Zellic

**Prepared for**
Luke
Eddy
Mitosis

**Prepared by**
Juchang Lee
Kuilin Li
Zellic

**October 16, 2024**

# Mitosis EOL
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 Mitosis from September 23rd to October 3rd, 2024. During this engagement, Zellic reviewed Mitosis EOL'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:

- Could an attacker steal liquidity in the vault?
- Could an attacker make the contracts DOS?
- Could a strategy executor call the wrong strategy?
- Is the strategy implemented correctly?

## 1.3. Non-goals and Limitations

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

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

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

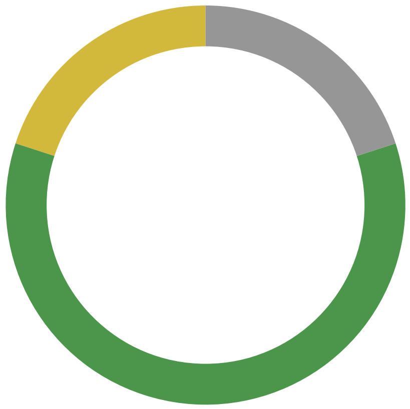
## 1.4. Results

During our assessment on the scoped Mitosis EOL contracts, we discovered five findings. No critical issues were found. One finding was of medium impact, three were of low impact, and the remaining finding was informational in nature.

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

## Breakdown of Finding Impacts

| Impact Level | Count |
|---|---|
| 🟥 Critical | 0 |
| 🟧 High | 0 |
| 🟨 Medium | 1 |
| 🟩 Low | 3 |
| ⬜ Informational | 1 |



# 2. Introduction

## 2.1. About Mitosis EOL

Mitosis contributed the following description of Mitosis EOL:

> Mitosis is an innovative Layer 1 blockchain platform optimizing on-chain liquidity and yield sourcing in DeFi. It implements Ecosystem-Owned Liquidity (EOL) across multiple chains, features miAsset for governance and rewards, and leverages Ethereum's security. Mitosis aims to revolutionize liquidity management in multi-chain DeFi, creating an efficient marketplace for all participants.

## 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:

### Mitosis EOL Contracts

| | |
|---|---|
| **Type** | Solidity |
| **Platform** | EVM-compatible |

| | |
|---|---|
| **Target** | evm |
| **Repository** | https://github.com/mitosis-org/evm ↗ |
| **Version** | `f736c4185c7b94bc384fe19832318d1657584b0e` |
| **Programs** | `hooks/*`<br>`vault/storage/BasicVaultStorageV2.sol`<br>`vault/BasicVault.sol`<br>`strategy/storage/*`<br>`strategy/strategies/*`<br>`strategy/StrategyExecutor.sol`<br>`lib/RedeemQueue.sol` |

## 2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 2.6 person-weeks. The assessment was conducted by two consultants over the course of 1.8 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:

**Juchang Lee**
Engineer
lee@zellic.io ↗

**Kuilin Li**
Engineer
kuilin@zellic.io ↗

## 2.5. Project Timeline

The key dates of the engagement are detailed below.

| | |
|---|---|
| **September 23, 2024** | Kick-off call |
| **September 23, 2024** | Start of primary review period |
| **October 3, 2024** | End of primary review period |

## 3. Detailed Findings

### 3.1. Unsafe hook trigger during manual deposit

| Target | BasicVault | | |
|---|---|---|---|
| **Category** | Business Logic | **Severity** | Medium |
| **Likelihood** | High | **Impact** | Medium |

#### Description

In the BasicVault contract, the `manualDeposit` function is identical to the `deposit` function, except it can only be called by authorized depositors, and it credits the deposit to the receiver without transferring the underlying asset:

```
function deposit(uint256 amount, address receiver)
    public withNoHalt(Action.Deposit) returns (uint256) {
  StorageV1 storage $ = _getStorageV1();
  StorageV2 storage $v2 = _getStorageV2();

  IERC20 asset_ = $.asset;
  IDepositHook hook = _depositHook($v2);
  uint256 newAmount =
    address(hook) == address(0) ? amount : hook.reportDeposit(_msgSender(),
    receiver, address(asset_), amount, amount);

  // [... omitted comment ...]
  asset_.safeTransferFrom(_msgSender(), address(this), newAmount);

  _deposit($v2, newAmount, receiver);

  return newAmount;
}

// [...]

function manualDeposit(uint256 amount, address receiver)
    external withNoHalt(Action.Deposit) returns (uint256) {
  StorageV1 storage $ = _getStorageV1();
  StorageV2 storage $v2 = _getStorageV2();

  if (!_isAllowed($, _msgSender(), Action.Deposit))
    revert Error.Unauthorized();

  IERC20 asset_ = $.asset;
```

```
    IDepositHook hook = _depositHook($v2);
    uint256 newAmount =
      address(hook) == address(0) ? amount : hook.reportDeposit(_msgSender(),
      receiver, address(asset_), amount, amount);

    _deposit($v2, newAmount, receiver);

    return newAmount;
  }
```

This function is used to account for a deposit made through a trusted, centralized caller address. The caller is responsible for ensuring that the amount of tokens minted by the call corresponds to a value that has already been transferred into, or will atomically be transferred into, the contract.

However, the amount that is minted is `newAmount`, which may differ from `amount` if a hook modifies it. If `manualDeposit` were called by an externally owned account (EOA), they cannot absolutely control the exact `newAmount` that the on-chain code is executed with, and so the receiver may receive a different amount of minted tokens than they expect.

### Impact

Incorrect accounting, which must be externally noticed and then manually fixed, occurs if a deposit hook unexpectedly modifies the amount of tokens minted during a call to `manualDeposit`.

For instance, the Cap contract uses a deposit hook to implement a simple cap on the amount of deposits. If a victim uses the centralized service to deposit some amount of tokens into a vault that has the Cap hook, then the trusted sender will call `manualDeposit` with that amount of tokens. In this case, an attacker can see this call and front-run it with a deposit that fills up the deposit cap, reducing `newAmount` to zero and causing no tokens to actually be minted to the user.

### Recommendations

We recommend clarifying what responsibilities the caller of `manualDeposit` has, in the case where a deposit hook changes the amount that is actually deposited. This is because, if it is called by an EOA, even though the `newAmount` is returned, it cannot be safely used in a second transaction to actually transfer the assets, before or after, until the exact execution is confirmed. On the other hand, if this function is called by a contract that reads this return value and transfers an amount equal to the return value, then that contract could just call the regular `deposit` function instead, on behalf of the intended receiver.

### Remediation

This issue has been acknowledged by Mitosis.

## 3.2. Incorrect vault check when there are no vaults

| | | | |
|---|---|---|---|
| **Target** | BasicVaultFactory | | |
| **Category** | Coding Mistakes | **Severity** | Low |
| **Likelihood** | High | **Impact** | Low |

### Description

In BasicVaultFactory, the `isVault` function checks whether an address is a vault deployed by the factory:

```
function isVault(address vault) public view returns (bool) {
  StorageV1 storage $ = _getStorageV1();
  uint256 vaultId = $.vaultIds[vault];
  return $.vaults[vaultId] == vault;
}
```

Since the first vault deployed will have a `vaultId` of zero, if an unknown address is passed into this function, it will return false because `$.vaults[vaultId]` will be the address of the zero vault, which is different from the calldata argument.

However, before the first vault is deployed, that storage slot contains zero, so the return value will be true.

### Impact

Before any vaults are deployed, `isVault` will incorrectly return true for the zero address.

This does not impact any code, because no contract code calls the view function `isVault`. However, this may have implications for future or external code.

### Recommendations

We recommend skipping the zero-vault ID since, in the `$.vaultIds` storage mapping, a vault not existing must have a zero value. If the zero-vault ID needs to be supported, then the meaning of an entry in `$.vaultIds` being zero would be that either the vault does not exist, or the vault exists and it is the zeroth vault, which requires additional logic and another storage read to figure out.

### Remediation

This issue has been acknowledged by Mitosis.

### 3.3. Gifted dust can cause gas griefing in the redeem queue

| | | | |
|---|---|---|---|
| **Target** | RedeemQueue | | |
| **Category** | Business Logic | **Severity** | Low |
| **Likelihood** | Low | **Impact** | Low |

#### Description

The `redeem` function in BasicVault allows the sender to specify an arbitrary `receiver`:

```
function redeem(uint256 amount, address receiver)
    external withNoHalt(Action.Redeem) returns (uint256)
```

Although the sender's tokens are burned, the released value enters the redeem queue on behalf of the receiver, instead of the sender of the transaction. Later, according to the logic implemented in the RedeemQueue, when the released value becomes claimable, the receiver must spend gas to claim this unexpected queue entry before any queue entries that they themselves actually authorized can be claimed:

```
function claim(address receiver) external withNoHalt(Action.Claim)
    returns (uint256) {
  // [...]
  (uint256 totalResolvedAmount, uint256 totalResolvedRequestCount)
    = $v2.redeemQueue.resolve(receiver);
  // [...]
}

function resolve(Storage storage $, address requestor)
  internal
  returns (uint256 totalResolvedAmount, uint256 resolvedRequestsCount)
{
  IndexByRequestor storage index = $.indexByRequestor[requestor];
  if (index._count == 0) return (0, 0);

  (uint256 nextResolvedIdx, uint256 resolveToIdx, bool available)
    = getAvailableResolveRange(index);
  if (!available) return (0, 0);

  // [... continue with the available range]
```

Since the minimum redeem value is zero, an attacker can spend gas to spam a victim with worthless amounts of redemption-queue entries.

## Impact

The victim of such an attack would have to spend about as much on gas as the attacker spent, to clear out the spammed worthless queue entries before they can continue redeeming assets.

## Recommendations

We recommend adding a pinpoint redemption function, where a user can specify a request ID to redeem. This will ensure that even if such an attack occurs, the impact is not permanent, because a user would not have to redeem their redeemable-queue requests in order.

## Remediation

This issue has been acknowledged by Mitosis.

## 3.4. Hook logic runs before possible ERC-777 reentrancy

| **Target** | BasicVault | | |
|---|---|---|---|
| **Category** | Protocol Risks | **Severity** | Low |
| **Likelihood** | Medium | **Impact** | Low |

### Description

The `deposit` function in the BasicVault contract has the following comment copied from the OpenZeppelin ERC-4626 vault implementation:

```
function deposit(uint256 amount, address receiver)
    public withNoHalt(Action.Deposit) returns (uint256) {
  StorageV1 storage $ = _getStorageV1();
  StorageV2 storage $v2 = _getStorageV2();

  IERC20 asset_ = $.asset;
  IDepositHook hook = _depositHook($v2);
  uint256 newAmount =
    address(hook) == address(0) ? amount : hook.reportDeposit(_msgSender(),
    receiver, address(asset_), amount, amount);

  // If _asset is ERC777, `transferFrom` can trigger a reentrancy BEFORE the
    transfer happens through the
  // `tokensToSend` hook. On the other hand, the `tokenReceived` hook, that is
    triggered after the transfer,
  // calls the vault, which is assumed not malicious.
  //
  // Conclusion: we need to do the transfer before we mint so that any
    reentrancy would happen before the
  // assets are transferred and before the shares are minted, which is a valid
    state.
  // slither-disable-next-line reentrancy-no-eth
  asset_.safeTransferFrom(_msgSender(), address(this), newAmount);

  _deposit($v2, newAmount, receiver);

  return newAmount;
}
```

However, the added call to `hook.reportDeposit` is before the transfer. This means that ERC-777 reentrancy can happen after the deposit is reported but before the asset is transferred. In other words, the deposit hook needs to leave the contract in a valid state.

### Impact

For example, if a hook contract used deposit, withdrawal, and transfer hooks to keep track of some yield allocated over time to the owners of these tokens, normally, the hook contract's notion of the total supply of the vault token would equal to the net deposit and withdrawals. However, if an attacker employs reentrancy via an ERC-777 hook, during this reentrancy, the hook's accounting of the total supply would be greater than the actual total supply returned by a call to `totalSupply`, since the new tokens had not been minted yet. This discrepancy can cause accounting errors during a time that the attacker can make arbitrary calls to the hook contract.

### Recommendations

There is a circular dependency here, where the hooks can alter how much of the underlying asset is transferred, but in order to do that, they must be run before the transfer, which means they cannot leave the contract in an invalid state after returning but before the new tokens are actually minted. We recommend reconsidering the design to resolve this circular dependency when the protocol integrates with ERC-777 tokens.

Alternatively, remove this comment and explicitly document that the vault does not support ERC-777 tokens.

### Remediation

This issue has been acknowledged by Mitosis.

## 3.5. Strategies allow the strategist to exfiltrate value

| | | | |
|---|---|---|---|
| **Target** | KarakStrategyV1_2 and others | | |
| **Category** | Coding Mistakes | **Severity** | High |
| **Likelihood** | N/A | **Impact** | Informational |

### Description

The trusted strategist is allowed to execute trades through the strategy contracts, using the assets in the vault. This means that the strategist can already exfiltrate value from the vault, if they acted maliciously, by self-sandwiching their transactions and extracting value economically.

However, the individual strategy contracts contain even easier ways for a strategist to exfiltrate value — for example, in the KarakStrategyV1_2 contract:

```
function _deposit(uint256 amount, bytes memory context) internal override {
  IERC20 vaultAsset_ = _vaultAsset();
  IVault vault_ = IVault(abi.decode(context, (address)));

  uint256 minSharesOut = vault_.convertToShares(amount);
  vaultAsset_.forceApprove(address(vault_), amount);
  _vaultSupervisor.deposit(vault_, amount, minSharesOut);
}
```

The `context` parameter is a completely free parameter set by the strategist. It cannot be an arbitrary address, since the `_vaultSupervisor` does check that the vault was deployed by it. However, this vault could be owned by the strategist on another account, in which case the value may be taken.

A second example can be found in MendiStrategy and IonicStrategy:

```
function _deposit(uint256 amount, bytes memory context) internal override {
  IERC20 vaultAsset_ = _vaultAsset();
  CToken cToken = abi.decode(context, (CToken));

  vaultAsset_.forceApprove(address(cToken), amount);
  cToken.mint(amount);
}
```

Again, the `context` parameter is completely controlled by the strategist, which means `cToken` can be a malicious contract that takes the approval and mints the tokens. This easily exfiltrates the value.

### Impact

The strategy contracts are not designed to add any guard rails against the strategist exfiltrating value from the vault.

### Recommendations

If the strategy contracts should implement guard rails against the strategist doing this, then they must be updated to add much more counterparty-specific logic in order to guard against possibly-malicious context data.

On the other hand, if the strategist is fully trusted with the entire value of the vault, then these strategies may convey a false sense of trust, since the system also includes a constraint where the strategist is not allowed to add and remove strategies themselves. In order to not mislead users about the capabilities of the strategists, if this is the case, we recommend removing these bypassable guardrails in order to explicitly highlight the intended centralization of this implementation.

### Remediation

This issue has been acknowledged by Mitosis.

# 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. Contracts do not contain yield-tracking logic

In general, users need to be offered some incentive to deposit their assets into a vault. Although the BasicVault contains logic to allow a strategist to trade and stake the assets, it does not contain any logic that distributes the profits or losses from that trading activity. One deposited unit of asset will always correspond to one withdrawn unit, whether or not the vault itself has realized a profit or loss. We recommend adding explicit logic for the distribution of profits or losses between vault depositors, in order to clarify how those incentives are set up.

## 4.2. The `_resolveWithIdleBalance` function should be called `_reserveWithIdleBalance`

The function `_resolveWithIdleBalance` actually reserves assets. Resolving is defined elsewhere in the contract to mean completely finishing a queue entry, sending assets to the end user. Reserving, on the other hand, sets aside idle balance and allows a queue entry to be resolved in the future.

```
function _resolveWithIdleBalance(StorageV2 storage $v2, IERC20 asset_)
    internal returns (uint256 resolved) {
    uint256 remaining = $v2.redeemQueue.remaining();
    if (remaining == 0) return 0;

    uint256 idleBalance_ = _idleBalance($v2, asset_);
    resolved = remaining > idleBalance_ ? idleBalance_ : remaining;
    $v2.redeemQueue.reserve(resolved);

    return resolved;
  }
```

So, we recommend renaming this function to `_reserveWithIdleBalance`.

## 4.3. Unused internal resolve function cannot be used safely

In the RedeemQueue library, there is an unused alternate `resolve` function that tries to resolve specific request IDs, instead of the `indexByRequestor` list:

```
/**
 * @dev pin-point resolving
 * @param $ RedeemQueue.Storage
 * @param requestIds List of requestIds to resolve
 * @return totalResolvedAmount Total amount resolved
 * @return resolvedRequestsCount Number of requests resolved
 */
function resolve(Storage storage $, uint256[] memory requestIds)
  internal
  returns (uint256 totalResolvedAmount, uint256 resolvedRequestsCount)
{
  uint256 redeemPeriod = $.redeemPeriod;
  uint256 cumulativeReservedAmount = $.cumulativeReservedAmount;

  for (uint256 idx = 0; idx < requestIds.length; idx++) {
    uint256 requestId = requestIds[idx];

    // use `tryGet` to filter the request that does not exist
    (Request memory request, bool found) = tryGet($.queue, requestId);
    if (!found) continue;
    if (!isResolvable(request, redeemPeriod, cumulativeReservedAmount))
    continue;

    // mark the request as resolved
    $.queue._data[requestId].resolvedAt = block.timestamp;

    totalResolvedAmount += getRequestAmount($.queue, request, requestId);
    resolvedRequestsCount++;
  }

  $.cumulativeResolvedAmount += totalResolvedAmount;
  $.queue._resolved += resolvedRequestsCount;

  return (totalResolvedAmount, resolvedRequestsCount);
}
```

Since this function has explicit support for `requestIds` that do not exist, this means that the caller has not accessed or sanitized these request ID numbers yet, so it likely does not have an assurance about what the `requestor` field is. However, there is no logic inside this function that checks the `requestor` — a request in the list with a different `requestor` than expected would be added to the `totalResolvedAmount` blindly along with correct ones.

So, without modification, this function likely will cause an attacker to be able to claim and resolve queue entries that are not owned by them. This issue has no impact since this function is internal and unused, but, even so, we recommend removing or redesigning this function so that it does not cause any issues for future revisions of this library.

## 4.4. Confusion arising from the simultaneous use of function names

The RedeemQueue contract contains four structs, which are `Request`, `IndexByRequestor`, `Queue`, and `Storage`. The BasicVault then uses this library with the statement `using RedeemQueue for *;`.

Some library functions have repeated names, which makes the manual resolution of what a function is actually called confusing. Here is an exhaustive list of functions in RedeemQueue that share a name with another function:

```
function get(IndexByRequestor storage index, uint256 idx);
function get(Queue storage queue, uint256 requestId);
function isResolvable(Request memory request, uint256 redeemPeriod,
    uint256 cumulativeReservedAmount);
function isResolvable(Storage storage $, uint256 requestId);
function isResolvable(Storage storage $, uint256[] memory requestIds)
function isResolved(Request memory request);
function isResolved(Storage storage $, uint256 requestId);
function isResolved(Storage storage $, uint256[] memory requestIds)
function len(IndexByRequestor storage index);
function len(Queue storage queue);
function previewResolve(Storage storage $, address requestor);
function previewResolve(Storage storage $, address requestor,
    uint256 cumulativeReservedAmount);
function previewResolve(Storage storage $, uint256[] memory requestIds);
function push(IndexByRequestor storage index, uint256 requestId);
function push(Queue storage queue, address from, uint256 amount);
function push(Storage storage $, address from, uint256 amount);
function remaining(Queue storage queue);
function remaining(Storage storage $);
function tryGet(IndexByRequestor storage index, uint256 idx);
function tryGet(Queue storage queue, uint256 requestId);
```

When the library is called, which function is called depends on the type of the expression before the period. However, without the actual struct definitions pulled up, this can be unclear. For example, see this statement in BasicVault:

```
uint256 remaining = $v2.redeemQueue.remaining();
```

There are two `remaining` functions in the list above. The one attached to the `Queue` type returns the number of requests that have not yet been resolved:

```
function remaining(Queue storage queue) internal view returns (uint256) {
  return queue._count - queue._resolved;
}
```

On the other hand, the one attached to the `Storage` type returns the amount of assets that have not yet been reserved:

```
function remaining(Storage storage $)
    internal view returns (uint256 remaining_) {
  uint256 len_ = len($.queue);
  return len_ == 0 ? 0 : get($.queue, len_ - 1).cumulativeAmount
    - $.cumulativeReservedAmount;
}
```

It is not initially clear which `remaining` function is called. Even though the member name is `redeemQueue`, the struct member actually has type `RedeemQueue.Storage`, so it is the second one. In order to call the first variant, the code would need to do `$v2.redeemQueue.queue.remaining()`, which is what it does elsewhere in the BasicVault. It would be more practical for each function name to be distinct from one another if they do distinct actions.

On the other hand, for the `isResolvable` and `isResolved` functions, the first one of each merely first get the request from the storage and calls the second one, and the third one similarly loops through the multiple `requestIds` to call the second one with some quantities memoized for performance. This is an acceptable use of identical function names, because the disambiguating arguments are after the first argument, and because the actions that the functions do are logically the same.

We recommend renaming the implementations of the `get`, `len`, `push`, `remaining`, and `tryGet` functions because they either have identical argument types past the first argument, or because they do logically different actions with their identically-named counterparts.

# 5. Threat Model

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

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

## 5.1. Module: BasicVault.sol

### Function: `claim(address receiver)`

This function is for claiming.

### Inputs

- `receiver`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Address of the receiver.

### Branches and code coverage

**Intended branches**

- Check if the `redeemQueue` has any remaining items.
    - ☑ Test coverage
- Call `reportDeposit` if hook is not zero address.
    - ☑ Test coverage
- Send asset token to the receiver by `totalResolvedAmount`.
    - ☑ Test coverage

**Negative behavior**

- The `redeemQueue` does not have any remaining items.
    - ☑ Negative test

### Function: `deposit(uint256 amount, address receiver, bytes permitData)`

This function is for depositing using the ECDSA. It is implemented for abstraction.

### Inputs

- `amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Amount to deposit.
- `receiver`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Address of the receiver.
- `permitData`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must verify the caller using the given ECDSA.
    - **Impact**: ECDSA information with the deadline.

### Branches and code coverage

**Intended branches**

- Call `reportDeposit` if hook is not zero address.
    - ☑ Test coverage
- Call another abstraction deposit; it will take the proper amount of tokens from the caller.
    - ☑ Test coverage

**Negative behavior**

- Revert when the ECDSA cannot verify the request.
    - ☑ Negative test

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

This function is for depositing. It is implemented for abstraction.

### Inputs

- `amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Amount to deposit.
- `receiver`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Address of the receiver.

### Branches and code coverage

**Intended branches**

- Call `reportDeposit` if hook is not zero address.
  - ☑ Test coverage
- Take the proper amount of tokens from the caller.
  - ☑ Test coverage

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

This function is for redeeming. It is implemented for abstraction. It calls the actual redeem function, `_redeem`.

### Inputs

- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: N/A.
- `receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: N/A.

## Function: `_deposit(StorageV2 $v2, uint256 amount, address receiver)`

This function is for depositing. This is the actual deposit logic.

### Inputs

- `$v2`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Arguments for storage.
- `amount`
  - **Control**: It is passed from the abstraction deposit, which in this case is fully controlled by the caller.
  - **Constraints**: Must not be zero.
  - **Impact**: Amount to deposit.
- `receiver`
  - **Control**: It is passed from the abstraction deposit, which in this case is fully

controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Address of the receiver.

### Branches and code coverage

**Intended branches**

- Check if the amount is zero.
  - ☑ Test coverage
- Mint the vault tokens to the receiver.
  - ☑ Test coverage

**Negative behavior**

- The amount is zero.
  - ☑ Negative test

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

This function is for redeeming. This is the actual redeeming logic.

### Inputs

- `amount`
    - **Control**: It is passed from the abstraction redeem, which in this case is fully controlled by the caller.
    - **Constraints**: Must not be zero.
    - **Impact**: Amount to deposit.
- `receiver`
    - **Control**: It is passed from the abstraction redeem, which in this case is fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Address of the receiver.

### Branches and code coverage

**Intended branches**

- Check if the amount is zero.
  - ☑ Test coverage
- Call `reportRedeem` if hook is not zero address.
  - ☑ Test coverage

- Burn `newAmount` tokens for `_msgSender()`.
  - ☑ Test coverage
- Push request in `redeemQueue` if `redeemQueue` is enabled.
  - ☑ Test coverage

**Negative behavior**

- The amount is zero.
  - ☑ Negative test
- The `idleBalance` is smaller than `newAmount`.
  - ☑ Negative test

### 5.2. Module: StrategyExecutor.sol

#### Function: `execute(IStrategyExecutor.Call[] calls)`

This function is for a strategist to execute strategy.

#### Inputs

- `calls`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Calldata for calling strategy.

#### Branches and code coverage

**Intended branches**

- Check if the caller is a strategist.
  - ☑ Test coverage
- Check that each strategy of calls is enabled.
  - ☑ Test coverage
- `Delegatecall` each strategy of calls.
  - ☑ Test coverage
- Transfer all remaining assets to the vault.
  - ☑ Test coverage
- Transfer all remaining Ether to the strategist.
  - ☑ Test coverage

**Negative behavior**

- The caller is not a strategist.
  - ☑ Negative test
- Strategy is not enabled.

☑ Negative test

## 6. Assessment Results

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

During our assessment on the scoped Mitosis EOL contracts, we discovered five findings. No critical issues were found. One finding was of medium impact, three were of low impact, and the remaining finding was informational in nature.

### 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.