Zellic

August 1, 2025

Prepared for
thai@mitosis.org
ray@mitosis.org
eddy@mitosis.org
Mitosis

Prepared by
Kritsada Dechawattana
Qingying Jie
Zellic

Extensible Vaults

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 July 30th to August 1st, 2025. During this engagement, Zellic reviewed Extensible Vaults'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:

- Is the protocol working correctly?
- Are access controls implemented effectively to prevent unauthorized operations?
- Could an on-chain attacker drain the vaults?
- Are there any vulnerabilities that could result in the loss of user funds?

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 Extensible Vaults contracts, we discovered four findings, all of which were medium impact.

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	4
🟩 Low	0
⬜ Informational	0

2. Introduction

2.1. About Extensible Vaults

Mitosis contributed the following description of Extensible Vaults:

> Extensible Vaults is an ERC4626-compliant tokenized vault system with modular manager contracts for flexible and secure liquidity management. Designed for flexible and rapid adaptation to a wide variety of DeFi protocols, it implements operational safeguards including rate limiting and role-based access controls for secure liquidity management.

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:

Extensible Vaults Contracts

Type	Solidity
Platform	EVM-compatible
Target	extensible-vaults
Repository	https://github.com/mitosis-org/extensible-vaults ↗
Version	`17b1d6b8a7c6f872df20e9ceb7ec160da872a537`
Programs	`src/*.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 3.5 person-days. The assessment was conducted by two consultants over the course of three calendar days.

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 ↗

Pedro Moura
Engagement Manager
pedro@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Kritsada Dechawattana
Engineer
kritsada@zellic.io ↗

Qingying Jie
Engineer
qingying@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

July 30, 2025	Start of primary review period
August 1, 2025	End of primary review period

3. Detailed Findings

3.1. Centralization risks

Target	ManagedAssetVault		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

The `ManagedAssetVault` contract allows an `OPERATOR_ROLE` to directly update the `managedAssets` value through the `updateManagedAssets()` function without any verification that the reported amount accurately reflects the actual assets managed externally. The `managedAssets` value is a critical component in the vault's `totalAssets()` calculation, which directly impacts share pricing, deposit/withdrawal ratios, and overall vault valuation.

While the function includes rate limiting through time intervals and maximum change percentages, these controls only limit the frequency and magnitude of updates but do not verify the accuracy of the reported amounts.

```
function updateManagedAssets(uint256 amount) external onlyRole(OPERATOR_ROLE)
    {
  ManagedAssetVaultStorage storage $ = _getManagedAssetVaultStorage();
  uint256 prev = $.managedAssets;

  // Check time interval
  uint32 timeSinceLastUpdate = uint32(Time.timestamp()
    - $.managedAssetsUpdatedAt);
  if (timeSinceLastUpdate < $.managedAssetsMinUpdateInterval) {
    revert IManagedAssetVault__UpdateTooFrequent(
      timeSinceLastUpdate, $.managedAssetsMinUpdateInterval
    );
  }

  // Check change amount limit
  if (prev > 0 && $.managedAssetsMaxChangeBps > 0) {
    uint256 maxChange = (prev * $.managedAssetsMaxChangeBps) / 10000;
    uint256 change = amount > prev ? amount - prev : prev - amount;
    if (change > maxChange) {
      revert IManagedAssetVault__ChangeTooLarge(prev, amount, maxChange);
    }
  }

  $.managedAssets = amount;
  $.managedAssetsUpdatedAt = Time.timestamp();
  emit IManagedAssetVault.ManagedAssetsUpdated(prev, amount);
}

[...]

function totalAssets()
  public
  view
  virtual
  override(IManagedAssetVault, ERC4626Upgradeable)
  returns (uint256)
{
  ManagedAssetVaultStorage storage $ = _getManagedAssetVaultStorage();
  return $.managedAssets + IERC20(asset()).balanceOf(address(this));
}
```

Impact

A malicious or compromised operator could exploit this vulnerability to manipulate vault accounting for their benefit. Specifically, they could:

High Impact Scenarios:

- **Inflate Share Value:** Report inflated `managedAssets` to artificially increase `totalAssets()`, making existing shares worth more than they should be, allowing the operator to withdraw more value than deposited.
- **Deflate Share Value:** Report deflated `managedAssets` before making deposits to acquire shares at artificially low prices, then restore accurate reporting.
- **Front-run Large Deposits/Withdrawals:** Manipulate reported assets just before large transactions to extract value from other users.

Financial Impact: Given that modern DeFi vaults often manage millions in TVL, even small percentage manipulations could result in substantial financial losses for users. The vulnerability essentially creates a scenario where users must completely trust the operator's honesty regarding external asset management.

Recommendations

Implement a verification mechanism to ensure reported managed assets reflect actual external balances:

- **Add Oracle or Proof-Based Verification:** Require the operator to provide cryptographic proofs or use price oracles to verify that reported amounts match actual external positions.

- **Implement Multi-Signature Requirements:** Replace single-operator control with a multi-signature scheme requiring multiple parties to approve asset updates.
- **Add Time-Delayed Updates with Challenge Period:** Implement a timelock mechanism where asset updates are announced but not immediately effective, allowing other parties to challenge inaccurate reports.
- **Create Automated Verification:** Integrate with external protocols directly where possible to automatically query actual managed balances rather than relying on manual reporting.

Remediation

This issue was mitigated in commit 015c2052eedb0781ddfcf8a91e75a82f3876655d ↗ by moving the centralized logic we mentioned to the new `ExternalManager` contract, however the new contracts introduced in this commit have not been fully reviewed.

Mitosis acknowledged the finding and provided this additional response:

> The centralization issue only exists in the ExternalManager contract. Excluding ExternalManager, all other contracts track balances on-chain and have on-chain DeFi integrations, so there are no centralization issues. ExternalManager is intentionally designed as a centralized contract beyond just the totalBalance calculation logic. It is a contract that trusts the ASSET_MANAGER_ROLE and is intended to be used only for very specific protocol integrations, while generally on-chain integration-based contracts will be used. Since ExternalManager was originally designed in this trusted manner, we determined that additional safeguards would be meaningless. The OPERATOR_ROLE will be implemented as a multi-sig wallet in practice. And we have a rate limiting feature. Therefore, we believe a certain level of safety is guaranteed.

3.2. Inflation attack vulnerability on empty vault

Target	ManagedAssetVault		
Category	Business Logic	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The `ManagedAssetVault` contract is vulnerable to an inflation attack when the vault is empty and `managedAssets` is zero, allowing an attacker to steal funds from subsequent depositors. This vulnerability exploits the ERC4626 share calculation mechanism when `totalSupply()` and `totalAssets()` are both zero. The attack works through the following scenario:

1. Attacker deposits 1 wei and receives 1 share when vault is empty.
2. Attacker directly transfers 1000 ETH to the vault contract (not through deposit), at this step: `totalAssets = 1000 ETH + 1 wei`, `totalSupply = 1 share`.
3. When victim deposits 999 ETH, they receive shares = (999e18 * 1) / (1000e18 + 1) ≈ 0 shares due to rounding down.
4. Attacker withdraws all funds using their 1 share, stealing the victim's 999 ETH.

The vulnerability is present in the `deposit()`, which inherits the standard ERC4626 share calculation: `shares = assets * totalSupply / totalAssets`. The contract's `totalAssets()` function includes both the contract's token balance and `managedAssets`, making it susceptible to balance manipulation through direct transfers.

```
function totalAssets()
  public
  view
  virtual
  override(IManagedAssetVault, ERC4626Upgradeable)
  returns (uint256)
{
  ManagedAssetVaultStorage storage $ = _getManagedAssetVaultStorage();
  return $.managedAssets + IERC20(asset()).balanceOf(address(this));
}
```

Impact

An attacker can entirely steal a victim's deposit when the vault is empty. This affects the vault's ability to bootstrap liquidity and poses significant risk during initial deployment or if the vault is ever completely drained. The attack is economically viable for any deposit larger than the cost of the direct transfer used for inflation, making it a significant threat to the protocol's security and user funds.

Recommendations

There are several approaches to mitigate this. Implement one of the following mitigation strategies that aligns with the business requirements:

1. Override `_decimalsOffset()` function from the `ERC4626Upgradeable` base contract.

   ```
   function _decimalsOffset() internal view override returns (uint8) {
       return 6; // Adds 10^6 virtual shares/assets to prevent inflation
       attacks
   }
   ```

   The `_decimalsOffset()` approach is preferred as it's the standard OpenZeppelin solution that adds virtual shares and assets to the calculation, making the exchange rate manipulation economically infeasible while maintaining full compatibility with the ERC4626 standard.

   However, this solution will increase the share decimals, which will require refactoring the test suite.

2. Perform a small initial deposit (>=1000 wei) and burn the received shares upon deployment. This prevents the shares from being inflatable.

Note: These approaches do not completely eliminate the inflation attack vulnerability but rather increase the economic cost of executing the attack, making it unprofitable for attackers to perform the manipulation relative to the potential gains from victim deposits.

Remediation

This issue has been acknowledged by Mitosis, and a fix was implemented in commit 04bbb2a6 ↗.

3.3. ERC4626 inflation attack on first depositors in `yoVaultManager` contract

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

Description

The `yoVaultManager.deposit()` function is vulnerable to inflation attacks against the underlying `yoVault` vault. When users deposit assets via the manager, the function calls `IERC4626(address(yoVault)).deposit(amount, address(this))` without checking the returned shares value. In ERC4626 vaults, the number of shares minted is calculated as `assets * totalSupply / totalAssets`, which can be manipulated through inflation attacks. An attacker can execute this attack by:

1. being the first depositor in the `yoVault` and depositing a minimal amount (e.g., 1 wei).
2. directly transferring a large amount of the underlying asset to the vault to inflate the asset-to-share ratio.
3. causing subsequent deposits through the yoVaultManager to receive zero shares due to precision loss from rounding down.

```
function deposit(address asset, uint256 amount) external nonReentrant {
  require(IERC4626(address(yoVault)).asset() == asset,
    StdError.InvalidParameter('asset'));
  require(amount > 0, StdError.ZeroAmount());

  IERC20(asset).transferFrom(_msgSender(), address(this), amount);

  IERC20(asset).approve(address(yoVault), amount);
  IERC4626(address(yoVault)).deposit(amount, address(this));
  IERC20(asset).approve(address(yoVault), 0);

  emit Deposit(asset, _msgSender(), amount);
}
```

Impact

Users depositing through the `yoVaultManager` could lose their entire deposit without receiving any vault shares in return. The first depositors are most vulnerable as they deposit into a vault with a manipulated exchange rate. While the deposited assets would technically remain in the vault, they would be inaccessible to the `yoVaultManager` since no shares were minted to represent the ownership. This represents a complete loss of funds for affected users, with the potential value at risk being any deposits made through the manager during the attack window.

Recommendations

Implement a shares validation check after the deposit to ensure users receive a proportional amount of shares for their deposit:

```
uint256 shares = IERC4626(address(yoVault)).deposit(amount, address(this));
require(shares > 0, "Deposit failed: received zero shares");
```

Remediation

This issue has been acknowledged by Mitosis, and a fix was implemented in commit 89cde6b3 ↗.

3.4. Unsafe ERC20 operations

Target	ManagedAssetVault, ManagerVault		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

Some ERC20 tokens silently fail by returning `false` instead of reverting when transfer or approve operations fail. Certain contracts execute these ERC20 operations without using OpenZeppelin's `SafeERC20` library, which is the security best practice for ERC20 operations.

The following contracts are using the unsafe ERC20 functions:

- ManagedAssetVault.sol#187
- ManagedAssetVault.sol#189
- ManagerVault.sol#51
- ManagerVault.sol#62

Impact

When transfer operations fail silently without reverting the transaction, the contract continues execution and updates its state variables while the underlying token balances remain unchanged.

Recommendations

Using the `safeTransfer`, `safeTransferFrom` and `forceApprove` instead. This properly implements OpenZeppelin's `SafeERC20` library to ensure safe token operations.

Remediation

This issue has been acknowledged by Mitosis, and a fix was implemented in commit `a1761778` ↗.

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 return value type of the function `managedAssetsUpdatedAt` is inconsistent with the type of variable `managedAssetsUpdatedAt`

The type of the variable `managedAssetsUpdatedAt` is `uint48`.

```
struct ManagedAssetVaultStorage {
    // [...]
    uint48 managedAssetsUpdatedAt;
    // [...]
}
```

However, the return value type of the function `managedAssetsUpdatedAt` is `uint256`, which is inconsistent with the variable `managedAssetsUpdatedAt`.

```
function managedAssetsUpdatedAt() external view returns (uint256) {
    return _getManagedAssetVaultStorage().managedAssetsUpdatedAt;
}
```

Consider changing the return value type of the function `managedAssetsUpdatedAt` to `uint48`.

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: ManagedAssetVaultWithExtraData.sol

Function: `depositWithExtraData(uint256 assets, address receiver, bytes extraData)`

Allows users to provide extra data when depositing a specified amount of assets into the vault.

Inputs

- `assets`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be greater than zero.
  - **Impact**: The amount of assets to deposit.
- `receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must not be a zero address.
  - **Impact**: The address that will receive the shares.
- `extraData`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Additional data to include with the deposit.

Branches and code coverage

Intended branches

- Successfully deposits assets into the vault.
  - ☑ Test coverage

Negative behavior

- Reverts if the return value of the function `maxDeposit` is zero.
  - ☑ Negative test

- Reverts if the `assets` is zero.
    - ☐ Negative test

Function: `mintWithExtraData(uint256 shares, address receiver, bytes extraData)`

Allows users to provide extra data when minting a specified amount of shares in exchange for assets.

Inputs

- `shares`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be greater than zero.
    - **Impact**: The amount of shares to mint.
- `receiver`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must not be a zero address.
    - **Impact**: The address that will receive the shares.
- `extraData`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Additional data to include with the minting.

Branches and code coverage

Intended branches

- Successfully deposits assets into the vault.
    - ☐ Test coverage

Negative behavior

- Reverts if the return value of the function `maxMint` is zero.
    - ☐ Negative test
- Reverts if the `shares` amount is zero.
    - ☐ Negative test

5.2. Module: ManagedAssetVault.sol

Function: `depositToManager(address manager, uint256 amount)`

Allows a sender with the `OPERATOR_ROLE` to deposit assets from the vault into a specified manager.

Inputs

- `manager`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The `manager` must have the `ASSET_MANAGER_ROLE`.
  - **Impact**: The `manager` will receive the deposited assets.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The `amount` must be greater than zero.
  - **Impact**: Transfers the `amount` of assets from the vault to the `manager`, and increases the `managedAssets` by the same amount.

Branches and code coverage

Intended branches

- The asset balance of the manager is increased by the `amount`.
  - [x] Test coverage
- The asset balance of the vault is decreased by the `amount`.
  - [ ] Test coverage
- The `managedAssets` is increased by the `amount`.
  - [x] Test coverage

Negative behavior

- Reverts if the sender does not have the `OPERATOR_ROLE`.
  - [x] Negative test
- Reverts if the `amount` is zero.
  - [ ] Negative test
- Reverts if the `manager` does not have the `ASSET_MANAGER_ROLE`.
  - [x] Negative test

Function: `deposit(uint256 assets, address receiver)`

Users can deposit a specified amount of assets into the vault and receive shares in return.

Inputs

- `assets`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be greater than zero.
    - **Impact**: The amount of assets to deposit.
- `receiver`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must not be a zero address.
    - **Impact**: The address that will receive the shares.

Branches and code coverage

Intended branches

- Successfully deposits assets into the vault.
    - ☑ Test coverage

Negative behavior

- Reverts if the return value of the function `maxDeposit` is zero.
    - ☑ Negative test
- Reverts if the `assets` is zero.
    - ☐ Negative test

Function: `mint(uint256 shares, address receiver)`

Users can deposit assets into the vault and receive a specified amount of shares in return.

Inputs

- `shares`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be greater than zero.
    - **Impact**: The amount of shares to mint.

- `receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must not be a zero address.
  - **Impact**: The address that will receive the shares.

Branches and code coverage

Intended branches

- Successfully deposits assets into the vault.
  - ☐ Test coverage

Negative behavior

- Reverts if the return value of the function `maxMint` is zero.
  - ☑ Negative test
- Reverts if the `shares` amount is zero.
  - ☐ Negative test

Function: `redeem(uint256 shares, address receiver, address owner_)`

Allows the `redeemQueue` to burn a specified amount of shares and withdraw assets from the vault on behalf of the owner, and send them to the receiver.

Inputs

- `shares`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must not exceed the return value of `maxRedeem(owner_)`.
  - **Impact**: The amount of shares to burn.
- `receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The address that will receive the assets.
- `owner_`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must not be a zero address.
  - **Impact**: The address that owns the assets being withdrawn.

Branches and code coverage

Intended branches

- Successfully withdraws assets from the vault via the `redeemQueue`.
  - ☐ Test coverage

Negative behavior

- Reverts if the caller is not the `redeemQueue`.
  - ☐ Negative test
- Reverts if the `owner_` does not have enough assets to withdraw.
  - ☐ Negative test

Function: `updateManagedAssets(uint256 amount)`

Allows a sender with the `OPERATOR_ROLE` to modify the value of `managedAssets` within a certain range.

Inputs

- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The difference between the `amount` and the current value of `managedAssets` must not exceed `managedAssets * managedAssetsMaxChangeBps / 10000`.
  - **Impact**: The new value of the `managedAssets`.

Branches and code coverage

Intended branches

- Successfully updates the `managedAssets` within the allowed range.
  - ☑ Test coverage
- The `managedAssetsUpdatedAt` is updated to the current timestamp.
  - ☑ Test coverage

Negative behavior

- Reverts if the sender does not have the `OPERATOR_ROLE`.
  - ☑ Negative test

- Reverts if the `amount` is not within the allowed range based on the previous value of `managedAssets`.
  - ☑ Negative test
- Reverts if the time since the last update is less than `managedAssetsMinUpdateInterval`.
  - ☑ Negative test

Function: `withdrawFromManager(address manager, uint256 amount)`

Allows a sender with the `OPERATOR_ROLE` to withdraw assets from a specified manager back to the vault.

Inputs

- `manager`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The `manager` must have the `ASSET_MANAGER_ROLE`.
  - **Impact**: Withdraws assets from the `manager`.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The `amount` must be greater than zero, must not be greater than the `managedAssets`, and the `manager` must have enough assets to withdraw.
  - **Impact**: Transfers the `amount` of assets from the `manager` back to the vault, and decreases the `managedAssets` by the same amount.

Branches and code coverage

Intended branches

- The asset balance of the vault is increased by the `amount`.
  - ☑ Test coverage
- The asset balance of the `manager` is decreased by the `amount`.
  - ☑ Test coverage
- The `managedAssets` is decreased by the `amount`.
  - ☑ Test coverage

Negative behavior

- Reverts if the sender does not have the `OPERATOR_ROLE`.
  - ☑ Negative test

- Reverts if the `manager` does not have the `ASSET_MANAGER_ROLE`.
  - ☑ Negative test
- Reverts if the `amount` is greater than the `managedAssets`.
  - ☑ Negative test
- Reverts if the `amount` is zero.
  - ☐ Negative test

Function: `withdraw(uint256 assets, address receiver, address owner_)`

Allows the `redeemQueue` to withdraw a specified amount of assets from the vault on behalf of the owner and send them to the receiver.

Inputs

- `assets`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must not exceed the return value of `maxWithdraw(owner_)`.
  - **Impact**: The amount of assets to withdraw.
- `receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The address that will receive the assets.
- `owner_`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must not be a zero address.
  - **Impact**: The address that owns the assets being withdrawn.

Branches and code coverage

Intended branches

- Successfully withdraws assets from the vault via the `redeemQueue`.
  - ☐ Test coverage

Negative behavior

- Reverts if the caller is not the `redeemQueue`.
  - ☐ Negative test

- Reverts if the `owner_` does not have enough assets to withdraw.
    - ☐ Negative test

5.3. Module: ManagerVault.sol

Function: `deposit(address asset, uint256 amount)`

Deposits a specified amount of an asset into the vault.

Inputs

- `asset`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: The asset to be deposited.
- `amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must not be zero.
    - **Impact**: The amount of assets to be deposited.

Branches and code coverage

Intended branches

- Successfully deposits the specified amount of the asset into the vault.
    - ☑ Test coverage

Negative behavior

- Reverts if the amount is zero.
    - ☑ Negative test

Function: `withdraw(address asset, uint256 amount, address receiver)`

Allows a sender with the `WITHDRAWAL_ROLE` to withdraw a specified amount of an asset from the vault and send it to a receiver.

Inputs

- `asset`

    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: The asset to be withdrawn.
- `amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must not be zero and must not exceed the asset balance of the vault.
    - **Impact**: The amount of assets to be withdrawn.
- `receiver`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: The address that will receive the withdrawn assets.

Branches and code coverage

Intended branches

- Successfully withdraws the specified amount of the asset from the vault and sends it to the receiver.
    - ☑ Test coverage

Negative behavior

- Reverts if the amount is zero.
    - ☑ Negative test
- Reverts if the amount exceeds the asset balance of the vault.
    - ☑ Negative test
- Reverts if the caller does not have the `WITHDRAWAL_ROLE`.
    - ☑ Negative test

5.4. Module: ReclaimQueueWithExtraData.sol

Function: `requestWithExtraData(uint256 shares, address receiver, address vault, bytes extraData)`

Allows a user to send a redeem request with extra data.

Inputs

- `shares`

- **Control**: Fully controlled by the caller.
- **Constraints**: N/A.
- **Impact**: The amount of shares to burn.

- `receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The address that will claim the redeemed assets.
- `vault`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The queue of the vault must be enabled.
  - **Impact**: The address of the vault to redeem from.
- `extraData`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Extra data to include with the request.

Branches and code coverage

Intended branches

- The user successfully requests.
  - ☐ Test coverage

Negative behavior

- Reverts if the queue of the vault is not enabled.
  - ☐ Negative test
- Reverts if the sender does not have enough shares to redeem.
  - ☐ Negative test

5.5. Module: yoVaultManager.sol

Function: `deposit(address asset, uint256 amount)`

Allows users to deposit a specified amount of asset into the `yoVault`.

Inputs

- `asset`

    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must match the underlying asset of the `yoVault`.
    - **Impact**: The asset to be deposited.
- `amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be greater than 0.
    - **Impact**: The amount of asset to be deposited.

Branches and code coverage

Intended branches

- The asset balance of the `yoVault` is increased by the deposited amount.
    - ☑ Test coverage
- The asset balance of the user is reduced by the deposited amount.
    - ☑ Test coverage
- The contract yoVaultManager receives the shares from the `yoVault`.
    - ☑ Test coverage

Negative behavior

- Reverts if the asset does not match the underlying asset of the `yoVault`.
    - ☑ Negative test
- Reverts if the amount is zero.
    - ☑ Negative test

Function: `withdrawFromYoVault(address asset, uint256 shares)`

Allows a sender with the `OPERATOR_ROLE` to burn a specified amount of shares to withdraw the asset from the `yoVault`.

Inputs

- `asset`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must match the underlying asset of the `yoVault`.
    - **Impact**: The asset to be withdrawn.
- `shares`

- **Control**: Fully controlled by the caller.
- **Constraints**: Must be greater than 0, and must not exceed the shares held by the contract yoVaultManager.
- **Impact**: The amount of shares to be burned.

Branches and code coverage

Intended branches

- Successfully withdraws the asset from the `yoVault`.
  - ☑ Test coverage

Negative behavior

- Reverts if the sender does not have the `OPERATOR_ROLE`.
  - ☑ Negative test
- Reverts if the asset does not match the underlying asset of the `yoVault`.
  - ☐ Negative test
- Reverts if the `shares` is zero.
  - ☐ Negative test

Function call analysis

- `yoVault.requestRedeem(shares, address(this), address(this))`
  - **What is controllable?** `shares`.
  - **If the return value is controllable, how is it used and how can it go wrong?** If the request is immediately processed, the return value is the amount of asset withdrawn. Otherwise, the return value is 0.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `withdraw(address asset, uint256 amount, address receiver)`

Allows a sender with the `WITHDRAWAL_ROLE` to withdraw a specified amount of asset from the contract yoVaultManager to a receiver.

Inputs

- `asset`
  - **Control**: Fully controlled by the caller.

    - **Constraints**: Must match the underlying asset of the `yoVault`.
    - **Impact**: The asset to be withdrawn.
- `amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be greater than 0, and must not exceed the asset balance of the contract yoVaultManager.
    - **Impact**: The amount of asset to be withdrawn.
- `receiver`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be a valid address.
    - **Impact**: The address that will receive the withdrawn assets.

Branches and code coverage

Intended branches

- The asset balance of the contract yoVaultManager is reduced by the withdrawn amount.
    - ☐ Test coverage
- The asset balance of the receiver is increased by the withdrawn amount.
    - ☑ Test coverage

Negative behavior

- Reverts if the sender does not have the `WITHDRAWAL_ROLE`.
    - ☑ Negative test
- Reverts if the asset does not match the underlying asset of the `yoVault`.
    - ☑ Negative test
- Reverts if the amount is zero.
    - ☑ Negative test
- Reverts if the contract yoVaultManager does not have enough asset to withdraw.
    - ☑ Negative test

6. Assessment Results

During our assessment on the scoped Extensible Vaults contracts, we discovered four findings, all of which were medium impact.

After all findings have been resolved and tests updated, including regression coverage, the code appears to be better positioned for deployment.

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.