Zellic

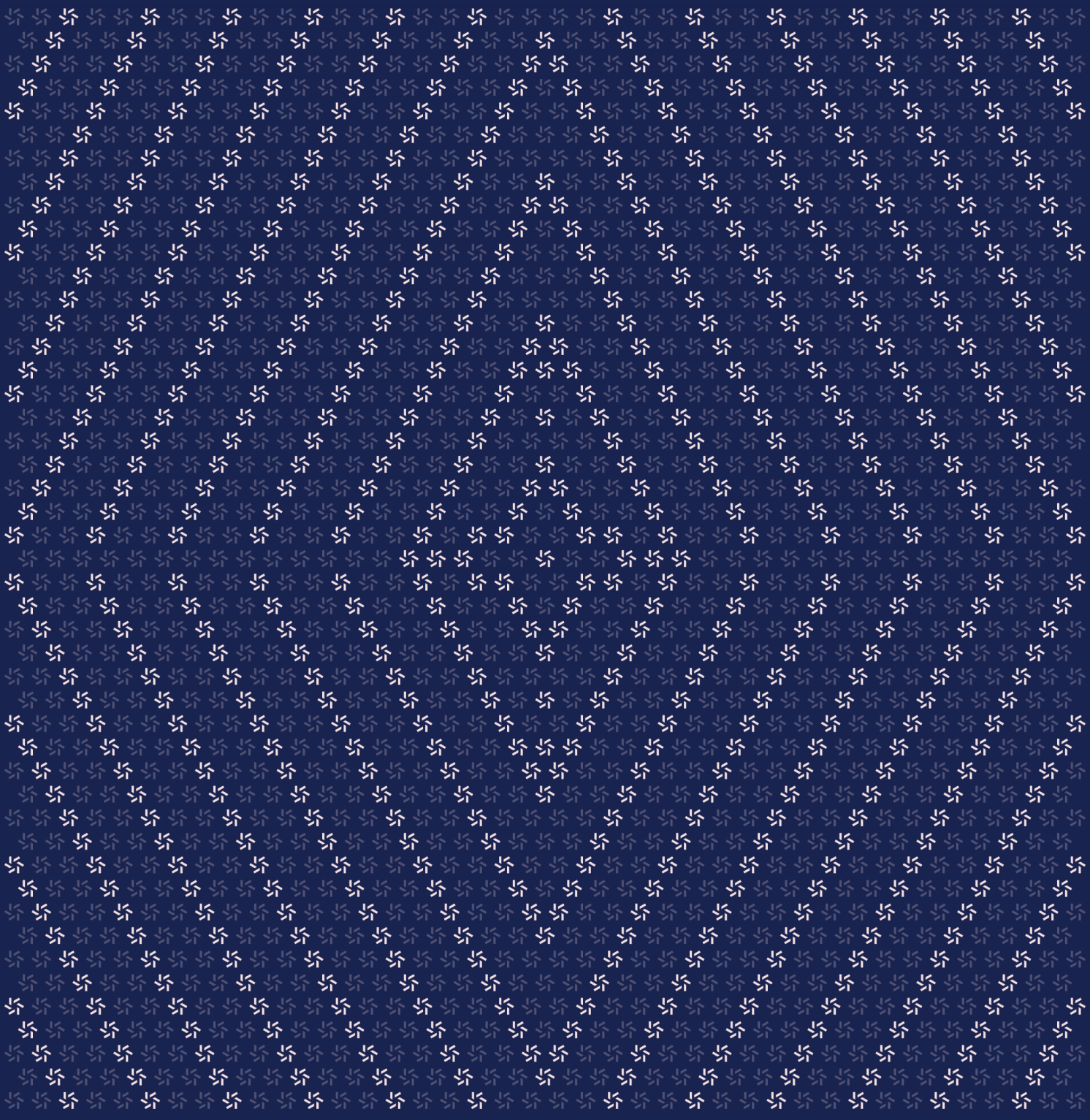
Prepared for
Luke
Eddy
Mitosis

Prepared by
Hojung Han
SeungHyeon Kim
Zellic

July 15, 2024

Mitosis Vault

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 10th to July 15th, 2024. During this engagement, Zellic reviewed Mitosis Vault'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 the funds be locked on the bridge due to incomplete logic implementation?
- Could an attacker make the contracts DOS?
- Could an attacker drain the tokens by reentrancy?

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 Vault contracts, we discovered one finding, which was of high impact.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	0
Low	0
Informational	0

2. Introduction

2.1. About Mitosis Vault

Mitosis contributed the following description of Mitosis Vault:

> Mitosis Vault is deployed across multiple chains, allowing users on each chain to deposit into the Mitosis Vault and obtain miAssets. The Mitosis Vault plays a crucial role in safeguarding users' funds.

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 Vault Contracts

Repository	https://github.com/mitosis-org/evm ↗
Version	evm: `ce63d9f4b2b63bbf23648b1ba83febf430d1492e`
Programs	• vault/* • helpers/ccdm/* • helpers/ATM.sol • helpers/EETHDepositHelper.sol • helpers/adapter/ArbitrumBridgeAdapter.sol • helpers/adapter/MantaPacificBridgeAdapter.sol • helpers/adapter/ModeBridgeAdapter.sol • helpers/adapter/OptimismBridgeAdapter.sol
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of 1.2 person weeks. The assessment was conducted over the course of six calendar days.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Hojung Han
Engineer
hojung@zellic.io

SeungHyeon Kim
Engineer
seunghyeon@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

July 10, 2024	Start of primary review period
July 11, 2024	Kick-off call
July 15, 2024	End of primary review period

3. Detailed Findings

3.1. Partial loss of ETH due to usage of the `outboundTransfer` function

Target	src/helpers/adapter/ArbitrumBridgeAdapter.sol		
Category	Protocol Risks	**Severity**	High
Likelihood	High	**Impact**	High

Description

In the `ArbitrumBridgeAdapter` contract, using the `outboundTransfer` function instead of the `outboundTransferCustomRefund` function results in a partial loss of ETH that should be refunded. The `outboundTransfer` function internally calls the `outboundTransferCustomRefund` function, setting the `_refundTo` address as the `to` address.

```
/**
 * @notice DEPRECATED - look at outboundTransferCustomRefund instead
 */
function outboundTransfer(
    address _l1Token,
    address _to,
    uint256 _amount,
    uint256 _maxGas,
    uint256 _gasPriceBid,
    bytes calldata _data
) public payable override returns (bytes memory res) {
    return
        outboundTransferCustomRefund(_l1Token, _to, _to, _amount, _maxGas,
    _gasPriceBid, _data);
}
```

The address receiving assets through the `ArbitrumBridgeAdapter` contract is the `BasicVault` contract in Arbitrum. However, the `BasicVault` contract is designed in such a way that it cannot send Ethereum to the ATM contract or allow the owner to withdraw the accumulated Ethereum native tokens, resulting in permanently frozen Ethereum native tokens.

Impact

The `BasicVault` contract accumulates Ethereum native tokens that are effectively locked and cannot be accessed or transferred. This issue was confirmed through simulation of past transactions, showing the accumulation of Ethereum native tokens in the deployed contract when assets were bridged.

Recommendations

It is recommended to replace the use of the `outboundTransfer` function with the `outboundTransferCustomRefund` function in the ArbitrumBridgeAdapter contract and to designate the ATM contract on L2 as the refund address to ensure that excess gas fees are properly refunded and accessible.

Remediation

This issue was fixed by Mitosis in commit 84a660131a ↗.

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. Inefficiency of `setEpochCap`

During our security audit, a discussion arose about the implementation of the `setEpochCap` function in the Cap contract. The primary concern is whether this approach is optimal from an administrative perspective.

In the current implementation, if the admin sets capacities for multiple epochs (e.g., 10 for Epoch 1, 20 for Epoch 2, and so on), reducing the capacity for a later epoch (e.g., from 50 to 25 for Epoch 5) requires sequentially updating all preceding epochs to maintain the sequential order. This process can be cumbersome, especially when dealing with a large number of epochs.

Eddy Hong from Mitosis responded, emphasizing that maintaining the sequential order of the epoch caps is more important than the inconvenience or cost on the owner's side. According to Eddy, ensuring this guarantee outweighs the administrative overhead.

4.2. Gas limit in OP Stack–related bridge adapters

In Op Stack–related bridges, setting the gas limit too low can cause transactions to fail on L2, risking funds being locked in the L1 Bridge contract. While reviewing the provided deployment script, the current gas limit setting does not appear problematic. However, this could become an issue in future operations.

It is important to remain cautious about this potential problem, especially as transactions may evolve to require more gas. Although this point is not considered a finding in this report, it is noted here to ensure it is kept in mind for future reference and potential adjustments.

4.3. Reentrancy on the EETHDepositHelper contract

For the EETHDepositHelper contract, we noticed many reentrancy vectors that are accessible by any user. However, the Mitosis team confirmed that this contract will not hold any tokens. Therefore, reentrancy attacks on this contract cannot drain any tokens. Following the Mitosis team's design, we have concluded that there are no security issues arising from reentrancy for this contract. Thus, we have moved this finding to the discussion section just to mention the possibility.

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: ArbitrumBridgeAdapter.sol

Function: `bridgeAsset(address destAddr, address l1Asset, byte[32], uint256 amount)`

This function is for bridging the asset.

Inputs

- `destAddr`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The destination address.
- `l1Asset`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The token to bridge.
- `<unnamed@2>`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: None.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The function caller must have enough tokens to transfer.
  - **Impact**: The amount to bridge.

Branches and code coverage

Intended branches

- Borrow some Ether from ATM and try to bridge the token.
  - ☑ Test coverage

Negative behavior

- Revert when the function caller is not in the allowlist.

- ☑ Negative test
- Revert when the function caller does not hold enough tokens.
  - ☑ Negative test

5.2. Module: BasicVault.sol

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

This function is for depositing using the ECDSA.

Inputs

- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Amount to deposit.
- `receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **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

- Take the proper amount of the tokens from the caller.
  - ☑ Test coverage
- Mint the vault tokens to the receiver.
  - ☑ 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.

Inputs

- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Amount to deposit.
- `receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Address of the receiver.

Branches and code coverage

Intended branches

- Take the proper amount of tokens from the caller.
  - ☑ Test coverage
- Mint the vault tokens to the receiver.
  - ☑ Test coverage

Negative behavior

- Revert when the deposit action status is halted.
  - ☑ Negative test

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

This function is for redeeming.

Inputs

- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The caller must have enough tokens.
  - **Impact**: Amount to redeem.
- `receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Address of the receiver.

5.3. Module: EETHDepositHelper.sol

Function: `depositTo(RemoteChainType remoteChain, address ccdm, uint256 amount, address receiver, address refundTo)`

This function is for depositing to the remote chain with setting of the token receiver and the refund address.

Inputs

- `remoteChain`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Type of the remote chain.
- `ccdm`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: CCDM host address to call the deposit function.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The caller must have enough eETH tokens.
  - **Impact**: Amount to deposit.
- `receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Address to receive the deposited tokens.
- `refundTo`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Address to refund.

Function: `depositTo(RemoteChainType remoteChain, address ccdm, uint256 amount, bytes permitData)`

This function is for depositing to the remote chain using the ECDSA.

Inputs

- `remoteChain`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.

- **Impact**: Type of the remote chain.

- `ccdm`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: CCDM host address to call the deposit function.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The caller must have enough eETH tokens.
  - **Impact**: Amount to deposit.
- `permitData`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must verify the caller using the given ECDSA.
  - **Impact**: ECDSA information with the deadline.

Function: `depositTo(RemoteChainType remoteChain, address ccdm, uint256 amount, address receiver, address refundTo, bytes permitData)`

This function is for depositing to the remote chain using the ECDSA with setting of the token receiver and the refund address.

Inputs

- `remoteChain`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Type of the remote chain.
- `ccdm`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: CCDM host address to call the deposit function.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The caller must have enough eETH tokens.
  - **Impact**: Amount to deposit.
- `receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Address to receive the deposited tokens.
- `refundTo`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.

    - **Impact**: Address to refund to.
- `permitData`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must verify the caller using the given ECDSA.
    - **Impact**: ECDSA information with the deadline.

Function: `depositTo(RemoteChainType remoteChain, address ccdm, uint256 amount)`

This function is for depositing to the remote chain.

Inputs

- `remoteChain`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: Type of the remote chain.
- `ccdm`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: CCDM host address to call the deposit function.
- `amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: The caller must have enough eETH tokens.
    - **Impact**: Amount to deposit.

Function: `depositWeEthTo(RemoteChainType remoteChain, address ccdm, uint256 amount, bytes permitData)`

This function is for depositing to the remote chain using the weETH tokens with the ECDSA and setting of the token receiver.

Inputs

- `remoteChain`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: Type of the remote chain.
- `ccdm`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.

    - **Impact**: CCDM host address to call the deposit function.
- `amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: The caller must have enough weETH tokens.
    - **Impact**: Amount to deposit.
- `receiver`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: Address to receive the deposited tokens.
- `permitData`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must verify the caller using the given ECDSA.
    - **Impact**: ECDSA information with the deadline.

Function: `depositWeEthTo(RemoteChainType remoteChain, address ccdm, uint256 amount, address receiver, address refundTo, bytes permitData)`

This function is for depositing to the remote chain using the weETH tokens with the ECDSA and setting of the token receiver and the refund address.

Inputs

- `remoteChain`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: Type of the remote chain.
- `ccdm`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: CCDM host address to call the deposit function.
- `amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: The caller must have enough weETH tokens.
    - **Impact**: Amount to deposit.
- `receiver`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: Address to receive the deposited tokens.
- `refundTo`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.

    - **Impact**: Address to refund to.
- `permitData`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must verify the caller using the given ECDSA.
    - **Impact**: ECDSA information with the deadline.

Function: `depositWeEthTo(RemoteChainType remoteChain, address ccdm, uint256 amount)`

This function is for depositing to the remote chain using the weETH tokens.

Inputs

- `remoteChain`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: Type of the remote chain.
- `ccdm`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: CCDM host address to call the deposit function.
- `amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: The caller must have enough weETH tokens.
    - **Impact**: Amount to deposit.

Function: `depositWeEthTo(RemoteChainType remoteChain, address ccdm, uint256 amount, address receiver, address refundTo)`

This function is for depositing to the remote chain using the weETH tokens with the setting of the token receiver and the refund address.

Inputs

- `remoteChain`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: Type of the remote chain.
- `ccdm`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.

- **Impact**: CCDM host address to call the deposit function.

- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The caller must have enough weETH tokens.
  - **Impact**: Amount to deposit.
- `receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Address to receive the deposited tokens.
- `refundTo`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Address to refund to.

Function: `depositWeEth(uint256 amount, address vault, bytes permitData)`

This function is for depositing using the weETH tokens with the ECDSA.

Inputs

- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The caller must have enough weETH tokens.
  - **Impact**: Amount to deposit.
- `vault`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Address to deposit to.
- `permitData`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must verify the caller using the given ECDSA.
  - **Impact**: ECDSA information with the deadline.

Function: `depositWeEth(uint256 amount, address vault)`

This function is for depositing using the weETH tokens.

Inputs

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

- **Constraints**: The caller must have enough weETH tokens.
- **Impact**: Amount to deposit.

- `vault`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Address to deposit to.

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

This function is for depositing using eETH.

Inputs

- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The caller must have enough eETH tokens.
  - **Impact**: Amount to deposit.
- `vault`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Address to deposit to.

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

This function is for depositing using eETH with the ECDSA.

Inputs

- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The caller must have enough eETH tokens.
  - **Impact**: Amount to deposit.
- `vault`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Address to deposit to.
- `permitData`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must verify the caller using the given ECDSA.
  - **Impact**: ECDSA information with the deadline.

Function: `redeem(uint256 amount, address vault)`

This function is for redeeming from the vault.

Inputs

- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The caller must have enough vault tokens.
  - **Impact**: Amount to redeem.
- `vault`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Address of the vault.

Function: `redeem(uint256 amount, address vault, bytes permitData)`

This function is for redeeming from the vault using the ECDSA.

Inputs

- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The caller must have enough vault tokens.
  - **Impact**: Amount to redeem.
- `vault`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Address of the vault.
- `permitData`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must verify the caller using the given ECDSA.
  - **Impact**: ECDSA information with the deadline.

5.4. Module: MantaPacificBridgeAdapter.sol

Function: `bridgeAsset(address destAddr, address l1Asset, byte[32] l2Asset, uint256 amount)`

This function is for bridging the asset.

Inputs

- `destAddr`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address to receive in the L2.
- `l1Asset`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The L1 token to bridge.
- `l2Asset`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The L2 token to bridge.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The function caller must have enough tokens to transfer.
  - **Impact**: The amount to bridge.

Branches and code coverage

Intended branches

- Bridge the token using the given input.
  - ☑ Test coverage

Negative behavior

- Revert when the function caller is not in the allowlist.
  - ☑ Negative test
- Revert when the function caller does not hold enough tokens.
  - ☑ Negative test

5.5. Module: ModeBridgeAdapter.sol

Function: `bridgeAsset(address destAddr, address l1Asset, byte[32] l2Asset, uint256 amount)`

This function is for bridging the asset.

Inputs

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

    - **Constraints**: None.
    - **Impact**: The address to receive in the L2.
- `l1Asset`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The L1 token to bridge.
- `l2Asset`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The L2 token to bridge.
- `amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: The function caller must have enough tokens to transfer.
    - **Impact**: The amount to bridge.

Branches and code coverage

Intended branches

- Bridge the token using the given input.
    - ☑ Test coverage

Negative behavior

- Revert when the function caller is not in the allowlist.
    - ☑ Negative test
- Revert when the function caller does not hold enough tokens.
    - ☑ Negative test

5.6. Module: OptimismBridgeAdapter.sol

Function: `bridgeAsset(address destAddr, address l1Asset, byte[32] l2Asset, uint256 amount)`

This function is for bridging the asset.

Inputs

- `destAddr`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The address to receive in the L2.
- `l1Asset`

    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The L1 token to bridge.
- `l2Asset`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The L2 token to bridge.
- `amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: The function caller must have enough tokens to transfer.
    - **Impact**: The amount to bridge.

Branches and code coverage

Intended branches

- Bridge the token using the given input.
    - ☑ Test coverage

Negative behavior

- Revert when the function caller is not in the allowlist.
    - ☑ Negative test
- Revert when the function caller does not hold enough tokens.
    - ☑ Negative test

6. Assessment Results

At the time of our assessment, the reviewed code was deployed to the Ethereum Mainnet and L2s.

During our assessment on the scoped Mitosis Vault contracts, we discovered one finding, which was of high impact.

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.