block gas limit.

Remediation

Place an upper bound on the number of iterations performed.

Patch

Resolved in #76 by updating the queue shifting algorithm.

OS-CEG-ADV-02 [low] | Unchecked Oracle Recency

Description

When calculating knock-in conditions and ratios, the timestamp of the oracle data is not considered. For instance, the `calculateKnockInRatio` function discards the `startedAt` and `updatedAt` fields of the round data. A delay in the oracle due to unforeseen circumstances may result in the use of outdated price values when computing the knock-in ratio.

contracts/FCNProduct.sol SOLIDITY

```
function calculateKnockInRatio(
    FCNVaultMetadata storage self,
    address cegaStateAddress
) public view returns (uint256) {
    /* abridged */
    for (uint256 i = 0; i < optionBarriersCount; i++) {
        OptionBarrier memory optionBarrier = optionBarriers[i];
        address oracle = getOracleAddress(optionBarrier,
    ↪   cegaStateAddress);
        (, int256 answer, , , ) = IOracle(oracle).latestRoundData();
        /* abridged */
    }
}
```

Remediation

The validity of the price data should be verified by checking its timestamp against a predefined threshold or interval, and any price that falls outside of this range should be considered outdated and discarded.

Patch

Fixed in #109.

05 | General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent antipatterns and could lead to security issues in the future.

ID	Description
OS-CEG-SUG-00	The location of `ERC20.transfer` call creates a risk of reentrancy.
OS-CEG-SUG-01	Explicit array bounds checks are unnecessary since they are inserted automatically by the `solc-0.8.17`.
OS-CEG-SUG-02	When checking for knock-ins, gas consumption may be reduced by performing oracle lookups for knock-in barriers only.
OS-CEG-SUG-03	A malicious trader admin may obtain a profit by concluding the trade immediately without any checks for the trade expiry date.
OS-CEG-SUG-04	There is a risk of inconsistent values if the `TenorInDays` parameter is set to a value that deviates from `tradeExpiry - tradeDate`.
OS-CEG-SUG-05	The current status should be validated before executing functions that change the vault status.
OS-CEG-SUG-06	Utilize explicit function overrides to prevent confusion.
OS-CEG-SUG-07	The `FCNVault` contains several functions that are not used by `FCNProduct`.
OS-CEG-SUG-08	The array removal algorithm used in the `removeOptionBarrier` function could be optimized to achieve constant gas complexity.

OS-CEG-SUG-00 | Potential Reentrancy Risk

Description

The functions `processWithdrawalQueue`, `sendAssetsToTrade`, and `collectFees` pose a reentrancy risk due to their usage of `ERC20.transfer`. When the transferred token is ERC-777, there is a possibility of interleaving, which triggers a callback on the receiver address upon receipt.

contracts/FCNProduct.sol SOLIDITY

```
function collectFees(address vaultAddress) public onlyTraderAdmin {
    FCNVaultMetadata storage vaultMetadata = vaults[vaultAddress];
    require(vaultMetadata.vaultStatus == VaultStatus.PayoffCalculated,
    ↪  "500:WS");

    (uint256 totalFees, uint256 managementFee, uint256 yieldFee) =
    ↪  calculateFees(vaultAddress);
    totalFees = Math.min(totalFees, vaultMetadata.vaultFinalPayoff);
    IERC20(asset).safeTransfer(cegaState.feeRecipient(), totalFees);
    vaultMetadata.currentAssetAmount -= totalFees;

    vaultMetadata.vaultStatus = VaultStatus.FeesCollected;
    sumVaultUnderlyingAmounts -= vaultMetadata.underlyingAmount;
    vaultMetadata.underlyingAmount = vaultMetadata.vaultFinalPayoff -
    ↪  totalFees;
    sumVaultUnderlyingAmounts += vaultMetadata.underlyingAmount;
    /* abridged */
}
```

Currently, the reentrancies are benign due to access control. For instance, the interleaving due to `ERC20.transferFrom` is only exploitable if the attacker possesses both the `traderAdmin` and `feeRecipient` roles. However, it may escalate to a security concern as more functionalities are added. Therefore, hardening the aforementioned functions is recommended.

Remediation

There are two methods of remediation:

1. Place the interleaving call after all state updates are performed.
2. Use mutex primitives, such as OpenZeppelin's `reentrancyGuard`.

Patch

Fixed in #74 and #101.

OS-CEG-SUG-01 | Redundant Array Bounds Checks

Description

We have observed several instances where explicit bounds checks are used for array accesses.

contracts/FCNProduct.sol SOLIDITY

```
function updateOptionBarrier(
    address vaultAddress,
    uint256 index,
    string calldata _asset,
    uint256 _strikeAbsoluteValue,
    uint256 _barrierAbsoluteValue
) public onlyTraderAdmin {
    FCNVaultMetadata storage vaultMetadata = vaults[vaultAddress];
    require(index < vaultMetadata.optionBarriersCount, "400:I");

    OptionBarrier storage optionBarrier =
    ↪  vaultMetadata.optionBarriers[index];
    require(keccak256(abi.encodePacked(optionBarrier.asset)) ==
    ↪  keccak256(abi.encodePacked(_asset)), "400:AS");

    optionBarrier.strikeAbsoluteValue = _strikeAbsoluteValue;
    optionBarrier.barrierAbsoluteValue = _barrierAbsoluteValue;
}
```

However, such checks are redundant because `solc-0.8.17` automatically inserts bounds checks for all array accesses.

Remediation

Remove explicit array bounds checks for array accesses.

Patch

Fixed in #99.

OS-CEG-SUG-02 | Oracle Lookup Optimization

Description

Within the functions, `checkBarriers` and `calculateKnockInRatio`, the number of oracle lookups is equal to the number of option barriers present in the vault. However, the number of lookups may be reduced by performing lookups only when a knock-in type barrier is encountered.

contracts/Calculations.sol SOLIDITY

```
 function checkBarriers(FCNVaultMetadata storage self, address
    ↪  cegaStateAddress) public {
    if (self.isKnockedIn == true) {
        return;
    }

    require(self.vaultStatus == VaultStatus.Traded, "500:WS");

    for (uint256 i = 0; i < self.optionBarriersCount; i++) {
        OptionBarrier storage optionBarrier = self.optionBarriers[i];
        address oracle = getOracleAddress(optionBarrier,
    ↪  cegaStateAddress);
        (, int256 answer, , , ) = Oracle(oracle).latestRoundData();

        // Knock In: Check if current price is less than barrier
        if (optionBarrier.barrierType == OptionBarrierType.KnockIn) {
            if (uint256(answer) <= optionBarrier.barrierAbsoluteValue) {
                self.isKnockedIn = true;
            }
        }
    }
}
```

Determining the current price is unnecessary for None type barriers.

Remediation

Move the oracle lookup invocation into the if statement so that it is performed only for knock-in type barriers.

Patch

Fixed in #92.

OS-CEG-SUG-03 | Unchecked Expiry Date

Description

The trader admin has the authorization to freely set `tradeDate`, `tradeExpiry`, `aprBps`, and `tenorInDays`. A malicious trader admin can exploit this freedom by manipulating the `aprBps` and `tenorInDays` parameters to terminate the coupon prematurely. This is achievable because there are no safeguards in place to ensure a minimum expiry date.

contracts/FCNProduct.sol SOLIDITY

```
) public onlyTraderAdmin validVault(vaultAddress) {
    FCNVaultMetadata storage metadata = vaults[vaultAddress];
    require(metadata.vaultStatus == VaultStatus.NotTraded, "500:WS");
    require(_tradeExpiry > 0, "400:TE");
    metadata.tradeDate = _tradeDate;
    metadata.tradeExpiry = _tradeExpiry;
    metadata.aprBps = _aprBps;
    metadata.tenorInDays = _tenorInDays;
}
```

Remediation

To prevent malicious actions by a trader admin, it is advised to establish a minimum expiry date. If the admin sets the `aprBps` and `tenorInDays` to an abnormally high value, other privileged roles may modify them before the trade is finalized. This ensures that the trader admin cannot receive a payoff before the trade has expired.

contracts/FCNProduct.sol SOLIDITY

```
    require(_tradeExpiry > block.timestamp + 1 days, "400:TE");
    metadata.tradeDate = _tradeDate;
    metadata.tradeExpiry = _tradeExpiry;
```

Patch

Fixed in #83.

OS-CEG-SUG-04 | Unchecked Tenor Period

Description

`tenorInDays` is used when calculating the final payoff.

contracts/Calculations.sol SOLIDITY

```
// Calculate coupon payment
totalCouponPayment = calculateCouponPayment(self.underlyingAmount,
    ↪  self.aprBps, self.tenorInDays);
```

`tradeDate` is used when calculating the final payoff in `calculateCurrentYield()`. Therefore, `tenorInDays` should be equal to `tradeExpiry - tradeDate`. However, there are no measures in place to enforce this equivalence.

contracts/Calculations.sol SOLIDITY

```
uint256 numberOfDaysPassed = (currentTime - self.tradeDate) /
    ↪  SECONDS_TO_DAYS;
self.totalCouponPayoff = calculateCouponPayment(self.underlyingAmount,
    ↪  self.aprBps, numberOfDaysPassed);
```

Remediation

Add a consistency check for `tenorInDays`.

contracts/FCNProduct.sol SOLIDITY

```
    require(metadata.tenorInDays == (metadata.tradeExpiry -
    ↪  metadata.tradeDate));
    metadata.tradeDate = _tradeDate;
    metadata.tradeExpiry = _tradeExpiry;
    metadata.aprBps = _aprBps;
    metadata.tenorInDays = _tenorInDays;
```

Patch

Fixed in #83 by accepting a difference of one day or less between `tenorInDays` and `tradeExpiry - tradeDate`.

OS-CEG-SUG-05 | Insufficient Status Validation

Description

To avoid an inconsistent vault state, it's important to validate the current state when invoking functions that modify the vault status, in order to maintain a strict ordering between the calls.

For instance, the `openVaultDeposits` function alters the current vault state to `DepositsOpen`. The function does not verify whether the vault status is `DepositsClosed`, and invoking it on a non-closed vault may result in undefined behaviour.

contracts/FCNProduct.sol SOLIDITY

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

Remediation

Add validation for the current vault status in state-transition functions, such as `openVaultDeposits`.

contracts/FCNProduct.sol SOLIDITY

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

Patch

Fixed in #75.

OS-CEG-SUG-06 | Use Explicit Function Overrides

Description

Use explicit function overrides in `FCNVault`. The use of explicit function overrides may improve the clarity and safety of contracts with inheritance by ensuring a defined implementation.
The list of overridden functions are:

- `totalAssets()`
- `convertToAssets(uint256 shares)`
- `convertToShares(uint256 assets)`
- `maxDeposit(address)`
- `previewDeposit(uint256 assets)`
- `deposit(uint256 assets, address receiver)`
- `deposit(uint256 assets)`
- `maxMint(address)`
- `previewMint(uint256 shares)`
- `mint(uint256 shares, address receiver)`
- `mint(uint256 shares)`
- `maxWithdraw(address`
- `previewWithdraw(uint256 assets)`
- `withdraw(uint256 assets, address receiver, address owner)`
- `withdraw(uint256 assets)`
- `maxRedeem(address)`
- `previewRedeem(uint256 shares)`
- `redeem(uint256 shares, address receiver, address owner)`
- `redeem(uint256 shares, address receiver)`
- `redeem(uint256 shares)`

Remediation

Add `override` keyword to the presented functions.

Patch

Fixed in #87.

OS-CEG-SUG-07 | Unused Functions

Description

Unused functions in `FCNVault` should have a function body with an unconditional `revert()` to prevent unintended behaviour. The list of unused functions are:

- `maxDeposit(address)`
- `previewDeposit(uint256 assets)`
- `deposit(uint256 assets)`
- `maxMint(address)`
- `previewMint(uint256 shares)`
- `mint(uint256 shares, address receiver)`
- `mint(uint256 shares)`
- `maxWithdraw(address`
- `previewWithdraw(uint256 assets)`
- `withdraw(uint256 assets, address receiver, address owner)`
- `withdraw(uint256 assets)`
- `maxRedeem(address)`
- `previewRedeem(uint256 shares)`
- `redeem(uint256 shares)`

Remediation

Change the body of the presented functions to unconditionally revert.

Patch

Fixed in #87 by changing `FCNVault` to no longer inherit to `IERC4626`.

OS-CEG-SUG-08 | Optimize Array Removal

The element removal algorithm used by the `removeOptionBarrier` function, which involves left-shifting multiple elements, may lead to a gas consumption that scales linearly with the number of option barriers. The algorithm can be optimized to have a constant gas consumption instead.

contracts/FCNProduct.sol SOLIDITY

```
// Shift all elements to the left.
// Element at "index" becomes overwritten. Last element is now duplicated,
    ↪  so we can remove it.
for (uint256 i = index; i < vaultMetadata.optionBarriersCount - 1; i++) {
    optionBarriers[i] = optionBarriers[i + 1];
}
optionBarriers.pop();
vaultMetadata.optionBarriersCount -= 1;
```

Remediation

Employ an optimized algorithm that involves exchanging the element to be deleted with the last element in the array, and then removing the last element with a pop operation, instead of shifting multiple elements.

Patch

Fixed in #95.

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

Critical	Vulnerabilities that immediately lead to loss of user funds with minimal preconditions Examples: • Misconfigured authority or access control validation • Improperly designed economic incentives leading to loss of funds
High	Vulnerabilities that could lead to loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions • Exploitation involving high capital requirement with respect to payout
Medium	Vulnerabilities that could lead to denial of service scenarios or degraded usability. Examples: • Malicious input that causes computational limit exhaustion • Forced exceptions in normal user flow
Low	Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants • Improved input validation

B | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the implementation of the program requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of sum, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.