Zellic

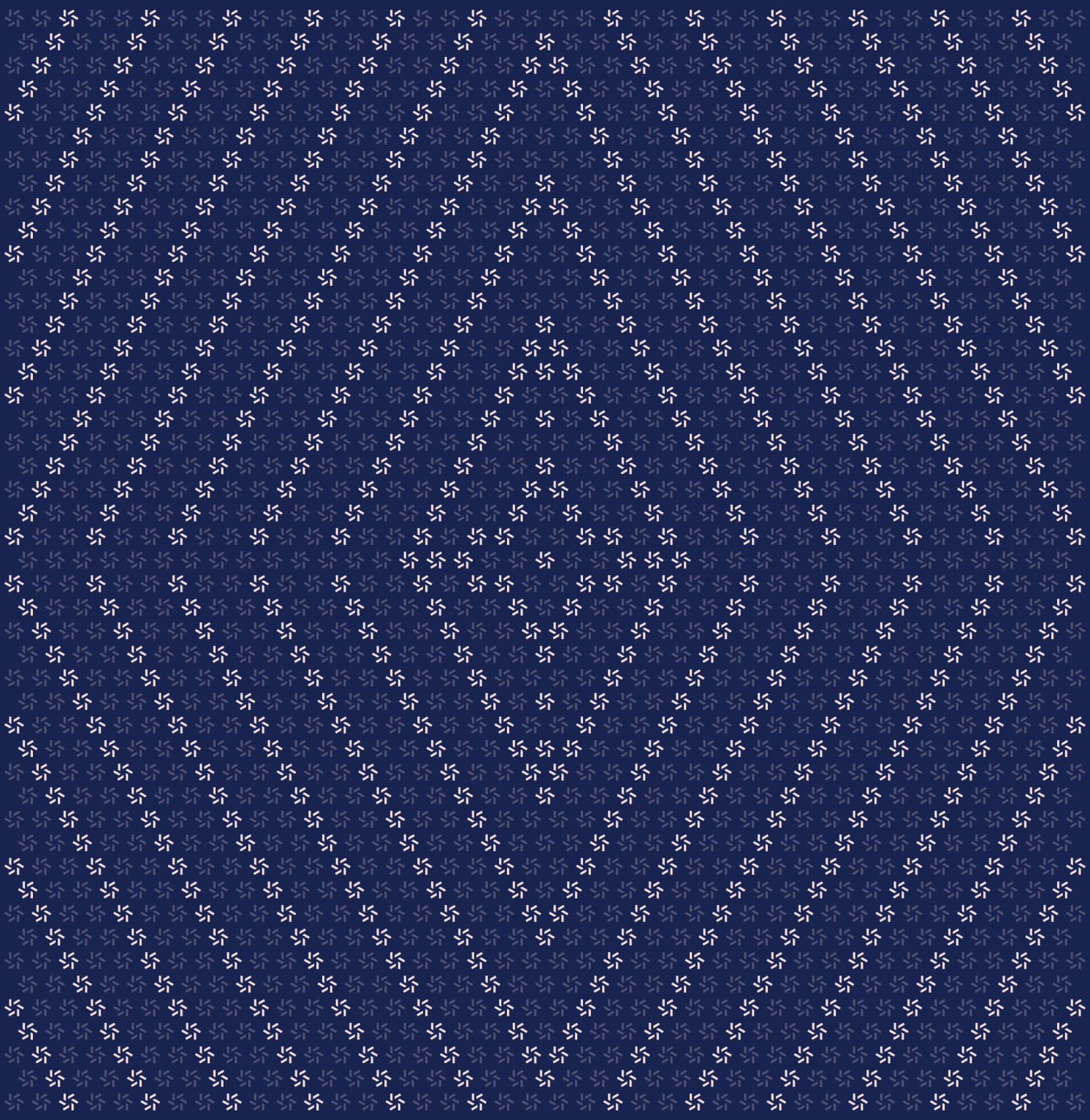
Prepared for
red (@infraredBear)
Infrared Finance

Prepared by
Seunghyeon Kim
Jaeeu Kim
Zellic

April 25, 2024

Infrared

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 Infrared Finance from April 15th, 2024 to April 25th, 2024. During this engagement, Zellic reviewed Infrared's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Could an attacker exploit issues related to assets?
- Are there any potential attack vectors involving reentrancy?
- Is the reward logic implemented correctly?

1.3. Non-goals and Limitations

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

- Deploy script of contracts
- 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 Infrared contracts, we discovered one finding, which was of low impact.

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

Breakdown of Finding Impacts

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

2. Introduction

2.1. About Infrared

Infrared Finance contributed the following description of Infrared:

> Infrared offers liquid staking in Berachain's Proof-of-Liquidity protocol, allowing users to stake their assets and receive IBGT, a liquid staked representation of BGT.

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:

Infrared Contracts

Repository	https://github.com/infrared-dao/infrared-contracts ↗
Version	infrared-contracts: `35b90e2657f2de7b2085487f3a07b57f31aec132`
Programs	• core/*.sol • utils/*.sol • vendors/*.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 4.5 person-days. The assessment was conducted over the course of 9 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:

Seunghyeon Kim
Engineer
seunghyeon@zellic.io ↗

Jaeeu Kim
Engineer
jaeeu@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

April 15, 2024	Kick-off call
April 15, 2024	Start of primary review period
April 25, 2024	End of primary review period

3. Detailed Findings

3.1. The `rewardRate` can be zero

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

Description

In the `_notifyRewardAmount` function, the reward is divided by `rewardsDuration`. If the reward amount is less than than `rewardsDuration`, it's possible for the `rewardRate` to be zero. Since the reward tokens have 18 decimals, the likelihood of the reward being less than the duration is small. However, there is a minor concern because `harvestVault` can be called by anyone at anytime, potentially causing the reward to be smaller than `rewardsDuration` when the function is called.

```
if (block.timestamp >= rewardData[_rewardsToken].periodFinish) {
    rewardData[_rewardsToken].rewardRate =
        reward.div(rewardData[_rewardsToken].rewardsDuration);
} else {
    uint256 remaining =
        rewardData[_rewardsToken].periodFinish.sub(block.timestamp);
    uint256 leftover =
        remaining.mul(rewardData[_rewardsToken].rewardRate);
    rewardData[_rewardsToken].rewardRate = reward.add(leftover).div(
        rewardData[_rewardsToken].rewardsDuration
    );
}
```

Impact

The `rewardRate` could be zero which prevents an increase in rewards at the time of `rewardRate` being zero.

Recommendations

We recommend adding logic which reverts when the reward is smaller than `rewardsDuration`.

Remediation

This issue has been acknowledged by Infrared Finance.

According to Infrared Finance, they plan to set one to two days for reward duration because the reward must be less than the `rewardsDuration`. The proper duration setting will prevent this issue from occurring.

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. Potential reentrancy with recoverERC20

The `recoverERC20` function in the Infrared and InfraredVault contracts is designed for recovering tokens that have been mistakenly sent to the contracts. This function accepts an arbitrary token address and does `safeTransfer` on the given token. We raised concerns about potential reentrancy attacks from malicious tokens.

During our discussions, the Infrared team clarified that this function is accessible only to the governor, whose actions are either determined by a council or directly voted on. This cannot be exploited in the commit hash we audited. However, we wanted to highlight this for future reference and as a reminder to future developers if this code is forked.

4.2. Permissionless migrate function

The `migrate` function is used to stake the `totalSupply` into Berachain's rewards vault. It's used for updating the vault address after deploying InfraredVault without a predefined `rewardsVault` address. The address is obtained from an immutable address of Berachain's factory, although the specific implementation of Berachain's reward-factory logic has not been finalized.

To mitigate potential risks that could move all funds, consider adding permission controls to this function.

```
function migrate() external {
    console.log(address(rewardsVault));
    if (stakedInRewardsVault()) revert Errors.StakedInRewardsVault();

    // set berachain rewards vault and set operator on vault as infrared
    address _rewardsVaultAddress =
        getRewardsVaultAddress(infrared, address(stakingToken));
    if (_rewardsVaultAddress == address(0)) {
        revert Errors.NoRewardsVault();
    }
    rewardsVault = IBerachainRewardsVault(_rewardsVaultAddress);
    _setOperator(infrared);

    // stake total supply for berachain proof of liquidity rewards
    stakingToken.safeIncreaseAllowance(_rewardsVaultAddress, _totalSupply);
    rewardsVault.stake(_totalSupply);
```

```
}
```

4.3. Undeveloped features in Berachain's contract have potential risks

Berachain's depositor contract, which is used in some validator related functions (`delegate` and `redelegate`) in the Infrared contract, has not yet been developed. This could lead to potential risks in the future. Although these functions require permissions, which reduces some risks, verifying and testing will be necessary after Berachain's contract is fully established.

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. Core functions

These core functions can be found at infrared-dao-infrared-contracts/contracts/src/core.

IBGT

This is the Infrared BGT token contract. It is a standard ERC-20 token contract to support the Infrared ecosystem.

Mint

The `mint` function is used to mint new tokens to the target address. The `MINTER_ROLE` caller could mint new tokens to the target address.

Pause/unpause

The `pause` and `unpause` functions are used to pause and unpause the token. The `PAUSER_ROLE` caller could pause/unpause the token.

Infrared

The admin is a trusted entity and has extensive control over the protocol. In this contract, functions are only accessible to users who hold specific roles, except for the `harvestVault` function. As these functions are set and called by roles with significant powers, they require special caution.

Register vault

The `registerVault` function is used to register the vault to the Infrared contract. The admin could call the function to register the vault to the Infrared contract.

This function calls `_registerVault` internally, which deploys the vault contract and sets the vault address to the Infrared contract.

The `registerVault` has modifiers.

- `onlyKeeper` modifier (the caller must have the KEEPER_ROLE role)
- `whenInitialized` modifier (the Infrared contract must be initialized)

The `registerVault` function checks that the vault address is not zero.

Update whitelisted rewards tokens

The `updateWhiteListedRewardTokens` function is used to update the whitelist token. The admin could call the function to update the whitelist token.

The `updateWhiteListedRewardTokens` has modifiers.

- `onlyGovernor` modifier (the caller must have the GOVERNANCE_ROLE role)
- `whenInitialized` modifier (the Infrared contract must be initialized)

Update reward duration

The `updateRewardsDuration` function is used to update the reward duration. The admin could call the function to update the reward duration.

The `updateRewardsDuration` has modifiers.

- `onlyGovernor` modifier (the caller must have the GOVERNANCE_ROLE role)
- `whenInitialized` modifier (the Infrared contract must be initialized)

The `updateRewardsDuration` function checks that the new reward duration is not zero.

Pause vault

The `pauseVault` function is used to pause the vault. The governor could call the function to pause the vault.

The `pauseVault` has modifiers.

- `onlyGovernor` modifier (the caller must have the GOVERNANCE_ROLE role)
- `whenInitialized` modifier (the Infrared contract must be initialized)

The `pauseVault` function checks that the vault address is valid.

Add validators

The `addValidators` function is used to add validators to the vault. The Governor role could set the validators and call the function to add validators to the vault.

The `addValidators` has modifiers.

- `onlyGovernor` modifier (the caller must have the GOVERNANCE_ROLE role)
- `whenInitialized` modifier (the Infrared contract must be initialized)

The `addValidators` function checks the following.

- The validator's pubkey is provided.
- The validator's pubkey is not already added.

Remove validators

The `removeValidators` function is used to remove validators from the vault. The Governor role could set the validators and call the function to remove validators from the vault.

The `removeValidators` has modifiers.

- `onlyGovernor` modifier (the caller must have the GOVERNANCE_ROLE role)
- `whenInitialized` modifier (the Infrared contract must be initialized)

The `removeValidators` function checks the following.

- The validator's pubkey is provided.
- The validator's pubkey already exists.

Replace validators

The `replaceValidator` function is used to replace validators in the vault. The Governor role could set the validators and call the function to replace validators in the vault.

The `replaceValidator` has modifiers.

- `onlyGovernor` modifier (the caller must have the GOVERNANCE_ROLE role)
- `whenInitialized` modifier (the Infrared contract must be initialized)

The `replaceValidator` function checks the following.

- The validator's pubkeys are provided.
- The old validator's pubkey is valid.
- The new validator's pubkey is not already added.

Delegate

The `delegate` function is used to delegate the stake to the validator. The user could call the function to delegate the stake to the validator.

This function calls a depositor's `deposit` internally. This function is one of Berachain's contract functions that are not implemented yet.

The `delegate` has modifiers.

- `onlyGovernor` modifier (the caller must have the GOVERNANCE_ROLE role)
- `whenInitialized` modifier (the Infrared contract must be initialized)

The `delegate` function checks the following.

- The amount is not zero.
- The validator's pubkey is valid.

Redelegate

The `redelegate` function is used to redelegate the stake to the validator. The user could call the function to redelegate the stake to the new validator.

This function calls a depositor's `redirect` internally. This function is one of Berachain's contract functions that are not implemented yet.

The `redelegate` has modifiers.

- `onlyGovernor` modifier (the caller must have the GOVERNANCE_ROLE role)
- `whenInitialized` modifier (the Infrared contract must be initialized)

The `redelegate` function checks the following.

- The amount is not zero.
- The validator's pubkey is valid.

Queue

The `queue` function is used to queue the stake to the validator. The user could call the function to queue to Berachain's chef contract.

This function calls chef's `queueNewCuttingBoard` internally. This function is one of Berachain's contract functions that are not implemented yet.

The `queue` has a modifier.

- `onlyGovernor` modifier (the caller must have the GOVERNANCE_ROLE role)

The `queue` function checks that the validator's pubkey is valid.

Harvest rewards

The `harvestVault` function is used to harvest the rewards from Berachain's reward vault. Is called to harvest the rewards from Berachain's reward vault.

This function is called by the `onReward` function in the MultiRewards contract as hook.

This function calls `getReward` of Berachain's reward-vault contract internally. This function claims the reward from Berachain's reward vault.

This function calls `_handleBGTRewards` internally, mints the BGT token to the user, and updates the protocol fee. Then it calls `notifyRewardAmount` of the vault contract to update the reward state with transferring IBGT to the vault. This function checks that the amount of reward is not zero.

The `harvestVault` has a modifier.

- `whenInitialized` modifier (the Infrared contract must be initialized)

The `harvestVault` function checks that the vault's reward token address is not zero.

Update protocol fee rate

The `updateProtocolFeeRate` function is used to update the protocol fee rate. The admin could call the function to update the protocol fee rate.

The `updateProtocolFeeRate` has modifiers.

- `onlyGovernor` modifier (the caller must have the GOVERNANCE_ROLE role)
- `whenInitialized` modifier (the Infrared contract must be initialized)

The `updateProtocolFeeRate` function checks that the new fee rate is not bigger than `FEE_UNIT`.

Claim protocol fee

The `claimProtocolFees` function is used to claim the protocol fee from the vault. The admin could call the function to claim the protocol fee from the vault.

The `claimProtocolFees` has modifiers.

- `onlyGovernor` modifier (the caller must have the GOVERNANCE_ROLE role)
- `whenInitialized` modifier (the Infrared contract must be initialized)

The `claimProtocolFees` function checks that the claim amount is not bigger than the accumulated protocol fee.

Recover tokens

The `recoverERC20` function is used to recover the token from the vault. The admin could call the function to recover the token from the vault.

The `recoverERC20` has modifiers.

- `onlyGovernor` modifier (the caller must have the GOVERNANCE_ROLE role)
- `whenInitialized` modifier (the Infrared contract must be initialized)

The `recoverERC20` function checks the following.

- The destination address is not zero.
- The token address is not zero.
- The amount is not zero.

InfraredVault

The owner is a trusted entity and has extensive control over the protocol. In this contract, except for the migrate function, only users set as Infrared can use the external functions. Since these are set and called by Infrared, they hold power and require special caution.

Migrate

This function is used to migrate the vault to a new version. The owner can set Berachain's reward-vault address and call the migrate function to transfer the total staked tokens to Berachain's new reward vault.

Update rewards duration

The `updateRewardsDuration` function is used to update the rewards duration. The Infrared role could set the rewards duration of reward token in the vault.

The `updateRewardsDuration` has a modifier.

- `onlyInfrared` modifier (the caller must have the Infrared role)

The `updateRewardsDuration` function checks the following.

- The new reward-token address is not zero.
- The new reward duration is not zero.

Add rewards

The `addReward` function is used to add reward tokens to the vault. The Infrared role could set the rewards and call the function to add rewards to the vault.

The `addReward` has a modifier.

- `onlyInfrared` modifier (the caller must have the Infrared role)

The `addReward` function checks the following.

- The reward-token address is not zero.
- The reward duration is not zero.

Notify rewards

The `notifyRewardAmount` function is used to notify the reward amount to the vault. The Infrared role could set the rewards and call the function to notify the reward amount to the vault.

The `notifyRewardAmount` has a modifier.

- `onlyInfrared` modifier (the caller must have the Infrared role)

The `notifyRewardAmount` function checks the following.

- The reward-token address is not zero.
- The reward amount is not zero.

Recover tokens

The `recoverERC20` function is used to recover the tokens from the vault. The Infrared role could call the function to recover the tokens from the vault.

This function calls `_recoverERC20` internally, which checks the following.

- The token address is not the stake token address.
- The token address is not one of the reward token addresses.

The `recoverERC20` has a modifier.

- `onlyInfrared` modifier (the caller must have the Infrared role)

The `recoverERC20` function checks the following.

- The token address is not zero.
- The amount is not zero.

MultiRewards

Users are unauthenticated. So any vulnerability exploited by users are more severe than those that can only be exploited by a user who has a permission role. Users have the ability to stake, withdraw, claim reward, and exit.

Stake

To stake in this protocol, the user has to call `stake`. This makes the user transfer staking tokens to the vault. The vault will then stake the tokens to Berachain's reward vault if it is set.

The `stake` function has modifiers.

- `nonReentrant` modifier (this modifier will prevent reentrancy attacks)
- `updateReward` modifier (this modifier will update the reward state for the user before the user's stake)
- `whenNotPaused` modifier (this modifier will prevent the user from staking when the vault is paused)

Withdraw

To withdraw from this protocol, the user has to call `withdraw`. This makes the user withdraw the staking token from the vault. The vault will then withdraw the token from Berachain's reward vault if it is set.

The `withdraw` function has modifiers.

- `nonReentrant` modifier (this modifier will prevent reentrancy attacks)
- `updateReward` modifier (this modifier will update the reward state for the user before the user's withdraw)
- `whenNotPaused` modifier (this modifier will prevent the user from withdrawing when the vault is paused)

Get rewards

To get rewards from this protocol, the user has to call `getReward` or `getRewardForUser`. This makes the user get the reward token from the vault. This function invokes the `onReward` function to call `harvestVault` to update the reward from Berachain's reward vault.

The `getRewardForUser` function has modifiers.

- `nonReentrant` modifier (this modifier will prevent reentrancy attacks)
- `updateReward` modifier (this modifier will update the reward state for the user before the user gets reward)

Exit

The `exit` function is a combination of `withdraw` and `getReward`. This makes the user withdraw the staking token and get the reward token from the vault.

5.2. Util functions

These util functions can be found at infrared-dao-infrared-contracts/contracts/src/utils.

ValidatorSet

This is a library for setting the validators. The library can get, add, remove, and replace the validators.

Get

This function returns a validator from the set.

Add

This function tries adding a validator to the set.

The `add` function checks that the set must not contain the given validator.

Remove

This function tries removing a validator from the set.

The `remove` function checks that the set must contain the given validator when this call is called.

Replace

This function tries replacing a validator of the set.

The `replace` function checks the following.

- The set must contain the given original validator when this call is called.
- The set must contain the given new validator when this call is called.

InfraredVaultDeployer

This is a library for deploying Infrared's contracts.

Deploy

This function deploys the InfraredVault contract and returns the address.

6. Assessment Results

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

During our assessment on the scoped Infrared contracts, we discovered one finding, which was of low impact. Infrared Finance acknowledged the finding and implemented a fix.

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.