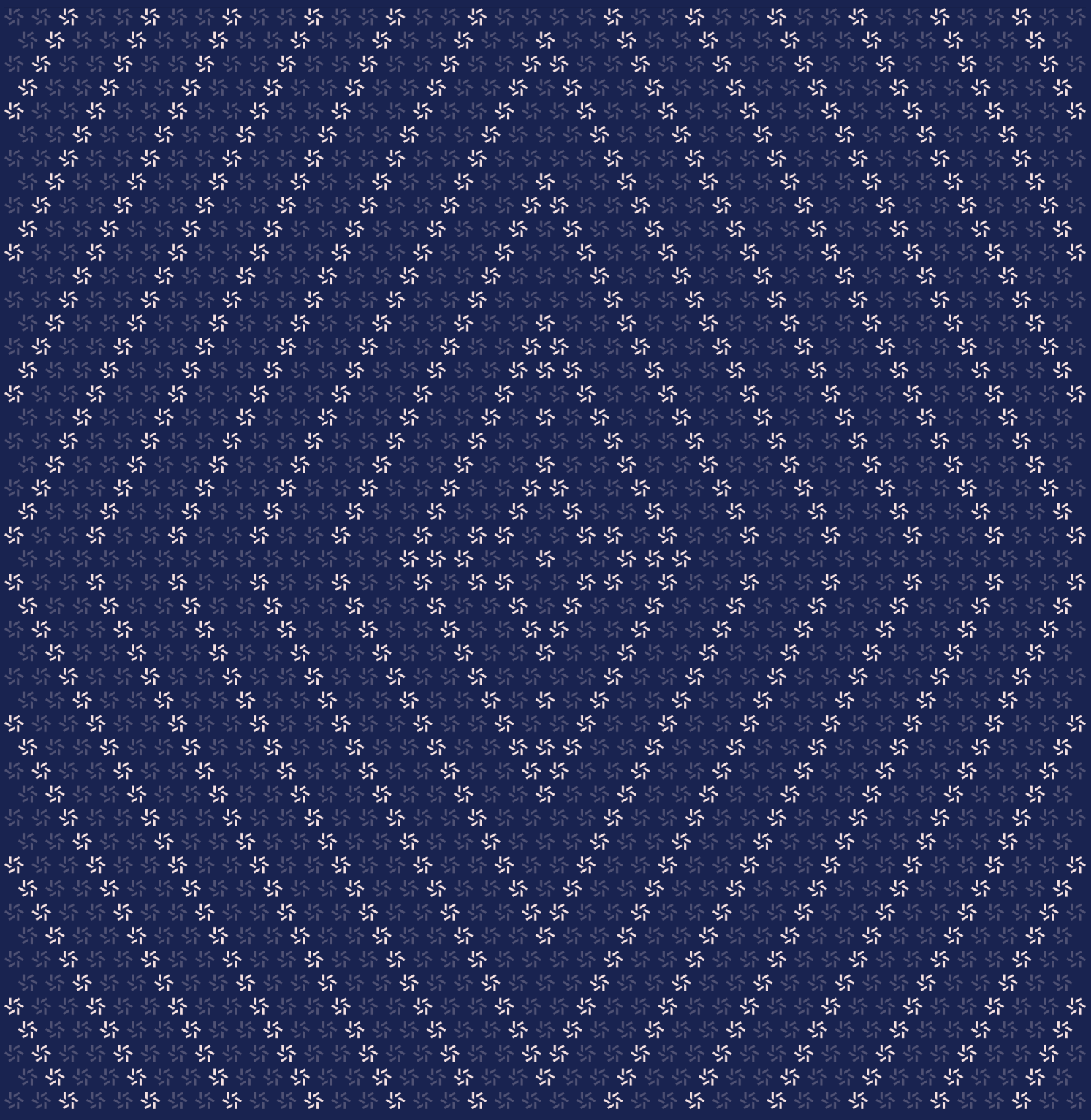
Zellic

Prepared for
Nicholas Roberts-Huntley
Blueprint Finance

Prepared by
Qingying Jie
Weipeng Lai
Zellic

June 19, 2025

Concrete
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 Blueprint Finance from May 16th to June 6th, 2025. During this engagement, Zellic reviewed Concrete's 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:

- Are there any vulnerabilities that could result in the loss of user funds?
- Are there any missing access controls on critical functions?
- Can an attacker manipulate share-price calculations to steal funds?
- Can reward calculations be manipulated to receive more rewards?
- Can the high water mark be manipulated to affect performance fees?

1.3. Non-goals and Limitations

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

- Strategies not listed in the Scope section
- 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 Concrete contracts, we discovered 25 findings. No critical issues were found. Two findings were of high impact, six were of medium impact, eight were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Blueprint Finance in the Discussion section (4. ↗).

Based on the high number of findings uncovered during the audit, it is our opinion that the project is not yet ready for production. We strongly advise a comprehensive reassessment before deployment to help identify any potential issues or vulnerabilities introduced by necessary fixes or changes. We also recommend adopting a security-focused development workflow, including (but not limited to) augmenting the repository with comprehensive end-to-end tests that achieve 100% branch coverage using any common, maintainable testing framework, thoroughly documenting all function requirements, and training developers to have a security mindset while writing code.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	2
Medium	6
Low	8
Informational	9

2. Introduction

2.1. About Concrete

Blueprint Finance contributed the following description of Concrete:

> Concrete is the DeFi Liquidity Metalayer — powering the highest yields and unlocking new derivatives for any on-chain asset.
>
> Concrete offers a one-stop solution that automates everything from yield optimization to liquidation protection. Handling the research, security, and optimization while giving users the best of DeFi without the risks or headaches.

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.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Concrete Contracts

Type	Solidity
Platform	EVM-compatible
Target	sc_earn-v1
Repository	https://github.com/Blueprint-Finance/sc_earn-v1 ↗
Version	`c68890cea19b88b3360f306d1f5a37ebb7679f64`

Programs	`registries/*.sol`
	`utils/Constants.sol`
	`managers/*.sol`
	`parking/ParkingLot.sol`
	`strategies/StrategyBase.sol`
	`strategies/Aave/DataTypes.sol`
	`strategies/Aave/AaveV3Strategy.sol`
	`strategies/Morpho/MorphoVaultStrategy.sol`
	`strategies/MultiSigStrat/MultiSigStrategy.sol`
	`libraries/*.sol`
	`vault/ConcreteMultiStrategyVault.sol`
	`vault/ConcreteMultiStrategyVaultUpgradeableV1.sol`
	`queue/WithdrawalQueue.sol`
	`factories/VaultFactory.sol`
	`oracle/BTCLinkedPriceFeed.sol`
	`oracle/OracleFactory.sol`
	`oracle/ConcreteOracle.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 4.5 person-weeks. The assessment was conducted by two consultants over the course of 3.2 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:

Qingying Jie
Engineer
qingying@zellic.io ↗

Weipeng Lai
Engineer
weipeng.lai@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

May 16, 2025	Start of primary review period
May 20, 2025	Kick-off call
June 6, 2025	End of primary review period

3. Detailed Findings

3.1. Incorrect performance-fee calculation can lead to a denial-of-service condition

Target	FeesHelper		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	Medium	**Impact**	High

Description

The `calculateTieredFee` function in the FeesHelper library calculates the performance fee based on the current price of 1e18 vault shares (`shareValue`) and the last reached highest price (`highWaterMark`). The function incorrectly calculates the fee by dividing by `10 ** underlayingDecimals` instead of the `shareValue`.

```
function calculateTieredFee(
    uint256 shareValue,
    uint256 highWaterMark,
    uint256 totalAssets,
    VaultFees storage fees,
    uint256 underlayingDecimals
) public view returns (uint256 fee) {
    // [...]
            fee = ((shareValue - highWaterMark) * totalAssets).mulDiv(
                fees.performanceFee[i].fee, MAX_BASIS_POINTS
    * 10 ** underlayingDecimals, Math.Rounding.Floor
            );
    // [...]
    }
}
```

This error can cause `calculateTieredFee` to return a `fee` value that is larger than the vault's `totalAssets`. When this happens, the subtraction in `_totalAssets - totalFee` within the `takeFees` modifier will underflow and cause a revert.

```
modifier takeFees() {
    if (!paused()) {
        uint256 totalFee = accruedProtocolFee() + accruedPerformanceFee();
        uint256 shareValue = convertToAssets(1e18);
        uint256 _totalAssets = totalAssets();

        if (shareValue > highWaterMark) highWaterMark = shareValue;
```

```
        if (totalFee > 0 && _totalAssets > 0) {
            uint256 supply = totalSupply();
            uint256 feeInShare =
                supply == 0 ? totalFee : totalFee.mulDiv(supply, _totalAssets
    - totalFee, Math.Rounding.Floor);
            _mint(feeRecipient, feeInShare);
            feesUpdatedAt = block.timestamp;
        }
    }
    _;
}
```

Because the `takeFees` modifier executes before key user operations — including `deposit`, `mint`, `withdraw`, and `redeem` — this vulnerability can place the vault into a permanent denial-of-service (DOS) state, blocking all core functionality.

Impact

An attacker can trigger this vulnerability by directly transferring assets into the vault, which inflates the `shareValue`.

By inflating `shareValue` sufficiently, an attacker can force `calculateTieredFee` to return a `fee` that exceeds the vault's total assets. This condition will cause the `takeFees` modifier to revert consistently. As a result, all user-facing functions that use the modifier will be blocked, leading to a permanent DOS condition and freezing all user funds within the vault.

Recommendations

We recommend correcting the performance-fee calculation in `calculateTieredFee`. The fee should be calculated relative to the `shareValue` to ensure it is always proportional to the actual profit generated.

```
fee = ((shareValue - highWaterMark) * totalAssets).mulDiv(

    fees.performanceFee[i].fee, MAX_BASIS_POINTS * 10 ** underlayingDecimals,
        Math.Rounding.Floor
    fees.performanceFee[i].fee, MAX_BASIS_POINTS * shareValue, Math.Rounding.
        Floor
);
```

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit `c8e45fa4`.

3.2. Unhandled code path in `WithdrawalQueueHelper._withdrawStrategyFunds` could result in users receiving less assets than they are entitled to

Target	WithdrawalQueueHelper		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

During a user withdrawal from a Concrete Vault, if the withdrawable amount is sufficient for the requested withdrawal, execution proceeds to the `WithdrawalQueueHelper._withdrawStrategyFunds` function:

```
function processWithdrawal(
    uint256 assets_,
    address receiver_,
    uint256 availableAssets,
    address asset,
    address withdrawalQueue,
    uint256 minQueueRequest,
    Strategy[] memory strategies,
    IParkingLot parkingLot
) external {
    if (availableAssets >= assets_) {
        _withdrawStrategyFunds(assets_, receiver_, asset, strategies,
    parkingLot);
    } else {
        // [...]
    }
}
```

The `_withdrawStrategyFunds` function first checks the asset balance in the vault contract (`float`). If the balance is sufficient for the withdrawal, the function withdraws funds from the balance. Otherwise, the vault withdraws proportional amounts from the strategies to the user based on the strategies' allocations:

```
function _withdrawStrategyFunds(
    uint256 amount_,
    address receiver_,
    address asset_,
    Strategy[] memory strategies,
```

```
    IParkingLot parkingLot
) internal {
    // [...]
    uint256 float = _asset.balanceOf(address(this));

    if (amount_ <= float) {
        bool result = TokenHelper.attemptSafeTransfer(address(asset_),
    receiver_, amount_, false);
        // [...]
    } else {
        uint256 diff = amount_ - float;
        uint256 totalWithdrawn = 0;
        uint256 len = strategies.length;
        for (uint256 i; i < len;) {
            // [...]
            uint256 amountToWithdraw = _calculateWithdrawalAmount(amount_,
    strategy);
            // [...]
            try strategy.strategy.withdraw(amountToWithdraw, receiver_,
    address(this)) {}
            catch {
                // [...]
            }
            totalWithdrawn += amountToWithdraw;
            unchecked {
                i++;
            }
        }
        if (totalWithdrawn < amount_ && amount_ - totalWithdrawn <= float) {
            uint256 net = amount_ - totalWithdrawn;
            bool result = TokenHelper.attemptSafeTransfer(address(asset_),
    receiver_, net, false);
            // [...]
        }
    }
}


function _calculateWithdrawalAmount(uint256 amount_, Strategy memory strategy)
    internal pure returns (uint256) {
    return amount_.mulDiv(strategy.allocation.amount, MAX_BASIS_POINTS,
    Math.Rounding.Floor);
}
```

In the latter case, if withdrawals from the strategies are insufficient for the requested amount, the function withdraws the remaining amount from the vault's asset balance. However, if insufficient

asset balance remains in the vault, the function proceeds without reverting. Consequently, users receive less than they are entitled to.

Impact

Users receive less than they are entitled to if the asset balance in the vault is insufficient for the remaining amount after proportional withdrawals from the strategies.

The following scenario illustrates this issue. Assume the Concrete Vault has 20% allocation to strategy X and 30% allocation to strategy Y.

1. Alice and Bob each deposit 500 tokens to the vault. The vault then contains 200 tokens in strategy X, 300 tokens in strategy Y, and 500 tokens in the vault balance.
2. Bob withdraws 500 tokens from the vault. The `_withdrawStrategyFunds` function transfers 500 tokens from the vault's balance to Bob. After the withdrawal, zero asset balance remains in the vault.
3. Alice requests a withdrawal of 500 tokens. The `_withdrawStrategyFunds` function transfers 100 tokens (500 × 20%) from strategy X to Alice and transfers 150 tokens (500 × 30%) from strategy Y to Alice, then proceeds with execution.

As a result, Alice burns shares worth 500 tokens but only receives 250 tokens during withdrawal.

Recommendations

Consider reverting if the asset balance in the contract is insufficient for the remaining amount after proportional withdrawals from the strategies:

```
if (totalWithdrawn < amount_ && amount_ - totalWithdrawn <= float) {
    uint256 net = amount_ - totalWithdrawn;
    bool result = TokenHelper.attemptSafeTransfer(address(asset_), receiver_,
    net, false);
    if (!result) {
        parkingLot.deposit(receiver_, net);
    }
}
} else {revert("...");}
```

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit 4bc00f6a ↗.

Blueprint Finance provided the following response to this finding:

> The `_withdrawStrategyFunds` function has been restructured. The strategy allocation percentage is now calculated relative to the **total strategy allocation**, rather than using `MAX_BASIS_POINTS`.

3.3. Potential underflow in `MorphoVaultStrategy._protocolWithdraw` could lead to withdrawal failure

Target	MorphoVaultStrategy		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

When users withdraw from a Concrete Vault that uses a MorphoVaultStrategy, the vault withdraws a portion of assets from the MorphoVaultStrategy, and execution proceeds to the `_protocolWithdraw` function.

The `_protocolWithdraw` function in MorphoVaultStrategy does not validate whether the `assets_` parameter exceeds the contract's current `_assetBalance` before performing the subtraction `assets_ -= _assetBalance`. This omission causes an arithmetic underflow when `_assetBalance` exceeds `assets_`.

```
function _protocolWithdraw(uint256 assets_, uint256)
    internal virtual override {
    address _asset = asset();
    uint256 _assetBalance = IERC20(_asset).balanceOf(address(this));
    if (_assetBalance > 0) {
        assets_ -= _assetBalance;
    }
    // [...]
}
```

The asset balance of the MorphoVaultStrategy contract (`_assetBalance`) can exceed the requested withdrawal amount (`assets_`) in two scenarios:

1. The `_autoCompoundRewards` function executes before the withdrawal and swaps harvested MORPHO reward tokens into asset tokens.
2. An attacker transfers assets directly to the MorphoVaultStrategy to inflate `_assetBalance`.

When either scenario occurs, withdrawal attempts revert due to arithmetic underflow.

Impact

Users cannot withdraw their funds from the vault when the MorphoVaultStrategy's asset balance exceeds the requested withdrawal amount. Withdrawals remain blocked until other users' withdrawals reduce the strategy's asset balance below the requested amount.

Recommendations

Consider adding an early return in `_protocolWithdraw` when the strategy's asset balance satisfies the withdrawal request.

```
function _protocolWithdraw(uint256 assets_, uint256)
    internal virtual override {
    address _asset = asset();
    uint256 _assetBalance = IERC20(_asset).balanceOf(address(this));
    if (_assetBalance > 0) {
        if (assets_ <= _assetBalance) return;
        assets_ -= _assetBalance;
    }
    // [...]
}
```

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit `4b5f74fa` ↗.

3.4. The `feesUpdatedAt` state variable is used before initialization

Target	ConcreteMultiStrategyVault		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	Medium

Description

The `takeFees` modifier in the ConcreteMultiStrategyVault contract calculates protocol fees using the `feesUpdatedAt` state variable and updates it after accruing fees:

```
modifier takeFees() {
    if (!paused()) {
        uint256 totalFee = accruedProtocolFee() + accruedPerformanceFee();
        uint256 shareValue = convertToAssets(1e18);
        uint256 _totalAssets = totalAssets();

        if (shareValue > highWaterMark) highWaterMark = shareValue;

        if (totalFee > 0 && _totalAssets > 0) {
            uint256 supply = totalSupply();
            uint256 feeInShare =
                supply == 0 ? totalFee : totalFee.mulDiv(supply, _totalAssets
    - totalFee, Math.Rounding.Floor);
            _mint(feeRecipient, feeInShare);
            feesUpdatedAt = block.timestamp;
        }
    }
    _;
}
```

The `deposit` and `mint` functions call `_validateAndUpdateDepositTimestamps` to initialize `feesUpdatedAt` to `block.timestamp` during the first deposit or mint:

```
function _validateAndUpdateDepositTimestamps(address receiver_) private {
    // [...]
    if (totalSupply() == 0) feesUpdatedAt = block.timestamp;
    // [...]
}
```

However, the `takeFees` modifier executes before `_validateAndUpdateDepositTimestamps` within both the `deposit` and `mint` functions:

```
function deposit(uint256 assets_, address receiver_)
    public
    override
    nonReentrant
    whenNotPaused
    takeFees
    returns (uint256 shares)
{
    _validateAndUpdateDepositTimestamps(receiver_);
    // [...]
}

function mint(uint256 shares_, address receiver_)
    public
    override
    nonReentrant
    whenNotPaused
    takeFees
    returns (uint256 assets)
{
    _validateAndUpdateDepositTimestamps(receiver_);
    // [...]
}
```

As a result, when `takeFees` runs for the first time, `feesUpdatedAt` remains uninitialized (i.e., zero). This causes `accruedProtocolFee` to treat the entire `block.timestamp` as elapsed time, resulting in protocol fees being calculated even though no time has actually elapsed for fee accrual. If the contract holds asset tokens at this point, the `_mint(feeRecipient, feeInShare)` call issues these unearned shares to the `feeRecipient`.

Impact

The `feeRecipient` can receive unearned shares during the first deposit or mint if the vault already holds asset tokens. This may result in an incorrect allocation of protocol fees and a dilution of other users' shares.

Recommendations

Consider refactoring `_validateAndUpdateDepositTimestamps` into a modifier, and ensure it executes before the `takeFees` modifier in the `deposit` and `mint` functions. This guarantees that `feesUpdatedAt` is properly initialized before any fee calculation or accrual logic runs.

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit `4f5b9785`.

3.5. The `protectStrategy` state variable is not updated during `emergencyRemoveStrategy` call

Target	ConcreteMultiStrategyVault		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

The `emergencyRemoveStrategy` function allows the owner of ConcreteMultiStrategyVault to remove a strategy in emergency situations.

```
function emergencyRemoveStrategy(uint256 index_, bool forceEject_)
    external onlyOwner {
    StrategyHelper.emergencyRemoveStrategy(strategies, asset(), index_,
    forceEject_, protectStrategy);
}
```

The `StrategyHelper.emergencyRemoveStrategy` function attempts to clear `protectStrategy` if the strategy being removed is the designated protected strategy. However, the `protectStrategy` function argument is passed to the library function as a memory copy, not a storage reference. As a result, any update to `protectStrategy` inside the library function does not affect the `protectStrategy` state variable in the ConcreteMultiStrategyVault contract.

```
function emergencyRemoveStrategy(
    Strategy[] storage strategies,
    address asset,
    uint256 index_,
    bool forceEject_,
    address protectStrategy
) external {
    // [...]
    if (forceEject_) {
        // [...]
        if (address(stratToRemove.strategy) == protectStrategy) {
            protectStrategy = address(0);
        }
        // [...]
    } else {
        // Normal removal process
        removeStrategy(stratToRemove.strategy, protectStrategy,
```

```
    IERC20(asset));
    }
}

function removeStrategy(IStrategy stratToBeRemoved_, address protectStrategy_,
    IERC20 asset)
    public
    returns (address protectStrategy)
{
    // [...]
    if (protectStrategy_ == address(stratToBeRemoved_)) {
        protectStrategy = address(0);
    } else {
        // [...]
    }
    // [...]
}
```

Consequently, after the owner calls `emergencyRemoveStrategy` to remove the protected strategy, the `protectStrategy` state variable remains unchanged.

Impact

When the owner removes the current `protectStrategy` using `emergencyRemoveStrategy`, the strategy's address remains in the `protectStrategy` state variable. The removed strategy retains its privileged permissions and can still call functions protected by the `onlyProtect` modifier, such as the function `requestFunds`, which could request funds from the vault.

Recommendations

Modify `StrategyHelper.emergencyRemoveStrategy` to return the updated protected strategy address. Update `ConcreteMultiStrategyVault.emergencyRemoveStrategy` to capture this return value, and update the `protectStrategy` state variable accordingly.

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit `8b5c7fd0` ↗.

3.6. Incorrect L2 sequencer uptime feed integration

Target	ConcreteOracle		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	Medium

Description

When the L2 sequencer feed (`_sequencerOracle`) is set, the `getAssetPrice` function in `ConcreteOracle` checks data freshness from the sequencer feed before consuming data from the price feed:

```
function getAssetPrice(address asset) public view override returns (uint256) {
    // [...]
        (, int256 price,, uint256 updatedAt,) = source.latestRoundData();
        bool isFreshPrice = updatedAt >= (block.timestamp
    - getGracePeriodOfAsset(asset));
        if (price > 0 && isFreshPrice && _isSequencerOracleFresh()) {
            return _normalizePrice(source, uint256(price));
        } else {
            return _getAssetPriceFromFallbackOracle(asset);
        }
    // [...]
}

function _isSequencerOracleFresh() internal view returns (bool) {
    // for L1 layers, we always return true
    if (address(_sequencerOracle) == address(0)) return true;
    // for L2 layers, we check if the price is fresh
    (,,, uint256 updatedAt,) = _sequencerOracle.latestRoundData();
    return updatedAt >= (block.timestamp - DEFAULT_GRACE_PERIOD);
}
```

However, L2 sequencer uptime feeds update only when the sequencer status changes. If the sequencer operates normally and the last update occurred long ago, `_isSequencerOracleFresh` returns false. This causes `getAssetPrice` to use the fallback oracle price instead of the primary source price, which is not the intended design.

According to the Chainlink "L2 Sequencer Uptime Feeds" documentation ↗, correct integration with L2 sequencer uptime feeds requires 1) checking the sequencer status and reverting if it is down, and 2) implementing a grace period after the sequencer restarts.

Impact

The incorrect freshness check from the sequencer feed causes `getAssetPrice` to use the fallback oracle price when it should use the primary source price. If the fallback oracle is unset, `getAssetPrice` returns 0, which causes reverts in calling functions.

Additionally, the missing sequencer status check allows `getAssetPrice` to use stale prices. When the L2 sequencer goes down, price oracles stop updating data. Stale prices can appear fresh during sequencer downtime.

Recommendations

We recommend removing the incorrect `_isSequencerOracleFresh` check and implementing the L2 sequencer uptime check specified by Chainlink.

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit `731f01d7` ↗.

3.7. Unclaimed withdrawal requests may be incorrectly finalized

Target	WithdrawalQueue, WithdrawalQueueHelper		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

When the owner of the vault batch claims withdrawal requests, the function `batchClaim` determines whether to terminate the current batch claim based on the remaining available assets and the value of `maxRequests` set by the owner. For available assets, the function `prepareWithdrawal` is expected to return the `newAvaliableAssets` as the difference between the `avaliableAssets` and the withdrawal amount when the withdrawal amount does not exceed the `avaliableAssets`; otherwise, if the withdrawal amount if greater than `avaliableAssets`, the returned `newAvaliableAssets` should be equal to `avaliableAssets`.

```
function batchClaim(
    IWithdrawalQueue withdrawalQueue,
    uint256 maxRequests,
    uint256 availableAssets,
    // [...]
) external {
    // [...]

    uint256 max = lastCreatedId < lastFinalizedId + maxRequests ?
    lastCreatedId : lastFinalizedId + maxRequests;

    for (uint256 i = lastFinalizedId + 1; i <= max;) {
        uint256 newAvailiableAssets =
            claimWithdrawal(i, availableAssets, withdrawalQueue, asset,
    strategies, parkingLot);
        // slither-disable-next-line incorrect-equality
        if (newAvailiableAssets == availableAssets) break;

        availableAssets = newAvailiableAssets;
        newLastFinalized = i;
        unchecked {
            i++;
        }
    }
    // [...]
}
```

```
function claimWithdrawal(
    uint256 _requestId,
    uint256 avaliableAssets,
    IWithdrawalQueue withdrawalQueue,
    // [...]
) public returns (uint256) {
    (address recipient, uint256 amount, uint256 newAvaliableAssets) =
        withdrawalQueue.prepareWithdrawal(_requestId, avaliableAssets);

    if (avaliableAssets != newAvaliableAssets) {
        _withdrawStrategyFunds(amount, recipient, asset, strategies,
    parkingLot);
    }
    return newAvaliableAssets;
}
```

However, the function `prepareWithdrawal` will not initialize the returned `avaliableAssets` when `_avaliableAssets` is less than `amount`.

```
function prepareWithdrawal(uint256 _requestId, uint256 _avaliableAssets)
    external
    onlyOwner
    returns (address recipient, uint256 amount, uint256 avaliableAssets)
{
    // [...]

    recipient = request.recipient;

    WithdrawalRequest storage prevRequest = _requests[_requestId - 1];

    amount = request.cumulativeAmount - prevRequest.cumulativeAmount;

    if (_avaliableAssets >= amount) {
        assert(_requestsByOwner[recipient].remove(_requestId));
        avaliableAssets = _avaliableAssets - amount;
        request.claimed = true;
        // [...]
    }
}
```

Impact

When the withdrawal amount is greater than the remaining available assets, the value of the `newAvaliableAssets` returned by the function `prepareWithdrawal` will be zero, which does not equal the `avaliableAssets` before the call.

With Finding 3.2. ↗, because the transaction will not be reverted when `totalWithdrawn < amount_` and `amount_ - totalWithdrawn > float`, the function `_withdrawStrategyFunds` may revert the transaction or may be successfully executed when `avaliableAssets` is insufficient. If the execution succeeds, the receiver can only receive part of the assets they are supposed to receive, and the request that has not been marked as claimed in the WithdrawalQueue contract may be incorrectly recorded as the last finalized ID. As a result, the next batch claim will start processing from this request's following request. This request, which has not been fully claimed, will not be executed again.

Recommendations

Consider initializing the `avaliableAssets` to the value of `_avaliableAssets` in the function `prepareWithdrawal`.

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit `d66613bb` ↗.

3.8. Incorrect swap logic in `_protocolWithdraw`

Target	MorphoVaultStrategy		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The `_protocolWithdraw` function in MorphoVaultStrategy contains two issues in its background swap logic:

```
function _protocolWithdraw(uint256 assets_, uint256)
    internal virtual override {
    // [...]
    if (isBackgroundSwapEnabled) {
        uint256 expectedOutput = UniswapV3HelperV1.getExpectedOutput(
            IQuoterV2(uniswapQuoter), _asset, address(_backgroundSwap),
    assets_, poolFee
        );
        isValidQuote(_asset, address(_backgroundSwap), assets_,
    expectedOutput);
        //@dev add slippage protection and increase the amount to withdraw by
    1%
        uint256 swapAssets = expectedOutput.mulDiv(100_00 + MAX_SLIPPAGE,
    100_00, Math.Rounding.Floor);
        uint256 _totalAssets_
    = _morphoVault.convertToAssets(_morphoVault.balanceOf(address(this)));
        uint256 assetsToWithdraw = swapAssets > _totalAssets_ ? _totalAssets_
    : swapAssets;
        //slither-disable-next-line unused-return
        _morphoVault.withdraw(assetsToWithdraw, address(this), address(this));
        uint256 swappedAmount
    = _swapExactTokenToToken(address(_backgroundSwap), _asset,
    assetsToWithdraw, 0);
        emit BackgroundSwapWithdraw(assetsToWithdraw, swappedAmount);
    } else {
        // [...]
    }
}
```

First, `UniswapV3HelperV1.getExpectedOutput` calls `quoterV2.quoteExactInputSingle` to

calculate output tokens (`_backgroundSwap`) received for `assets_` input tokens (`_asset`):

```
function getExpectedOutput(IQuoterV2 quoterV2, address tokenIn,
    address tokenOut, uint256 amountIn, uint24 poolFee)
    external
    returns (uint256 amountOut)
{
    IQuoterV2.QuoteExactInputSingleParams memory quoteExactInputSingleParams
    = IQuoterV2.QuoteExactInputSingleParams({
        tokenIn: tokenIn,
        tokenOut: tokenOut,
        amountIn: amountIn,
        fee: poolFee,
        sqrtPriceLimitX96: 0
    });
    validateFeeTier(poolFee);
    (amountOut,,,)
    = quoterV2.quoteExactInputSingle(quoteExactInputSingleParams);
}
```

However, the intended behavior in `_protocolWithdraw` requires calculating input tokens (`_backgroundSwap`) needed to obtain `assets_` amount of output tokens (`_asset`). In this case, `quoterV2.quoteExactOutputSingle` instead of `quoterV2.quoteExactInputSingle` should be used.

Second, `_protocolWithdraw` adds `MAX_SLIPPAGE` percentage to the input token amount:

```
uint256 swapAssets = expectedOutput.mulDiv(100_00 + MAX_SLIPPAGE, 100_00,
    Math.Rounding.Floor);
```

This addition is unnecessary. The quote simulates the actual swap, and both operations occur in the same transaction. No slippage occurs between simulation and execution. Instead of withdrawing extra tokens, the function should specify `minAmountOut` in `_swapExactTokenToToken`:

```
uint256 swappedAmount = _swapExactTokenToToken(address(_backgroundSwap), _
    asset, assetsToWithdraw, 0);
uint256 swappedAmount = _swapExactTokenToToken(address(_backgroundSwap), _
    asset, assetsToWithdraw, assets_);
```

Impact

Each background swap in `_protocolWithdraw` withdraws an extra `MAX_SLIPPAGE` percentage from `_morphoVault`. These excess tokens remain idle in the strategy, reducing yield potential.

Recommendations

First, add a `getExpectedInput` function to UniswapV3HelperV1 that uses `quoterV2.quoteExactOutputSingle` to calculate required input tokens for a specific output amount. Replace `getExpectedOutput` with this function in `_protocolWithdraw`.

Second, execute `_swapExactTokenToToken` with `minAmountOut` set to `assets_`, and also add an `isValidQuote` check to ensure the swap price is within a valid range.

```
uint256 swappedAmount = _swapExactTokenToToken(address(_backgroundSwap), _
    asset, assetsToWithdraw, 0);
isValidQuote(address(_backgroundSwap), _asset, assetsToWithdraw, assets_);
uint256 swappedAmount = _swapExactTokenToToken(address(_backgroundSwap), _
    asset, assetsToWithdraw, assets_);
```

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit 0a705658 ↗.

3.9. The VaultManager contract lacks the code to call the function `emergencyRemoveStrategy`

Target	VaultManager		
Category	Coding Mistakes	**Severity**	Low
Likelihood	N/A	**Impact**	Low

Description

The function `emergencyRemoveStrategy` can only be invoked by the owner of the ConcreteMultiStrategyVault contract. However, the owner — the VaultManager contract — does not implement code to call this function.

```
function emergencyRemoveStrategy(uint256 index_, bool forceEject_)
    external onlyOwner
```

Impact

This function cannot be invoked unless the VaultManager contract is upgraded to include the logic for managing it or the ownership of the ConcreteMultiStrategyVault is transferred to an account capable of invoking it.

Recommendations

Add the function `emergencyRemoveStrategy` to the VaultManager contract to call the function `emergencyRemoveStrategy` of the ConcreteMultiStrategyVault.

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit 64361970 ↗.

3.10. Incorrect transfer amount in the function `rescueFunds`

Target	MultiSigStrategyV1		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The function `rescueFunds` in the MultiSigStrategyV1 contract can rescue any assets held by this strategy. However, the amount to transfer is the underlying asset balance of the contract, instead of the `asset_` balance.

```
function rescueFunds(address asset_) external onlyOwner {
    IERC20(asset_).safeTransfer(owner(), IERC20(asset()).balanceOf(address(
        this)));
}
```

Impact

The owner needs to control the balance of the underlying asset to rescue a specific asset in the contract.

Recommendations

Consider updating the function according to the following code:

```
function rescueFunds(address asset_) external onlyOwner {
    IERC20(asset_).safeTransfer(owner(), IERC20(asset()).balanceOf(address(
        this)));
    IERC20(asset_).safeTransfer(owner(), IERC20(asset_).balanceOf(address(
        this)));
}
```

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit `e71d5bda`.

3.11. The function `redeem` lacks dust-amount check

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

Description

Functions `deposit` and `mint` check whether the amount of shares to be minted is greater than the `DUST` amount, and the function `withdraw` checks whether the amount of shares to be burned is greater than the `DUST` amount. However, the function `redeem` does not perform this check.

Impact

When a user redeems a dust amount of shares, they may receive zero assets, resulting in a loss of funds.

Recommendations

Add a check in the function `redeem` to ensure that the amount of shares is greater than `DUST`.

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit 7e38c044 ↗.

3.12. Inconsistent handling of infinite allowance in `_withdraw`

Target	ConcreteMultiStrategyVault		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

The ConcreteMultiStrategyVault contract inherits from OpenZeppelin's ERC-20 implementation, which uses `type(uint256).max` as a sentinel value to represent infinite allowance. The standard internal `_spendAllowance` function preserves this infinite approval by not decreasing the allowance when it is set to `type(uint256).max`.

```
function _spendAllowance(address owner, address spender, uint256 value)
    internal virtual {
    uint256 currentAllowance = allowance(owner, spender);
    if (currentAllowance != type(uint256).max) {
        if (currentAllowance < value) {
            revert ERC20InsufficientAllowance(spender, currentAllowance,
    value);
        }
        unchecked {
            _approve(owner, spender, currentAllowance - value, false);
        }
    }
}
```

However, the `_withdraw` function in ConcreteMultiStrategyVault does not use `_spendAllowance`. Instead, it manually calculates and reduces the allowance when an approved spender (`msg.sender != owner_`) executes a withdrawal. This approach does not check for the infinite allowance sentinel value.

```
function _withdraw(uint256 assets_, address receiver_, address owner_,
    uint256 shares, uint256 feeShares) private {
    // [...]
    if (msg.sender != owner_) {
        _approve(owner_, msg.sender, allowance(owner_, msg.sender) - shares);
    }
    // [...]
}
```

Impact

When token owners grant spenders infinite allowance, the allowance decreases if spenders withdraw assets from the vault on their behalf. This behavior is inconsistent with the `transferFrom` function, where infinite allowances remain unchanged after transfers.

Recommendations

Consider replacing the `_approve` call with the `_spendAllowance` internal function, which checks for infinite allowance before reducing the allowance.

```
function _withdraw(uint256 assets_, address receiver_, address owner_,
    uint256 shares, uint256 feeShares) private {
    // [...]
    if (msg.sender != owner_) {
        _approve(owner_, msg.sender, allowance(owner_, msg.sender) - shares);
        _spendAllowance(owner_, msg.sender, shares);
    }
    // [...]
}
```

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit 44114dac ↗.

3.13. Lack of support for the 0.01% fee tier for swaps in MorphoVaultStrategy

Target	UniswapV3HelperV1		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

MorphoVaultStrategy performs token swaps on Uniswap for `backgroundSwap` and `_autoCompoundRewards`. Prior to a swap, the `validateFeeTier` function in the UniswapV3HelperV1 library checks the pool fee:

```
function validateFeeTier(uint24 fee) public pure {
    if (fee != 500 && fee != 3000 && fee != 10000) {
        revert InvalidFeeTier();
    }
}
```

The `validateFeeTier` function accepts only a 0.05%, 0.3%, or 1% pool fee for Uniswap swaps. However, Uniswap V3 includes a 0.01% fee tier ↗ (100 basis points), particularly beneficial for stablecoin trading.

Consequently, swaps that use the 0.01% fee tier are incompatible with MorphoVaultStrategy.

Impact

MorphoVaultStrategy cannot execute swaps on Uniswap pools with the 0.01% fee tier, potentially missing optimal trading routes for stablecoin pairs.

Recommendations

We recommend updating the `validateFeeTier` function to support the 0.01% (100) fee tier:

```
function validateFeeTier(uint24 fee) public pure {
    if (fee != 500 && fee != 3000 && fee != 10000) {
    if (fee != 100 && fee != 500 && fee != 3000 && fee != 10000) {
        revert InvalidFeeTier();
    }
}
```

Remediation

This issue has been acknowledged by Blueprint Finance, and fixes were implemented in the following commits:

- 29666589
- b23b1095

3.14. Function `getAvailableAssetsForWithdrawal` should return zero when `withdrawEnabled` is false

Target	MultiSigStrategyV1		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

When `withdrawEnabled` is false, the revert in `_protocolWithdraw` prevents the Concrete Vault from withdrawing from the MultiSigStrategyV1 strategy:

```
function _protocolWithdraw(uint256 assets_, uint256)
    internal virtual override {
    if (!withdrawEnabled) revert WithdrawDisabled();
    // [...]
}
```

Therefore, `getAvailableAssetsForWithdrawal` should return 0 in this case. However, it currently returns `_vaultDepositedAmount`:

```
function getAvailableAssetsForWithdrawal()
    public view override returns (uint256) {
    if (!withdrawEnabled) return _vaultDepositedAmount;
    // [...]
}
```

Impact

For a Concrete Vault using the MultiSigStrategyV1 strategy, if `withdrawEnabled` is false, the vault receives an incorrect estimate of the withdrawable amount from the strategies.

Recommendations

Update the `getAvailableAssetsForWithdrawal` function to return zero when `withdrawEnabled` is false.

```
function getAvailableAssetsForWithdrawal()
    public view override returns (uint256) {
    if (!withdrawEnabled) return _vaultDepositedAmount;
    if (!withdrawEnabled) return 0;
    // [...]
}
```

Remediation

This issue has been acknowledged by Blueprint Finance.

Blueprint Finance provided the following response to this finding:

> We intentionally return _vaultDepositedAmount when withdrawEnabled is false. This was originally a workaround to force withdrawals to go through the queue, since some vaults weren't upgradeable.
>
> This behavior only applies when multiSigStrategy is the sole strategy in the vault, and we'd prefer to keep it as-is for backward compatibility.

3.15. Function `changeAllocations` does not check `vaultIdle` status before redistributing assets

Target	ConcreteMultiStrategyVault		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `changeAllocations` function in the ConcreteMultiStrategyVault contract does not check the `vaultIdle` status before redistributing assets:

```
function changeAllocations(
    Strategy[] storage strategies,
    Allocation[] calldata allocations_,
    bool redistribute,
    address asset
) external {
    // [...]
    if (redistribute) {
        pullFundsFromStrategies(strategies);
        distributeAssetsToStrategies(strategies,
    IERC20(asset).balanceOf(address(this)));
    }
    // [...]
}
```

According to the contract's design, setting `vaultIdle` to `true` should prevent all deposits into underlying strategies. However, when `changeAllocations` is called with the `redistribute` parameter set to `true`, the function proceeds to call `distributeAssetsToStrategies` without this validation.

Impact

The `changeAllocations` function allows deposits into strategies even when `vaultIdle` is true, bypassing the intended idle vault restrictions.

Recommendations

We recommend adding the `vaultIdle` check within the `changeAllocations` function before assets are redistributed.

```
function changeAllocations(
    Strategy[] storage strategies,
    Allocation[] calldata allocations_,
    bool redistribute,
    address asset
) external {
    // [...]
    if (redistribute) {
        pullFundsFromStrategies(strategies);
       if (vaultIdle) revert("vaultIdle");
        distributeAssetsToStrategies(strategies,
    IERC20(asset).balanceOf(address(this)));
    }
    // [...]
}
```

Remediation

This issue has been acknowledged by Blueprint Finance.

Blueprint Finance provided the following response to this finding:

> We haven't used the vaultIdle flag so far and it appears to be redundant in our current logic. The changeAllocations() function isn't intended to interact with vaultIdle, so we're choosing to skip this fix.

3.16. Incorrect rounding direction in `previewMint`

Target	ConcreteMultiStrategyVault		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The `previewMint` function in the ConcreteMultiStrategyVault contract uses `Rounding.Floor` as the rounding direction for `grossShares` after fee calculation and in `_convertToAssets`:

```
function previewMint(uint256 shares_) public view override returns (uint256) {
    uint256 grossShares = shares_.mulDiv(MAX_BASIS_POINTS, MAX_BASIS_POINTS
    - fees.depositFee, Math.Rounding.Floor);
    return _convertToAssets(grossShares, Math.Rounding.Floor);
}
```

However, the `mint` function uses `Rounding.Ceil` for both operations:

```
function mint(uint256 shares_, address receiver_)
    public
    override
    nonReentrant
    whenNotPaused
    takeFees
    returns (uint256 assets)
{
    // [...]
    uint256 feeShares =
        shares_.mulDiv(MAX_BASIS_POINTS, MAX_BASIS_POINTS - depositFee,
    Math.Rounding.Ceil) - shares_;
    // [...]
    assets = _convertToAssets(shares_ + feeShares, Math.Rounding.Ceil);
    // [...]
}
```

Impact

This inconsistency in rounding direction causes `previewMint` to return inaccurate mint previews.

Recommendations

Update the rounding direction in `previewMint` to match the `mint` function's use of `Rounding.Ceil`.

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit `c58be473` ↗.

3.17. The fees charged to the `feeRecipient` are inconsistent between functions `deposit` and `mint`

Target	ConcreteMultiStrategyVault		
Category	Business Logic	**Severity**	Informational
Likelihood	High	**Impact**	Informational

Description

Users can choose to use either the function `deposit` or the function `mint` to deposit an asset to the vault. However, the methods by which functions `deposit` and `mint` charge the `feeRecipient` are inconsistent. The function `deposit` does not charge a deposit fee when the caller is the `feeRecipient`, while the function `mint` always charges a deposit fee regardless of the caller.

```
function deposit(uint256 assets_, address receiver_)
    // [...]
    returns (uint256 shares)
{
    // [...]
    // Calculate shares based on whether sender is fee recipient
    if (msg.sender == feeRecipient) {
        shares = _convertToShares(assets_, Math.Rounding.Floor);
    } else {
        // Calculate the fee in shares
        uint256 feeShares = _convertToShares(
            assets_.mulDiv(uint256(fees.depositFee), MAX_BASIS_POINTS,
    Math.Rounding.Ceil), Math.Rounding.Ceil
        );

        // Calculate the net shares to mint for the deposited assets
        shares = _convertToShares(assets_, Math.Rounding.Floor) - feeShares;

        // Mint fee shares to fee recipient
        if (feeShares > 0) _mint(feeRecipient, feeShares);
    }
    // [...]
}
```

```
function mint(uint256 shares_, address receiver_)
    // [...]
```

```
    returns (uint256 assets)
{
    // [...]

    // Calculate the deposit fee in shares
    uint256 depositFee = uint256(fees.depositFee);
    uint256 feeShares =
        shares_.mulDiv(MAX_BASIS_POINTS, MAX_BASIS_POINTS - depositFee,
    Math.Rounding.Ceil) - shares_;

    // Calculate the total assets required for the minted shares, including
    fees
    assets = _convertToAssets(shares_ + feeShares, Math.Rounding.Ceil);
    // [...]
}
```

Impact

If the `feeRecipient` is depositing on behalf of a receiver, then, using the same amount of the asset, the receiver will receive more shares if the function `deposit` is used.

Recommendations

Consider unifying the way these two functions charge the `feeRecipient` and updating functions `previewDeposit` and `previewMint` accordingly.

Remediation

This issue has been acknowledged by Blueprint Finance, and fixes were implemented in the following commits:

- f3c41a5d ↗
- b23b1095 ↗

3.18. The BTCLinkedPriceFeed contract does not initialize the owner

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

Description

The BTCLinkedPriceFeed contract inherits from the OwnableUpgradeable contract, but it does not call the function `__Ownable_init` during the initialization to initialize the owner.

Impact

The owner address will be the zero address after initialization. If there are functions with the `onlyOwner` modifier planned to be added into this contract, it is recommended to fix this issue.

Recommendations

Consider adding a call to the function `__Ownable_init` in the function `initialize` or removing the inherited contract OwnableUpgradeable.

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit `4d72bd74` ↗.

3.19. Insufficient validation of `_requestId`

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

Description

The storage variable `lastFinalizedRequestId` records the last claimed request. And the function `prepareWithdrawal` is used to process unclaimed requests. Thus, this function should not process a request when its `_requestId` equals the `lastFinalizedRequestId`.

```
function prepareWithdrawal(uint256 _requestId, uint256 _avaliableAssets)
    external
    onlyOwner
    returns (address recipient, uint256 amount, uint256 avaliableAssets)
{
    if (_requestId == 0) revert InvalidRequestId(_requestId);

    if (_requestId < lastFinalizedRequestId) revert RequestNotFoundOrFinalized(
        _requestId);

    WithdrawalRequest storage request = _requests[_requestId];

    if (request.claimed) revert RequestAlreadyClaimed(_requestId);

    // [...]
}
```

Impact

If each request can be correctly marked as claimed or not, not checking whether the `_requestId` equals `lastFinalizedRequestId` will not have much impact — it would only cause the transaction to revert a bit later.

However, the vault has a function `claimWithdrawal`, which can claim any withdrawal request with the provided `_requestId`. Although it is currently a private function, if it were to be made public in the future and there exists a request that was incorrectly finalized but not yet marked as claimed (see Finding 3.7. ↗ for details), this request could be executed again through the function `prepareWithdrawal`. As a result, the receiver would receive more assets than intended, due to the

combination of assets sent during the erroneous finalization and the re-execution.

Recommendations

Consider updating the check according to the following code:

```
if (_requestId < lastFinalizedRequestId) revert RequestNotFoundOrFinalized(_
    requestId);
if (_requestId <= lastFinalizedRequestId) revert RequestNotFoundOrFinalized(_
    requestId);
```

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit 574ea328 ↗.

3.20. Unchecked return value in `setParkingLot`

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

Description

The `setParkingLot` function in the ConcreteMultiStrategyVault contract does not check the `successfulApproval` boolean returned by `TokenHelper.attemptForceApprove`. If the approval fails, subsequent deposits to the parking lot will revert.

```
function setParkingLot(address parkingLot_) external onlyOwner {
    // [...]
    bool successfulApproval = TokenHelper.attemptForceApprove(token,
    parkingLot_, type(uint256).max, false);
    emit ParkingLotUpdated(currentParkingLot, parkingLot_,
    successfulApproval);

    parkingLot = IParkingLot(parkingLot_); // Update the fee recipient
}
```

Impact

If the approval fails, the contract still updates `parkingLot` to the new address. Since the vault requires token approval to deposit into the parking lot, all future parking-lot deposit attempts will fail.

Recommendations

Verify that the approval succeeds.

```
function setParkingLot(address parkingLot_) external onlyOwner {
    // [...]
    bool successfulApproval = TokenHelper.attemptForceApprove(token,
    parkingLot_, type(uint256).max, false);
    require(successfulApproval, "Approve failed");
    // [...]
}
```

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit 7b237030 ↗.

3.21. Incorrect withdrawable check in `_withdrawStrategyFunds`

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

Description

The `_withdrawStrategyFunds` function in the WithdrawalQueueHelper library contains two issues in its withdrawable amount validation:

```
function _withdrawStrategyFunds(
    uint256 amount_,
    address receiver_,
    address asset_,
    Strategy[] memory strategies,
    IParkingLot parkingLot
) internal {
    // [...]
            uint256 withdrawable =
    strategy.strategy.previewRedeem(strategy.strategy.balanceOf(address(this)));
            if (diff.mulDiv(strategy.allocation.amount, MAX_BASIS_POINTS,
    Math.Rounding.Floor) > withdrawable) {
                revert Errors.InsufficientFunds(strategy.strategy, diff
    * strategy.allocation.amount, withdrawable);
            }
            uint256 amountToWithdraw = _calculateWithdrawalAmount(amount_,
    strategy);
    // [...]
}
```

First, the code compares `withdrawable` against a proportional amount calculated from `diff` (which equals `amount_ - float`) rather than from `amount_` itself.

Second, the code uses the `previewRedeem` method to determine the withdrawable amount. This method may not accurately reflect withdrawal limits. The `getAvailableAssetsForWithdrawal` method provides a more accurate withdrawable amount.

Impact

Since `diff` can be less than `amount_`, the withdrawable check may fail to detect insufficient funds for withdrawal. In this case, the execution would fail in the actual withdrawal process and spend more gas.

Recommendations

Check the `withdrawable` amount against `amountToWithdraw` rather than the calculated proportional diff amount. Additionally, use the `getAvailableAssetsForWithdrawal` method instead of `previewRedeem` to obtain the withdrawable amount.

```
function _withdrawStrategyFunds(
    uint256 amount_,
    address receiver_,
    address asset_,
    Strategy[] memory strategies,
    IParkingLot parkingLot
) internal {
    // [...]
          uint256 withdrawable = strategy.strategy.previewRedeem(strategy.
              strategy.balanceOf(address(this)));

    uint256 withdrawable = strategy.strategy.getAvailableAssetsForWithd
              rawal();

    if (diff.mulDiv(strategy.allocation.amount, MAX_BASIS_POINTS, Math.
              Rounding.Floor) > withdrawable) {

    revert Errors.InsufficientFunds(strategy.strategy, diff * strategy.
                  allocation.amount, withdrawable);
          }
          uint256 amountToWithdraw = _calculateWithdrawalAmount(amount_,
    strategy);

    if (amountToWithdraw > withdrawable) revert Errors.InsufficientFunds(
              strategy.strategy, amountToWithdraw, withdrawable);
    // [...]
}
```

Remediation

This issue has been acknowledged by Blueprint Finance, and fixes were implemented in the following commits:

- 236f073f ↗
- 4bc00f6a ↗

3.22. ConcreteMultiStrategy should inherit from ReentrancyGuardUpgradeable instead of ReentrancyGuard

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

Description

The ConcreteMultiStrategy contract deploys via clone, and its state initializes in the `initialize` function. Therefore, it should inherit from ReentrancyGuardUpgradeable instead of ReentrancyGuard.

Impact

The `_status` storage variable in ConcreteMultiStrategy remains uninitialized during the `initialize` call. Although it will be correctly set to `NOT_ENTERED` after the first invocation of the `nonReentrant` modifier, it is best practice to inherit from ReentrancyGuardUpgradeable.

Recommendations

ConcreteMultiStrategy should inherit ReentrancyGuardUpgradeable instead of ReentrancyGuard and initialize the `_status` storage variable in the `initialize` function.

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit `0884cb28` ↗.

3.23. Incorrect implementation of `max` functions in ConcreteMultiStrategy

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

Description

ConcreteMultiStrategy implements the `max` function family incorrectly according to the ERC-4626 standard.

First, `maxMint` returns depositable assets instead of mintable shares:

```
function maxMint(address) public view override returns (uint256) {
    return (paused() || totalAssets() >= depositLimit) ? 0 : depositLimit
    - totalAssets();
}
```

Second, `maxDeposit` does not verify the paused status or deposit limit.

Third, `maxWithdraw` subtracts all pending fees instead of the user's proportional share:

```
function maxWithdraw(address owner)
    public view virtual override returns (uint256) {
    // Get the raw max withdrawal amount
    uint256 rawMaxWithdraw = _convertToAssets(balanceOf(owner),
    Math.Rounding.Floor);

    // Calculate pending fees
    uint256 pendingFees = accruedProtocolFee();

    // Return max withdraw minus pending fees
    return rawMaxWithdraw.mulDiv(MAX_BASIS_POINTS - fees.withdrawalFee,
    MAX_BASIS_POINTS, Math.Rounding.Floor)
        - pendingFees;
}
```

Fourth, `maxWithdraw` accounts for pending fees while `maxRedeem` does not. Neither function accounts for performance fees.

Fifth, `maxWithdraw` accounts for withdrawal fees while `maxRedeem` does not.

Impact

The `max` function implementations deviate from ERC-4626 specifications, potentially causing integration issues with protocols expecting standard behavior.

Recommendations

Implement all `max` functions according to ERC-4626 specifications:

- The `maxMint` should return the maximum mintable shares.
- The `maxDeposit` should verify paused status and deposit limits.
- Fee calculations should apply proportionally to user shares.
- All relevant fees should be consistently accounted for across related functions.

Remediation

This issue has been acknowledged by Blueprint Finance, and a partial fix was implemented in commit f5b133d9 ↗.

Blueprint Finance partially resolved this finding and provided the following response:

> maxRedeem() returns the number of shares that can be redeemed without reverting, so it does not account for fees.
>
> maxWithdraw() returns the maximum amount of the underlying asset that can be withdrawn from the owner's balance, and does account for fees. This behavior is intentional and consistent with the ERC-4626 spec.

3.24. Strategy rewards accrue after the share price has been calculated

Target	StrategyBase		
Category	Business Logic	**Severity**	Informational
Likelihood	Medium	**Impact**	Informational

Description

During the withdrawal process, the vault may withdraw assets from strategies (see section 5.2. ↗ for more details). When withdrawing from a strategy, the strategy will call the function `_handleRewardsOnWithdraw` to accrue rewards, which may increase the amount of underlying assets held by the strategy, i.e. result in a change to the return value of the function `totalAssets`.

```
function _withdraw(address caller_, address receiver_, address owner_,
    uint256 assets_, uint256 shares_)
    internal
    virtual
    override(ERC4626Upgradeable)
    onlyVault
{
    // [...]
    _handleRewardsOnWithdraw();
    // [...]
}
```

Impact

Since the reward is accrued after the `assets_` and `shares_` have been calculated, if the accrued reward contains the underlying asset, it is not considered when calculating the share price for this withdrawal. As a result, the share price used when calculating `assets_` and `shares_` may be lower than the share price after rewards have accrued, which could result in users receiving fewer assets.

Recommendations

Consider harvesting rewards from the strategies before calculating the amount of assets to withdraw or the amount of shares to burn.

Remediation

This issue has been acknowledged by Blueprint Finance.

Blueprint Finance provided the following response to this finding:

> We acknowledge this issue, but do not plan to fix it for the following reasons:
>
> 1. Adding `_handleRewardsOnWithdraw()` in the withdraw/redeem before calculating assets will require another call and that would increase gas costs.
> 2. We currently have no use case where rewards are compounded as the same token during `_handleRewardsOnWithdraw()`.
> 3. A bot already calls `vault.harvestRewards()` every 30 minutes, effectively handling reward accruals. we can solve this by increasing the bot frequency as well.
> 4. It will be too much restructuring to implement this change in both strategy and vault codebase.
>
> Given the limited benefit and added complexity, we prefer to leave this as-is.

3.25. Incorrect use of the function `_getRewardTokens` to initialize the `rewardTokens`

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

Description

The constructor of the MorphoVaultStrategy contract tries to generate a reward-token array by using the function `_getRewardTokens`.

```
constructor(ConstructorParams memory params) {
    // [...]
    __StrategyBase_init(
        params.baseAsset,
        string.concat("Concrete Morpho Vault ", symbol, " Strategy"),
        string.concat("ctMV1-", symbol),
        params.feeRecipient,
        type(uint256).max,
        params.owner,
        _getRewardTokens(params.rewardFee),
        params.vault
    );

    // [...]
}
```

However, this function's returned array is based on the length of the storage variable `rewardTokens`, which is empty during deployment.

```
function _getRewardTokens(uint256 rewardFee_)
    internal view returns (RewardToken[] memory) {
    address[] memory rewards = getRewardTokenAddresses();
    RewardToken[] memory r = new RewardToken[](rewards.length);
    // [...]
    return r;
}

function getRewardTokenAddresses() public view virtual returns (address[]
```

```
    memory) {
    //Each strategy should avoid returning the token considered in the
    _totalAssets function as a reward token
    uint256 len = rewardTokens.length;
    address[] memory rT = new address[](len);
    // [..]
    return rT;
}
```

Impact

This means that the owner needs to add the reward token separately via the function `addRewardToken` after deployment, which should not be required if the deployment is correct.

Recommendations

Consider implementing appropriate logic to generate the reward-token array during deployment.

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit `7bec80ce` ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. The `ConcreteOracle.getAssetPrice` should ensure the returned price uses eight decimals

The `getAssetPrice` function has three main execution paths for returning a price:

1. **Primary oracle path.** It queries the price from the `source` oracle and normalizes it to eight decimals.

```
return _normalizePrice(source, uint256(price));
```

2. **Base currency path.** If the requested `asset` is the `BASE_CURRENCY`, the function returns the `BASE_CURRENCY_UNIT` constant directly.

```
// Assumes BASE_CURRENCY_UNIT is already 10**8
if (asset == BASE_CURRENCY) {
    return BASE_CURRENCY_UNIT;
}
```

3. **Fallback oracle path.** If the primary source is missing, the price is stale, or the sequencer is offline, the function returns the price from the fallback oracle.

```
return _getAssetPriceFromFallbackOracle(asset);
```

The primary path normalizes the price to ensure it uses eight decimals:

```
function _normalizePrice(AggregatorV3Interface source, uint256 _price)
    internal view returns (uint256) {
    uint256 decimals = source.decimals();
    if (decimals == 8) {
        return _price;
    } else if (decimals > 8) {
        return _price / (10 ** (decimals - 8));
    } else {
        return _price * (10 ** (8 - decimals));
    }
}
```

However, the other two paths do not guarantee that the returned price uses eight decimals. It is recommended to ensure that prices from these paths also use eight decimals.

4.2. Unused `_decimals` variable in strategies

The StrategyBase contract includes a `_decimals` storage variable, which is initialized to be nine greater than the asset's original decimals.

```
uint8 public _decimals;

uint8 public constant DECIMAL_OFFSET = 9;

function __StrategyBase_init(
    IERC20 baseAsset_,
    string memory shareName_,
    string memory shareSymbol_,
    address feeRecipient_,
    uint256 depositLimit_,
    address owner_,
    RewardToken[] memory rewardTokens_,
    address vault_
) internal nonReentrant initializer {
    // [...]
    _decimals = IERC20Metadata(address(baseAsset_)).decimals()
    + DECIMAL_OFFSET;
    // [...]
}
```

However, in the strategies covered by the current audit (AaveV3Strategy, MorphoVaultStrategy, and MultiSigStrategyV1), the `_decimals` variable is not used. These strategies do not override the `decimals`, `_convertToShares`, or `_convertToAssets` functions to utilize `_decimals` and `DECIMAL_OFFSET`, as demonstrated in the ExampleStrategyBaseImplementation contract.

If `_decimals` is not intended for use, consider removing this variable to prevent confusion in the code.

4.3. Test suite

The test suite of this project uses the `testFail` test prefix to handle negative test cases.

Since the `testFail` prefix support was removed in Foundry V1.0, we recommend using the

`expectRevert` cheat code to ensure that the transaction reverts with the correct error message.

5. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

5.1. Component: Controller

Registries, factories, and managers can be categorized as the controllers. They are capable of deploying and managing specific contracts.

Registry

The ImplementationRegistry contract maintains a whitelist of implementation contract addresses. And the VaultRegistry contract holds a whitelist of vault instance addresses, with a default cap of 1,000 vaults. The VaultRegistry contract also maintains information regarding the vault's underlying asset and its implementation.

Contract addresses can be added to or removed from both registries through the VaultManager contract.

Factory

Based on the implementation information provided by the owner, the VaultFactory contract can deploy new vault instances by cloning the implementation or can deploy new upgradable vault instances using ERC1967Proxy and `CREATE2`.

The OracleFactory contract itself manages a whitelist of oracle implementation contract addresses. The owner of the OracleFactory contract can deploy new oracle instances by cloning whitelisted implementations.

Manager

The DeploymentManager contract manages the VaultFactory, VaultRegistry, and ImplementationRegistry contracts, handling the deployment and registration of new vaults, along with the addition or removal of their implementation contracts. To deploy a new vault, the DeploymentManager first retrieves and validates the implementation address from the ImplementationRegistry contract, then deploys the vault via the VaultFactory contract and adds the new vault address to the VaultRegistry contract.

The VaultManager contract is the owner of the DeploymentManager contract and manages the administrative functions for multiple vaults. Specifically, after deploying a new vault via the DeploymentManager contract, it will deploy and configure the dedicated withdrawal component

for each vault — namely, the WithdrawalQueue contract and the ParkingLot contract. For more details about the withdrawal component, please refer to section 5.3. ↗. Additionally, when upgrading a vault, before calling the function `upgradeToAndCall` to execute the upgrade, the VaultManager contract first calls the function `upgradeVault` of the DeploymentManager contract to verify the upgradability of the vault and the validity of the new implementation and to update the vault's associated entries in the VaultRegistry contract.

5.2. Component: ConcreteMultiStrategyVault

The ConcreteMultiStrategyVault contract is an ERC-4626–compliant vault that can manage multiple yield-generating strategies. The owner of the vault can add or remove strategies and can set the proportion of the vault's underlying asset allocated to each strategy. Users who deposit into the vault receive shares. The vault offers a one-stop solution for shareholders, enabling automatic yield optimization and liquidity protection.

Deposit

When a user deposits assets into the vault via the function `deposit` or the function `mint`, the vault mints a certain amount of shares for the user based on the amounts of total assets and total shares. However, if the amount of shares to be minted is less than `DUST`, the transaction will be reverted.

If the vault is not idle, the vault will allocate the user's deposited assets to all registered strategies according to the configured allocation ratios. Different strategies handle deposits from the vault in different ways; refer to section 5.4. ↗ for more details.

```
function deposit(uint256 assets_, address receiver_)
    // [...]
    returns (uint256 shares)
{
    // [...]

    // Handle strategy allocation if vault is not idle
    if (!vaultIdle) {
        StrategyHelper.depositIntoStrategies(strategies, assets_,
    address(this), true);
    }
    emit Deposit(msg.sender, receiver_, assets_, shares);
}

function depositIntoStrategies(
    Strategy[] memory strategies,
    uint256 assets_,
    address vaultAddress,
    bool isRoundingFloor
) external {
```

```
        // [...]
        uint256 len = strategies.length;
        for (uint256 i; i < len;) {
            // [...]
            strategies[i].strategy.deposit(
                assets_.mulDiv(strategies[i].allocation.amount, MAX_BASIS_POINTS,
    rounding), vaultAddress
            );
            unchecked {
                i++;
            }
        }
    }
```

Withdrawal

Users can choose either the function `redeem` or `withdraw` to burn shares in exchange for the underlying asset when the withdrawal is not paused. If the amount of shares to be burned is less than `DUST`, only the function `withdraw` will revert the transaction.

Because the underlying assets are distributed in the vault and various strategies, and different strategies handle the underlying assets in different ways, some underlying assets may not be immediately withdrawable. Therefore, there are two possible scenarios when a user withdraws:

1. If the total withdrawable assets are sufficient to cover the amount the user wants to withdraw, the vault will first use the assets it holds and then withdraw the remaining amount from the strategies according to the configured allocations, before sending the assets to the receiver.
2. If the total withdrawable assets are insufficient to cover the amount the user wants to withdraw, the vault will not send any assets to the receiver in this transaction. Instead, the vault will create a withdrawal request and place it into the withdrawal queue. Afterwards, when there are sufficient withdrawable assets, the owner of the vault can claim the withdrawal requests. Except for consuming the requests in the withdrawal queue, the method of transferring assets to the receiver is the same as in the first scenario.

For withdrawals involving assets that need to be withdrawn from strategies, the receiver may receive an amount of assets less than expected. For details, please refer to Finding 3.2. ↗.

Reward

The owner of the vault can call the function `harvestRewards` to harvest rewards on every strategy. The function `harvestRewards` of the strategy transfers the rewards to the vault and returns an array recording the reward-token addresses and the effective amounts sent (excluding the

underlying asset). The vault then updates the `rewardIndex` for each reward token based on the effective reward amount and the total shares of the vault.

```
function harvestRewards(bytes calldata encodedData)
    external nonReentrant onlyOwner {
    // [...]
    for (uint256 i; i < lenStrategies;) {
        // [...]
        ReturnedRewards[] memory returnedRewards
    = strategies[i].strategy.harvestRewards(rewardsData);
        lenRewards = returnedRewards.length;
        for (uint256 j; j < lenRewards;) {
            uint256 amount = returnedRewards[j].rewardAmount;
            address rewardToken = returnedRewards[j].rewardAddress;
            if (amount != 0) {
                if (rewardIndex[rewardToken] == 0) {
                    rewardAddresses.push(rewardToken);
                }
                if (totalSupply > 0) {
                    rewardIndex[rewardToken] += amount.mulDiv(PRECISION,
    totalSupply, Math.Rounding.Floor);
                }
            }
    // [...]
}
```

Before each user-balance update, rewards are distributed to the user based on the amount of shares they hold and the difference between their `userRewardIndex_` and the current `rewardIndex_` for each reward token. After distribution, the `userRewardIndex_` is updated to the current vault's `rewardIndex_`.

```
function updateUserRewardsToCurrent(
    uint256 userBalance_,
    address userAddress_,
    address[] memory rewardAddresses_,
    mapping(address => uint256) storage rewardIndex_,
    mapping(address => mapping(address => uint256)) storage userRewardIndex_,
    mapping(address => mapping(address => uint256))
    storage totalRewardsClaimed_
) external {
    uint256 len = rewardAddresses_.length;
    for (uint256 i; i < len;) {
        uint256 tokenRewardIndex = rewardIndex_[rewardAddresses_[i]];
        uint256 _userRewardIndex
    = userRewardIndex_[userAddress_][rewardAddresses_[i]];
        userRewardIndex_[userAddress_][rewardAddresses_[i]]
```

```
    = tokenRewardIndex;
        if (userBalance_ != 0) {
            uint256 rewardsToTransfer =
                (tokenRewardIndex - _userRewardIndex).mulDiv(userBalance_,
    PRECISION, Math.Rounding.Floor);
            if (rewardsToTransfer != 0) {
                TokenHelper.attemptSafeTransfer(
                    address(rewardAddresses_[i]), userAddress_,
    rewardsToTransfer, false
                );
        // [...]
    }
}
```

5.3. Component: Withdrawal

During the withdrawal process, when there are not enough withdrawable assets, the WithdrawalQueue contract is used to store and manage pending withdrawal requests. When transferring assets to the receiver fails, the ParkingLot contract is used to temporarily hold the withdrawn assets.

WithdrawalQueue

When the vault calls the function `requestWithdrawal`, a new `WithdrawalRequest` is pushed into the array `_requests`, which records the following:

- `cumulativeAmount` — The sum of all assets submitted for withdrawals, including this request. The amount for this withdrawal request can be derived by calculating the difference between its `cumulativeAmount` and that of the previous request.
- `recipient` — The address that can receive the funds.
- `timestamp` — The `block.timestamp` when the request is created.
- `claimed` — This indicates whether the request has been claimed.

The vault owner claims withdrawal requests through the function `batchClaimWithdrawal`, which uses the function `batchClaim` of the WithdrawalQueueHelper library to process requests sequentially, starting from the first unclaimed request in the order they were added to the array `_requests`. During this process, the function `prepareWithdrawal` of the WithdrawalQueue contract is used to retrieve the `recipient` and `amount` for each withdrawal request and to check whether the current `_avaliableAssets` are sufficient to fulfill the request. If so, it returns the remaining available assets after processing this request; otherwise, it returns an uninitialized `avaliableAssets` (see Finding 3.7. ↗ for details).

```
function prepareWithdrawal(uint256 _requestId, uint256 _avaliableAssets)
    external
```

```
    onlyOwner
    returns (address recipient, uint256 amount, uint256 avaliableAssets)
{
    // [...]
    amount = request.cumulativeAmount - prevRequest.cumulativeAmount;

    if (_avaliableAssets >= amount) {
        assert(_requestsByOwner[recipient].remove(_requestId));
        avaliableAssets = _avaliableAssets - amount;
        request.claimed = true;
        // [...]
    }
}
```

When the available assets are insufficient or the number of processed requests reaches the maximum value set by the owner, the function `batchClaim` will call the function `_finalize` of the WithdrawalQueue contract to update the storage variable `lastFinalizedRequestId`, which records the last claimed request ID.

ParkingLot

When transferring assets to the receiver fails, the vault will deposit the withdrawn assets into the ParkingLot contract. This contract records the total amount of assets each receiver is entitled to receive, as well as the timestamp of each receiver's most recent deposit by the vault.

Users can withdraw temporarily stored assets at any time through the function `withdraw` of the ParkingLot contract. However, if the assets remain unclaimed for more than one year since the last deposit, the `_rescuer` can withdraw them using the function `rescueFunds`.

5.4. Component: Strategies

Different strategies offer distinct approaches to managing assets and generating yields. They all inherit from the StrategyBase contract, which implements the basic functionality of a strategy. The basic implemented functions include managing the reward token and configuring the fee settings as well as fundamental interactions when a vault deposits to the strategy, withdraws from the strategy, or harvests rewards, and so on. Each strategy can overwrite functions `_protocolDeposit` and `_protocolWithdraw` to implement specific ways of handling the asset during deposits and withdrawals. However, there are some nongeneric functions that must be implemented by each individual strategy, such as the following:

- `_totalAssets` — This function allows each strategy to implement its own custom logic for retrieving the total amount of underlying assets based on its implementation.
- `_handleRewardsOnWithdraw` — This function is called during a withdrawal operation to accrue rewards before executing the function `_protocolWithdraw`.
- `_getRewardsToStrategy` — This function is used to collect rewards into the strategy.

AaveV3Strategy

This strategy earns yield through the Aave V3 protocol. It supplies the deposited assets to the Aave liquidity pool, and it retrieves the assets from the pool on withdrawal. Rewards can be collected by calling the function `claimAllRewards` on the `aaveIncentives`.

MorphoVaultStrategy

This strategy earns yield through the Morpho Vaults. It forwards deposits to a Morpho Vault and retrieves them on withdrawal.

If the `isBackgroundSwapEnabled` is true, then the underlying asset of the Morpho Vault is not the underlying asset of the strategy. So, before depositing into the Morpho Vault, the underlying asset being deposited will be swapped into the Morpho Vault's underlying asset via Uniswap V3. And the Morpho Vault's underlying asset will be swapped back into the strategy's underlying asset during withdrawal.

The function `_getRewardsToStrategy` can claim rewards from Morpho Universal Rewards Distributor contracts with the vault's owner-provided data. And if the `isAutoCompoundingEnabled` is true and the `morphoRewardToken` balance of the strategy is not less than the `minRewardAmountForCompounding`, the `morphoRewardToken` will be swapped into the strategy's underlying asset, and then these assets will be deposited into the Morpho Vault via the function `_protocolDeposit`. The function `_handleRewardsOnWithdraw` of this strategy cannot claim rewards; it will only compound rewards if the `isAutoCompoundingEnabled` is true.

MultiSigStrategy

This strategy simply forwards deposits to a multi-sig wallet and retrieves them on withdrawal. It does not generate any rewards. If the `withdrawEnabled` is true, withdrawals are not allowed from this strategy.

The owner can rescue any assets held by this strategy, including the underlying asset.

6. Assessment Results

During our assessment on the scoped Concrete contracts, we discovered 25 findings. No critical issues were found. Two findings were of high impact, six were of medium impact, eight were of low impact, and the remaining findings were informational in nature.

At the time of our assessment, the ConcreteMultiStrategyVault was deployed to several networks, including Ethereum, Corn, Morph, and Berachain. The other reviewed code was not yet deployed.

6.1. 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.