Zellic

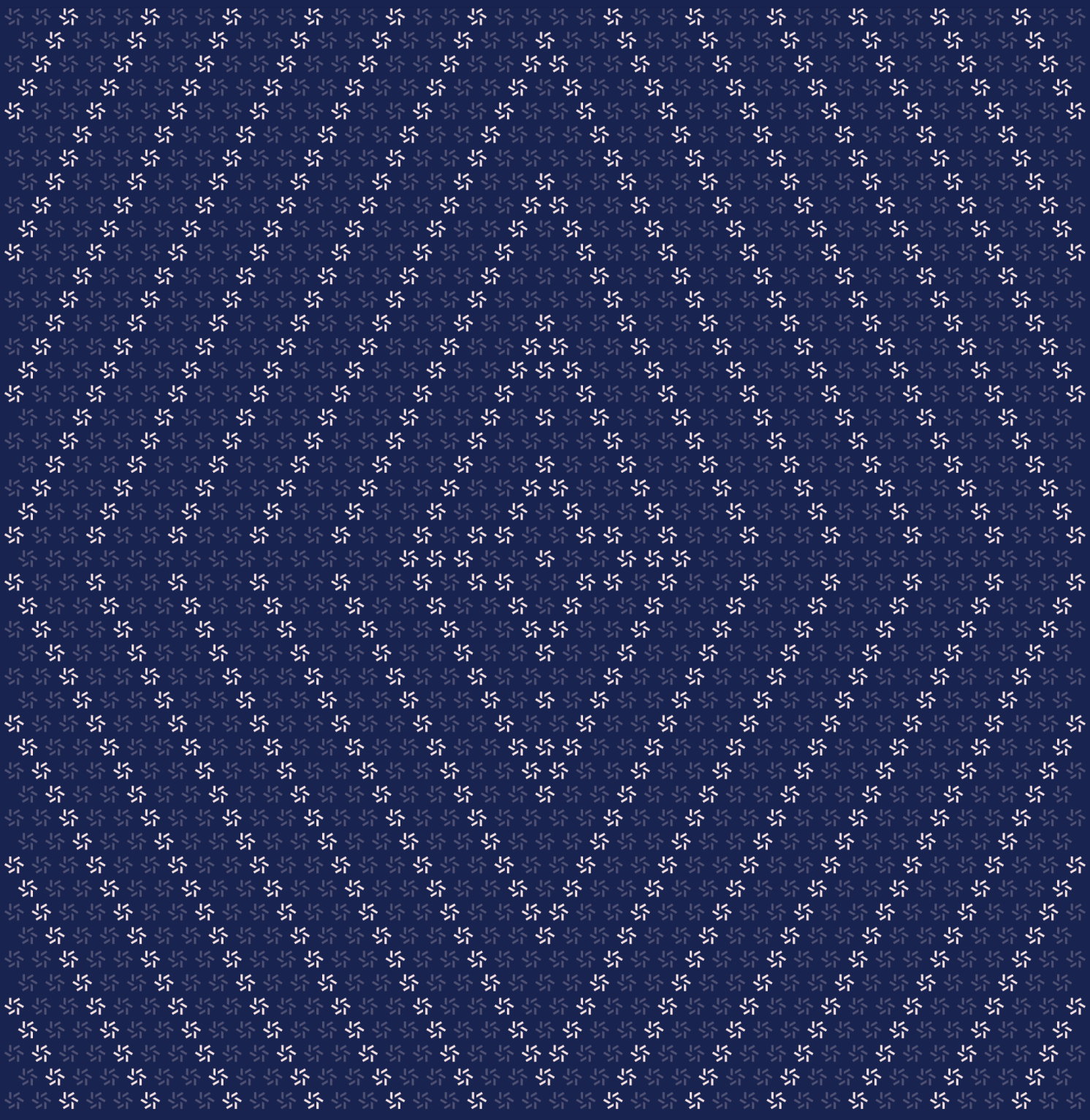
Prepared for
Infini
Infini Labs

Prepared by
Qingying Jie
Jisub Kim
Zellic

October 21, 2024

InfiniCard 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 Infini Labs from October 14th to October 15th, 2024. During this engagement, Zellic reviewed InfiniCard 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 an on-chain attacker drain the vaults?
- Are the strategy and custodian whitelists properly implemented to prevent unauthorized contracts or addresses from interacting with the vault?
- Are there any functions that allow unauthorized access to user funds within the vault?

1.3. Non-goals and Limitations

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

- Front-end components
- Backend 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.

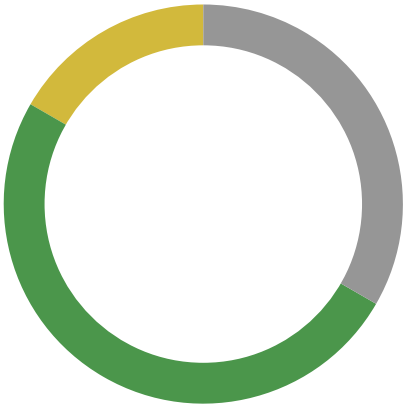
1.4. Results

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

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	3
Informational	2



2. Introduction

2.1. About InfiniCard Vault

Infini Labs contributed the following description of InfiniCard Vault:

> InfiniCard Vault is a centrally managed contract platform designed to efficiently manage and grow assets through various strategies. The platform supports multiple strategies, including Ethena, Morpho and so on, allowing Infini Card Service to deposit, withdraw, and settle profits.

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:

InfiniCard Vault Contracts

Type	Solidity
Platform	EVM-compatible
Target	infini-card-contract
Repository	https://github.com/infini-money/infini-card-contract ↗
Version	`38ea00f1dbfd79091689a51a6656a627916decd5`

Programs	`library/StrategyUtils.sol`
	`library/VaultUtils.sol`
	`strategys/BaseStrategyVault.sol`
	`strategys/BaseStrategyManager.sol`
	`strategys/ethena/InfiniEthenaStrategyManager.sol`
	`strategys/ethena/InfiniEthenaStrategyVault.sol`
	`strategys/morpho/InfiniMorphoStrategyVault.sol`
	`InfiniCardController.sol`
	`InfiniCardVault.sol`

2.4. Project Overview

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

The following consultants were engaged to conduct the assessment:

Qingying Jie
Engineer
qingying@zellic.io

Jisub Kim
Engineer
jisub@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

October 14, 2024	Start of primary review period
October 15, 2024	End of primary review period

3. Detailed Findings

3.1. Missing function to remove a delegated signer

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

Description

The EthenaMinting contract supports minting and redeeming USDe. The `mint` and `redeem` functions require a signature from the order, signed by the benefactor or delegated signers. Tokens are taken from the benefactor and sent to the beneficiary after a specific operation.

The InfiniEthenaStrategyVault contract includes a function to set desired delegated signers, allowing multiple signers to be delegated. However, there is no function to remove delegated signers.

Impact

If delegated signers become malicious, they could sign orders for which they are the beneficiary, potentially draining funds approved for EthenaMinting within the InfiniEthenaStrategyVault.

Recommendations

Consider adding a function to remove a delegated signer.

Remediation

This issue has been acknowledged by Infini Labs, and a fix was implemented in commit `e12e0666` ↗.

3.2. Missing function to remove tokens from the whitelist

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

Description

Within the InfiniCardController contract, the `addStrategy` function serves a dual purpose; it adds the specified strategy to the strategy whitelist and simultaneously includes the associated underlying token address in the token whitelist. This design ensures that only approved strategies and their corresponding tokens are recognized and utilized by the vault.

```
function addStrategy(address strategy) onlyRole(ADMIN_ROLE) external {
    strategyWhiteList[strategy] = true;
    _addToken(IStrategyVault(strategy).underlyingToken());
}
```

However, the `removeStrategy` function currently only removes the specified strategy from the strategy whitelist without addressing the underlying tokens. Given that multiple strategies can share the same underlying token, simply removing a strategy does not automatically reflect the token's usage status across all strategies. As a result, the `tokenWhiteList` may retain tokens that are no longer associated with any active strategy.

```
function removeStrategy(address strategy) onlyRole(ADMIN_ROLE) external {
    strategyWhiteList[strategy] = false;
}
```

Impact

If a token is no longer used by any strategy but remains in the token whitelist, and the InfiniCardVault contract holds some of that token, it can be drained from the contract using the `withdrawToCEX` function. This creates a potential security risk where unused tokens can be maliciously withdrawn.

Recommendations

Consider adding a function to remove unused underlying tokens from the `tokenWhiteList` and `tokenList`.

Remediation

This issue has been acknowledged by Infini Labs, and a fix was implemented in commit `773ef3b0`↗.

3.3. Unset state variable `custodianList`

Target	InfiniCardController, InfiniCardVault		
Category	Business Logic	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

The contract InfiniCardController contains an array-type state variable, `custodianList`. Additionally, the contract InfiniCardVault includes the function `getCustodianList`, which reads and returns this array. However, there is no implementation to set the `custodianList` array value.

Impact

While there is a mapping variable that records the whitelist of custodians, the associated array `custodianList` remains empty. This inconsistency can lead to confusion and misalignment between the actual whitelist and the data returned by `getCustodianList`.

Recommendations

Consider updating the array `custodianList` in functions `addCusdianToWhiteList` and `removeCusdianToWhiteList`.

Remediation

This issue has been acknowledged by Infini Labs, and a fix was implemented in commit `0f710ced` ↗.

3.4. Mismatched function parameter types affecting inheritance

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

Description

The interface IStrategyVault defines the functions `deposit` and `redeem` with parameters of types `uint256` and `bytes`:

```
function deposit(uint256 _amount, bytes calldata depositInfo) external;
function redeem(uint256 _amount, bytes calldata redeemInfo)
    external returns (uint256);
```

However, the contract BaseStrategyVault implements these functions with only `uint256` parameters:

```
function deposit(uint256 _amount) virtual external {}
function redeem(uint256 _amount)
    virtual external returns (uint256 actualRedeemedAmount) {}
```

Impact

This discrepancy causes the BaseStrategyVault contract and its child contracts to have two versions of each `deposit` and `redeem` function with different parameter types. Only one version of each function will function correctly, potentially leading to confusion and introducing bugs.

Recommendations

Consider updating the parameter types of functions `deposit` and `redeem` in the contract BaseStrategyVault to (`uint256`, `bytes`).

Remediation

This issue has been acknowledged by Infini Labs, and a fix was implemented in commit `e4d3944d` ↗.

3.5. Centralization risks

Target	InfiniCardVault		
Category	Business Logic	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

There are four privileged roles for the contract InfiniCardVault:

1. The default admin role is able to grant other roles to accounts and revoke roles from accounts.
2. The admin role has the ability to modify the strategy whitelists, custodian whitelists, and token whitelists.
3. The strategy operator role is able to interact with the valid strategies using associated underlying tokens and share tokens in the contract.
4. The Infini backend role can withdraw whitelisted tokens from the contract.

Impact

The above introduces centralization risks that users should be aware of, as it grants a single point of control over the system.

Recommendations

We recommend clearly documenting this centralized design to inform users about the owner's control over the contract. This transparency enables users to make informed decisions about their participation in the project. Additionally, outlining the specific circumstances under which the owner may exercise these powers can build trust and enhance transparency.

To further mitigate these risks, we suggest implementing measures such as a multi-signature requirement for owner access.

Remediation

This issue has been acknowledged by Infini Labs.

3.6. Unused variable

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

Description

There is an unused variable, `WITHDRAWOUT_STRATEGY_ADDRESS`, in the contract InfiniCardController.

```
address public constant WITHDRAWOUT_STRATEGY_ADDRESS = address(0);
```

Impact

It may compromise code readability and could consume more gas fees.

Recommendations

Consider removing the unused function or integrating the variable into the contract logic to ensure consistency and avoid potential confusion.

Remediation

This issue has been acknowledged by Infini Labs, and a fix was implemented in commit `96af3c4e` ↗.

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. Typographical errors

Upon reviewing the codebase, several minor typographical errors were identified. They do not affect code functionality but can lead to confusion and potential bugs.

In the contract InfiniCardVault,

- The function `getCusdianList` should be `getCustodianList`.

In the contract InfiniCardController,

- The error `CustianInvalid` should be `CustodianInvalid`.
- The word `STRATEGY` is repeated in `STRATEGY_STRATEGY_OPERATOR_ROLE`.
- The function `addCusdianToWhiteList` should be `addCustodianToWhiteList`.
- The function `removeCusdianToWhiteList` should be `removeCustodianToWhiteList`.
- The function `_isCusdianValid` should be `_isCustodianValid`.

In the interface IStrategyManager,

- In the `StrategyStatus` struct, the variable `poistion` should be `position`.

This issue has been acknowledged by Infini Labs, and a fix was implemented in commit `e3bf1fe7`↗.

4.2. Lack of comprehensive test suite

During this audit, we observed a number of findings that affect the core logic of the codebase. Some of these findings could result in the failure of a working production environment, even if no malicious attack is assumed.

When building a complex contract ecosystem with multiple moving parts and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios. Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch.

The test coverage for this project should be expanded to include more than just surface-level functions. It is important to test the invariants required for ensuring security.

Good test coverage has multiple effects:

- It finds bugs and design flaws early (pre-audit or pre-release).
- It gives insight into areas for optimization (e.g., gas cost).
- It displays code maturity.
- It bolsters customer trust in your product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.
- It increases development velocity long-term.

The last point seems contradictory, given the time investment to create and maintain tests. To expand upon that, tests help developers trust their own changes. It is difficult to know if a code refactor — or even just a small one-line fix — breaks something if there are no tests. This is especially true for new developers or those returning to the code after a prolonged absence. Tests have your back here. They are an indicator that the existing functionality *most likely* was not broken by your change to the code.

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: BaseStrategyVault.sol

Function: `withdraw(address token, uint256 amount)`

This function allows the InfiniCard Vault to withdraw a specified amount of the token from the strategy vault.

Inputs

- `token`
  - **Control**: Controlled by the caller with `INFINI_CARD_VAULT`.
  - **Constraints**: None.
  - **Impact**: The token to withdraw.
- `amount`
  - **Control**: Controlled by the caller with `INFINI_CARD_VAULT`.
  - **Constraints**: None.
  - **Impact**: The amount of tokens to withdraw.

Branches and code coverage

Intended branches

- Retrieves the vault's balance of the token.
  - ☐ Test coverage
- Sets `actualAmount` to `vaultBalance` if the vault's balance is smaller than the amount; otherwise, it sets `actualAmount` to `amount`.
  - ☐ Test coverage
- Transfers the `actualAmount` of tokens from the vault.
  - ☐ Test coverage

Negative behavior

- Reverts if the caller does not have the `INFINI_CARD_VAULT` role.
  - ☐ Negative test

5.2. Module: InfiniCardVault.sol

Function: `invest(address strategy, uint256 amount, bytes investmentInfo)`

This function allows the strategy operator to invest a specified amount of tokens into a strategy.

Inputs

- `strategy`
  - **Control**: Controlled by the caller with `STRATEGY_OPERATOR_ROLE`.
  - **Constraints**: None.
  - **Impact**: The strategy in which to invest funds.
- `amount`
  - **Control**: Controlled by the caller with `STRATEGY_OPERATOR_ROLE`.
  - **Constraints**: None.
  - **Impact**: The amount to invest into the strategy.
- `investmentInfo`
  - **Control**: Controlled by the caller with `STRATEGY_OPERATOR_ROLE`.
  - **Constraints**: None.
  - **Impact**: Additional data passed to the strategy when depositing tokens.

Branches and code coverage

Intended branches

- Deposits underlying tokens to the given vault.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller does not have `STRATEGY_OPERATOR_ROLE`.
  - ☑ Negative test
- Reverts if strategy is not in the strategy whitelist.
  - ☑ Negative test

Function: `redeem(address strategy, uint256 amount, bytes redeemInfo)`

This function allows the strategy operator to redeem a specified amount from a strategy back to the vault.

Inputs

- `strategy`
  - **Control**: Controlled by the caller with `STRATEGY_OPERATOR_ROLE`.
  - **Constraints**: None.
  - **Impact**: The strategy from which to redeem funds.
- `amount`
  - **Control**: Controlled by the caller with `STRATEGY_OPERATOR_ROLE`.
  - **Constraints**: None.
  - **Impact**: The amount to redeem from the strategy.
- `redeemInfo`
  - **Control**: Controlled by the caller with `STRATEGY_OPERATOR_ROLE`.
  - **Constraints**: None.
  - **Impact**: Additional data passed to the strategy when redeeming tokens.

Branches and code coverage

Intended branches

- Redeems underlying tokens from the given vault.
  - ☐ Test coverage

Negative behavior

- Reverts if the caller does not have `STRATEGY_OPERATOR_ROLE`.
  - ☐ Negative test
- Reverts if strategy is not in the strategy whitelist.
  - ☐ Negative test

Function: `withdrawFromStrategy(address strategy, uint256 amount, bytes redeemInfo)`

This function allows the strategy operator to withdraw a specified amount from a strategy back to the vault.

Inputs

- `strategy`
  - **Control**: Controlled by the caller with `STRATEGY_OPERATOR_ROLE`.
  - **Constraints**: None.
  - **Impact**: the strategy from which to withdraw funds.
- `amount`
  - **Control**: Controlled by the caller with `STRATEGY_OPERATOR_ROLE`.

- **Constraints**: None.
- **Impact**: The amount to withdraw from the strategy.

- `redeemInfo`
  - **Control**: Controlled by the caller with `STRATEGY_OPERATOR_ROLE`.
  - **Constraints**: None.
  - **Impact**: Additional data passed to the strategy when redeeming tokens.

Branches and code coverage

Intended branches

- Withdraws the amount of tokens from the strategy vault.
  - ☐ Test coverage

Negative behavior

- Reverts if the caller does not have `STRATEGY_OPERATOR_ROLE`.
  - ☐ Negative test

Function call analysis

- `_withdraw_from_strategy(strategy, amount, redeemInfo)`
  - **What is controllable?** `strategy`, `amount`, and `redeemInfo`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.

Function: `withdrawToCEX(address token, uint256 amount, address custodian, address strategy, bytes redeemInfo)`

This function allows the backend to withdraw a specified amount of a token to a custodian address. If the vault's balance of the token is insufficient, it attempts to withdraw the required amount from a specified strategy. The function ensures that both the token and the custodian are whitelisted.

Inputs

- `token`
  - **Control**: Controlled by the caller with `INFINI_BACKEND_ROLE`.
  - **Constraints**: None.
  - **Impact**: The token to be withdrawn to the custodian.
- `amount`
  - **Control**: Controlled by the caller with `INFINI_BACKEND_ROLE`.

    - **Constraints**: None.
    - **Impact**: The amount of tokens to withdraw.
- `custodian`
    - **Control**: Controlled by the caller with `INFINI_BACKEND_ROLE`.
    - **Constraints**: None.
    - **Impact**: The recipient of the withdrawn tokens.
- `strategy`
    - **Control**: Controlled by the caller with `INFINI_BACKEND_ROLE`.
    - **Constraints**: None.
    - **Impact**: Source strategy from which to withdraw tokens.
- `redeemInfo`
    - **Control**: Controlled by the caller with `INFINI_BACKEND_ROLE`.
    - **Constraints**: None.
    - **Impact**: Additional data passed to the strategy when redeeming tokens.

Branches and code coverage

Intended branches

- Tries to withdraw from the strategy if the balance is not enough.
    - ☐ Test coverage
- Calculates `actualAmount` based on the vault and transfers the tokens to the custodian.
    - ☑ Test coverage

Negative behavior

- Reverts if the caller does not have `INFINI_BACKEND_ROLE`.
    - ☐ Negative test
- Reverts if the `token` is not in the token whitelist.
    - ☐ Negative test
- Reverts if `custodian` is not in the custodian whitelist.
    - ☐ Negative test
- Reverts if the balance is insufficient and `strategy` is zero address.
    - ☐ Negative test
- Reverts if `underlyingToken` from the strategy does not match `token`.
    - ☐ Negative test

Function call analysis

- `_withdraw_from_strategy(strategy, amount, redeemInfo)`
    - **What is controllable?** `strategy`, `amount`, and `redeemInfo`.
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.

    - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.
- `_transferAsset(token, actualAmount, custodian)`
    - **What is controllable?** `token`, `actualAmount`, and `custodian`.
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.

5.3. Module: InfiniEthenaStrategyManager.sol

Function: `settle(uint256 unSettleProfit)`

This function settles a specified amount of profit, distributing it between the protocol's treasury and the strategy vault according to the carry rate.

Inputs

- `unSettleProfit`
    - **Control**: Controlled by the caller with `ADMIN_ROLE`.
    - **Constraints**: None.
    - **Impact**: The amount of profit to be settled and distribute.

Branches and code coverage

Intended branches

- Distributes profit between the protocol's treasury and the strategy vault according to the carry rate.
    - ☑ Test coverage

Negative behavior

- Reverts if the caller does not have the `ADMIN` role.
    - ☐ Negative test
- Reverts if `unSettleProfit` is bigger than the current profit of the vault.
    - ☐ Negative test

5.4. Module: InfiniMorphoStrategyVault.sol

Function: `deposit(uint256 amount, bytes)`

This function allows the InfiniCard Vault to deposit a specified amount of underlying tokens into the strategy.

Inputs

- `amount`
  - **Control**: Controlled by the caller with `INFINI_CARD_VAULT`.
  - **Constraints**: None.
  - **Impact**: The amount of tokens will be deposited into the strategy.
- `bytes`
  - **Control**: Controlled by the caller with `INFINI_CARD_VAULT`.
  - **Constraints**: None.
  - **Impact**: Not used.

Branches and code coverage

Intended branches

- Deposits the amount of tokens when sufficient underlying tokens are available.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller does not have the `INFINI_CARD_VAULT` role.
  - ☐ Negative test
- Reverts if the balance of underlying tokens is insufficient.
  - ☐ Negative test

Function: `getProfit()`

This function calculates and returns the current profit of the vault by comparing the total position value with the vault's recorded position.

Branches and code coverage

Intended branches

- Calculates profit when `totalPosition` is greater than or equal to `vaultPosition`.
  - ☐ Test coverage

Negative behavior

- Revert if `vaultPosition` exceeds `totalPosition`.
  - ☐ Negative test

Function: `getStrategyStatus()`

This function provides the current status of the strategy, including the position, profit, underlying token, and strategy address.

Branches and code coverage

Intended branches

- Constructs and returns the `StrategyStatus` struct.
  - ☐ Test coverage

Function: `redeem(uint256 amount, bytes)`

This function allows the InfiniCard Vault to redeem a specified amount of underlying tokens from the strategy.

Inputs

- `amount`
  - **Control**: Controlled by the caller with `INFINI_CARD_VAULT`.
  - **Constraints**: None.
  - **Impact**: The amount of tokens will be redeemed from the strategy.
- `bytes`
  - **Control**: Controlled by the caller with `INFINI_CARD_VAULT`.
  - **Constraints**: None.
  - **Impact**: Not used.

Branches and code coverage

Intended branches

- Redeems when `amount` is within `vaultPosition` and sufficient shares are available.
  - ☐ Test coverage

Negative behavior

- Reverts if the caller does not have the `INFINI_CARD_VAULT` role.
  - ☐ Negative test
- Reverts when `amount` exceeds `vaultPosition`.
  - ☐ Negative test
- Reverts when the contract lacks enough share tokens.
  - ☐ Negative test

Function call analysis

- `IERC4626(shareToken).convertToShares(amount)`
  - **What is controllable?** `amount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `IERC4626(market).redeem(shouldRedeemSharesAmount, address(this), address(this))`
  - **What is controllable?** `shouldRedeemSharesAmount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `settle(uint256 unSettleProfit)`

This function settles a portion of the accumulated profits, distributing them between the protocol's treasury and reinvesting the remainder.

Inputs

- `unSettleProfit`
  - **Control**: Controlled by the caller with `ADMIN_ROLE`.
  - **Constraints**: None.
  - **Impact**: The amount of profit to be settled and distributed.

Branches and code coverage

Intended branches

- Distributes profit between the protocol's treasury and the strategy vault according to the carry rate.
  - ☐ Test coverage
- Redeposits the underlying tokens.
  - ☐ Test coverage

Negative behavior

- Reverts if the caller does not have the `ADMIN_ROLE` role.
  - ☐ Negative test
- Reverts if `unSettleProfit` is bigger than the current profit of the vault.
  - ☐ Negative test

Function call analysis

- `getProfit()`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `IERC4626(market).convertToShares(unSettleProfit)`
  - **What is controllable?** `unSettleProfit`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `IERC4626(market).redeem(protocolProfitShare, infiniTreasure, address(this))`
  - **What is controllable?** `unSettleProfit`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Not used.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `IERC4626(market).redeem(settleProfitShare, address(this), address(this))`
  - **What is controllable?** `settleProfitShare`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `_deposit(underlyingTokenAmount)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Not used.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

6. Assessment Results

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

During our assessment on the scoped InfiniCard Vault contracts, we discovered six findings. No critical issues were found. One finding was of medium impact, three were of low impact, and the remaining findings were 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.