Zellic

Prepared for
Luka Avbreht
Flare Network

Prepared by
Kritsada Dechawattana
Weipeng Lai
Zellic

April 15, 2026

FAsset Redeem Composer

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 Flare Network from April 7th to April 8th, 2026. During this engagement, Zellic reviewed FAsset Redeem Composer'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 question:

- Could a user receive both redemption and redemption default assets when redeeming f-assets?

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

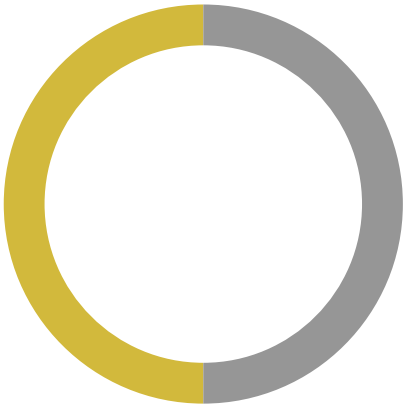
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped FAsset Redeem Composer contracts, we discovered two findings. No critical issues were found. One finding was of medium impact and the other finding was informational in nature.

Breakdown of Finding Impacts

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



2. Introduction

2.1. About FAsset Redeem Composer

Flare Network contributed the following description of FAsset Redeem Composer:

FAssetRedeemComposer and FAssetRedeemerAccount implement a cross-chain f-asset redemption flow via LayerZero. The FAssetRedeemComposer receives LayerZero compose callbacks when a user bridges f-assets from another chain, collects a configurable composer fee (PPM-based, with per-source-chain overrides), and then deploys a deterministic per-redeemer FAssetRedeemerAccount (BeaconProxy via `CREATE2`) to execute the actual redemption against the Flare AssetManager. If the redemption call fails, f-assets remain in the redeemer's account with pre-set max allowances, allowing the user to manually recover funds. The composer acts as the beacon for all redeemer account proxies, enabling implementation upgrades. All administrative operations (fee changes, implementation upgrades, token recovery) are protected by a configurable timelock via OwnableWithTimelock.

FAssetRedeemComposer (UUPS upgradeable, OwnableWithTimelock)

- Receives LayerZero compose callbacks (`lzCompose`) from a trusted endpoint/OApp when a user bridges f-assets from another chain
- Decodes `RedeemComposeMessage` containing: redeemer address, underlying destination, `redeemWithTag` flag, destination tag (for XRP), and optional custom executor
- Collects a configurable composer fee in f-assets (PPM-based, with per-source-chain overrides via `composerFeesPPM` mapping using fee+1 storage trick)
- Deploys a deterministic per-redeemer FAssetRedeemerAccount via `CREATE2` (beacon proxy pattern), or reuses an existing one
- Transfers remaining f-assets to the redeemer account and invokes redemption via `redeemFAsset`
- On redemption failure, tokens remain in the redeemer account with pre-set max allowances, allowing the user to manually recover funds
- Acts as the beacon (`IBeacon`) for all redeemer account proxies, enabling implementation upgrades via `setRedeemerAccountImplementation`
- Uses a default executor for redemption, overridable per-message via the compose payload
- Administrative operations (fee changes, executor/implementation updates, token recovery, UUPS upgrades) are protected by `OwnableWithTimelock`
- Owner can recover stuck f-assets (`transferFAsset`) from the composer in case compose flow reverts or is not invoked

FAssetRedeemComposerProxy (ERC1967 proxy)

- ERC1967 proxy that deploys and initializes FAssetRedeemComposer in its constructor
- Forwards all calls to the implementation via `delegatecall`; upgradeable through UUPS pattern on the implementation

FAssetRedeemerAccount (beacon proxy, one per redeemer)

- Executes f-asset redemption against the Flare `AssetManager`, routing to either `redeemAmount` or `redeemWithTag` (XRP destination tag support) based on the `redeemWithTag` flag
- Sets max token allowances (fAsset, stableCoin, wNat) to the redeemer/owner for manual fund recovery in case of redemption failure
- All functions restricted to the composer via `onlyComposer` modifier
- Uses a sentinel address (`0x...1111`) in the constructor to prevent direct use of the implementation contract

FAssetRedeemerAccountProxy (beacon proxy)

- BeaconProxy that uses the composer as its beacon, calling `IBeacon(composer).implementation()` for the current implementation address
- Initializes the FAssetRedeemerAccount with composer and owner addresses on deployment
- Deployed deterministically via `CREATE2` by the composer

OwnableWithTimelock (dependency)

- Extends OpenZeppelin OwnableUpgradeable with a configurable timelock mechanism (0–7 days) for sensitive owner operations
- Functions guarded by `onlyOwnerWithTimelock` are either executed immediately (when timelock is zero) or queued and executed after the timelock period expires
- Queued calls are identified by their calldata hash and can be cancelled by the owner before execution
- Used to protect all administrative operations on the composer (fee updates, executor changes, implementation upgrades, token recovery)

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.

2.3. Scope

The engagement involved a review of the following targets:

FAsset Redeem Composer Contracts

Type	Solidity
Platform	EVM-compatible
Target	Only changes between 7fa8cd6...4152050
Repository	https://github.com/flare-foundation/flare-smart-accounts ↗
Version	`415205069aeb25f9c722660d66878681871b08b6`

Programs	`composer/implementation/FAssetRedeemComposer.sol`
	`composer/implementation/FAssetRedeemerAccount.sol`
	`composer/proxy/FAssetRedeemComposerProxy.sol`
	`composer/proxy/FAssetRedeemerAccountProxy.sol`
	`utils/implementation/OwnableWithTimelock.sol`

2.4. Project Overview

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

Contact Information

The following project manager was associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Kritsada Dechawattana
Engineer
kritsada@zellic.io ↗

Weipeng Lai
Engineer
weipeng.lai@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

April 7, 2026	Start of primary review period
April 8, 2026	End of primary review period

3. Detailed Findings

3.1. The `lzCompose` function allows underfunded compose execution

Target	FAssetRedeemComposer		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

In LayerZero V2, EndpointV2 inherits from MessagingComposer, which exposes a fully permissionless `lzCompose` function ↗. Once a compose message has been queued via `sendCompose`, anyone can call `lzCompose` to deliver it to the target composer, and the caller fully controls `msg.value`.

In `FAssetRedeemComposer::lzCompose`, the contract forwards the received `msg.value` directly to `redeemerAccount.redeemFAsset`, which then forwards the same value to `assetManager.redeemAmount` or `assetManager.redeemWithTag` as the redemption executor fee.

```
function lzCompose(
    address _from,
    bytes32 _guid,
    bytes calldata _message,
    address /* _executor */,
    bytes calldata /* _extraData */
)
    external payable
    nonReentrant
{
    // [...]

    try IIFAssetRedeemerAccount(redeemerAccount).redeemFAsset{value: msg.
        value}(
        assetManager,
        amountToRedeemAfterFee,
        redeemComposeMessage.redeemerUnderlyingAddress,
        redeemComposeMessage.redeemWithTag,
        redeemComposeMessage.destinationTag,
        executor
    )
        returns (uint256 _redeemedAmountUBA)
    {
        // [...]
```

```
    } catch {
        // [...]
    }
}
```

```
function redeemFAsset(
    IAssetManager _assetManager,
    uint256 _amountLD,
    string calldata _redeemerUnderlyingAddress,
    bool _redeemWithTag,
    uint256 _destinationTag,
    address payable _executor
)
    external payable
    onlyComposer
    returns (uint256 _redeemedAmountUBA)
{
    require(!_redeemWithTag || _assetManager.redeemWithTagSupported(),
    RedeemWithTagNotSupported());
    if (_redeemWithTag) {
        _redeemedAmountUBA = _assetManager.redeemWithTag{value: msg.value}
            (_amountLD, _redeemerUnderlyingAddress, _executor,
    _destinationTag);
    } else {
        _redeemedAmountUBA = _assetManager.redeemAmount{value: msg.value}
            (_amountLD, _redeemerUnderlyingAddress, _executor);
    }
    // [...]
}
```

However, the composer never verifies that the delivered `msg.value` meets any minimum threshold or matches the amount the sender paid for on the source chain.

So an attacker could exploit it as follows:

1. A user initiates a source-chain redemption with a nonzero LayerZero compose value intended to become the redemption executor fee.
2. The OFT receives flow credits tokens and queues the compose payload in the endpoint.
3. An attacker observes the queued compose parameters from the `ComposeSent` event.
4. Before the intended LayerZero executor delivers the compose message, the attacker calls `endpoint.lzCompose` with the same queued payload but sets `msg.value = 0`.
5. The endpoint accepts the call, marks the compose as delivered, and invokes `FAssetRedeemComposer::lzCompose`.

6. The composer creates the redemption using zero native value, resulting in a redemption request with a zero executor fee.

Impact

An attacker can front-run the intended LayerZero executor and cause the redemption to be created with a smaller executor fee than the user paid for on the source chain. In the worst case, the fee can be reduced to zero.

This can cause the user to lose the compose value they paid as part of the LayerZero messaging fee, while also leaving the resulting redemption request without sufficient incentive for third parties to execute redemption default if required.

Recommendations

Encode the expected executor fee into the `RedeemComposeMessage` struct and validate it in `lzCompose`. If the delivered `msg.value` is lower than the expected amount, the call should revert so the compose message remains queued for delivery by the legitimate LayerZero executor with the correct compose value.

Remediation

This issue has been acknowledged by Flare Network, and fixes were implemented in the following commits:

- `2f9a9583` ↗
- `84781421` ↗

3.2. Owner of FAssetRedeemerAccount cannot directly exercise payment-default rights

Target	FAssetRedeemerAccount		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In FAssetRedeemComposer, the composer transfers f-assets into a per-user FAssetRedeemerAccount and then calls `redeemFAsset` on that account. Because the subsequent `redeemAmount` or `redeemWithTag` call to the `AssetManager` is made by the redeemer account contract itself, the `AssetManager` records the FAssetRedeemerAccount contract, rather than the end user, as the redeemer of record.

```
function redeemFAsset(
    IAssetManager _assetManager,
    uint256 _amountLD,
    string calldata _redeemerUnderlyingAddress,
    bool _redeemWithTag,
    uint256 _destinationTag,
    address payable _executor
)
    external payable
    onlyComposer
    returns (uint256 _redeemedAmountUBA)
{
    require(!_redeemWithTag || _assetManager.redeemWithTagSupported(),
    RedeemWithTagNotSupported());
    if (_redeemWithTag) {
        _redeemedAmountUBA = _assetManager.redeemWithTag{value: msg.value}
            (_amountLD, _redeemerUnderlyingAddress, _executor,
    _destinationTag);
    } else {
        _redeemedAmountUBA = _assetManager.redeemAmount{value: msg.value}
            (_amountLD, _redeemerUnderlyingAddress, _executor);
    }
    emit FAssetRedeemed(
        _amountLD,
        _redeemerUnderlyingAddress,
        _redeemWithTag,
```

```
            _destinationTag,
            _executor,
            msg.value,
            _redeemedAmountUBA
        );
    }
```

If the agent fails to settle the underlying payment, the redeemer can call `redemptionPaymentDefault` or `xrpRedemptionPaymentDefault` on the `AssetManager` to trigger default processing. However, FAssetRedeemerAccount does not expose any owner-authorized pass-through for either function. As a result, the end user, despite owning the FAssetRedeemerAccount, cannot directly exercise these payment-default rights.

Instead, the user must rely on the designated executor to trigger the default. If the executor does not act, the user must wait until the `confirmationByOthersAfterSeconds` window has elapsed before a third party can do so permissionlessly.

Impact

This does not fully break the redemption-default flow: the designated executor can still trigger the default, and after `confirmationByOthersAfterSeconds` elapses, any third party may do so as well. However, it removes a fallback capability that users would reasonably expect to retain.

When the selected executor is not controlled by the user, the owner must trust that executor to act correctly and promptly. If the executor fails to do so, the owner cannot intervene directly and must wait for the permissionless window to open.

Recommendations

Add owner-authorized pass-through functions to FAssetRedeemerAccount for `redemptionPaymentDefault` and `xrpRedemptionPaymentDefault`, so the account owner can directly exercise these rights when necessary.

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit c4b35695 ↗.

4. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

4.1. FAssetRedeemComposer

Description

The FAssetRedeemComposer is a LayerZero compose handler that orchestrates cross-chain f-asset redemption by managing deterministic per-redeemer accounts and coordinating with the Flare asset manager. It executes redemption requests to convert f-assets back to underlying assets on the source chain. Shown below is the core functionality of the FAssetRedeemComposer contract:

1. **Cross-chain compose handling.** It implements `ILayerZeroComposer.lzCompose` to receive and process f-asset transfers with embedded redemption instructions from trusted source chains.
2. **Deterministic account management.** It creates and manages per-redeemer accounts using `CREATE2` with a beacon proxy pattern, ensuring each redeemer has a unique, predictable address.
3. **Redemption orchestration.** It delegates actual redemption to redeemer accounts, which call the asset manager's `redeemAmount` or `redeemWithTag` functions.

Invariants

The following invariants must hold true:

1. Only the configured LayerZero endpoint can invoke `lzCompose`, and only messages from the trusted source OApp are processed.
2. Each redeemer address maps to exactly one deterministic redeemer account address.
3. Composer fees must always be strictly less than 1,000,000 PPM (100%), ensuring some amount always reaches the redeemer account.
4. The beacon implementation address for redeemer accounts must always point to a contract with nonzero code.
5. All privileged owner functions (fee updates, implementation changes, upgrades) are protected by the timelock mechanism.
6. If redemption fails, any native tokens sent with the compose message are wrapped as wNat and deposited to the redeemer account.

Attack surface

The following outlines the attack surface:

1. **Unauthorized compose invocation.** This is prevented via the `require(msg.sender == endpoint)` check and `require(_from == trustedSourceOApp)` validation in `lzCompose`, ensuring only the designated LayerZero endpoint can call the function and only messages from the trusted source OApp are processed. However, the `msg.value` is not validated, allowing an attacker to force the redemption request to be created with zero executor fee (see Finding 3.1. ↗).
2. **Reentrancy attacks during compose.** This is prevented via the `nonReentrant` modifier on `lzCompose`, blocking any reentrant calls that could manipulate balances or state during redemption execution.
3. **Front-running redeemer account creation.** This is prevented via deterministic `CREATE2` deployment with zero salt, ensuring the redeemer account address is predictable and only the composer can deploy it.
4. **Stuck funds from a failed compose.** This is prevented via the `transferFAsset` recovery function with `onlyOwnerWithTimelock` protection, allowing the owner to recover f-assets that fail to process through the compose flow.
5. **Emergency response.** Governance can cancel pending calls to respond to threats.

4.2. FAssetRedeemerAccount

Description

The FAssetRedeemerAccount is a deterministic per-redeemer proxy contract deployed via beacon pattern that serves as an isolated execution context for cross-chain f-asset redemptions orchestrated by the FAssetRedeemComposer. Each redeemer gets a unique account instance that holds f-asset tokens received from LayerZero OFT transfers and executes redemption calls to the Flare asset manager on behalf of the composer. Shown below is the core functionality of the FAssetRedeemerAccount contract:

1. **F-asset redemption execution.** It calls the asset manager's `redeemAmount` or `redeemWithTag` methods with native token fees, supporting both standard redemptions and XRP destination tag–based redemptions.
2. **Token allowance management.** It sets maximum (`type(uint256).max`) allowances from the account to the owner for f-asset, stablecoin, and wrapped native tokens, enabling the owner to withdraw any tokens held by the account.

Invariants

The following invariants must hold true:

1. The composer address is immutable after initialization and is the only entity authorized to call `redeemFAsset` and `setMaxAllowances` via the `onlyComposer` modifier.
2. The implementation contract's composer is permanently set to the sentinel address `0x0000000000000000000000000000000000001111` in the constructor, preventing direct initialization or use of the implementation.
3. Each account can only be initialized once, enforced by checking that `composer == address(0)` before initialization.
4. When `_redeemWithTag` is true, the asset manager must support `redeemWithTagSupported()`; otherwise, the transaction reverts with `RedeemWithTagNotSupported`.
5. All native tokens sent with `redeemFAsset` are forwarded to the asset manager as executor fees.

Attack surface

The following outlines the attack surface:

1. **Unauthorized redemption execution.** This is prevented via the `onlyComposer` modifier on `redeemFAsset`, which restricts redemption calls to the trusted composer contract only.
2. **Unauthorized allowance manipulation.** This is prevented via the `onlyComposer` modifier on `setMaxAllowances`, ensuring only the composer can grant token allowances to the owner.
3. **Double initialization.** This is prevented via the `require(composer == address(0), AlreadyInitialized())` check in initialize, ensuring the account can only be initialized once.
4. **Unsupported redemption method invocation.** This is prevented via the `require(!_redeemWithTag || _assetManager.redeemWithTagSupported(), RedeemWithTagNotSupported())` check before calling `redeemWithTag`, ensuring the asset manager supports tag-based redemptions when requested.
5. **Token approval front-running or manipulation.** This is prevented by using `SafeERC20.forceApprove`, which handles both standard and nonstandard ERC-20 implementations safely, resetting allowances to zero before setting the new value if necessary.

4.3. OwnableWithTimelock

Description

OwnableWithTimelock extends OpenZeppelin's OwnableUpgradeable to add a two-step timelock mechanism for sensitive owner operations, requiring owner calls to be queued with a configurable

delay before execution. Shown below is the core functionality of the OwnableWithTimelock contract:

1. **Call queueing.** When an owner invokes a function with the `onlyOwnerWithTimelock` modifier, the calldata is hashed and stored with a future execution timestamp (`block.timestamp + timelockDurationSeconds`).
2. **Delayed execution.** After the timelock period expires, anyone can execute the queued call via `executeTimelockedCall`, which performs a call to the contract itself with the original calldata.
3. **Cancellation.** The owner can cancel any queued call before execution using `cancelTimelockedCall`.
4. **Timelock configuration.** The timelock duration can be adjusted via `setTimelockDuration` (protected by `onlyOwnerWithTimelock`).
5. **Bypass mechanism.** When `timelockDurationSeconds` is set to zero, the timelock is disabled and calls `execute` immediately.

Invariants

The following invariants must hold true:

1. The `executing` flag must be false before and after any external call, ensuring no nested execution of timelocked calls.
2. The timelock duration cannot exceed `MAX_TIMELOCK_DURATION_SECONDS` (seven days).
3. A queued call can only be executed after `block.timestamp >= allowedAfterTimestamp`.
4. Only the owner can queue calls (via `_checkOwner()` in `_recordTimelockedCall`).
5. When `timelockDurationSeconds` is zero, `onlyOwnerWithTimelock` functions execute immediately without queueing.
6. A call hash can only be executed once.
7. After execution via `executeTimelockedCall`, the mapping entry is deleted.

Attack surface

The following outlines the attack surface:

1. **Reentrancy during execution.** This is prevented via the executing flag, which is set to true before the low-level call in `executeTimelockedCall` and reset to false afterward, blocking reentrant calls through the `_beforeExecuteTimelockedCall` check.

2. **Unauthorized call execution.** This is prevented via the `_checkOwner()` call in `_recordTimelockedCall`, ensuring only the owner can queue calls, and the timestamp check in `executeTimelockedCall` ensuring calls execute only after the delay.
3. **Front-running call execution.** This is mitigated by allowing anyone to execute a queued call after the timelock expires, ensuring the owner cannot be griefed by delaying execution, though the owner retains the ability to cancel via `cancelTimelockedCall`.
4. **Timelock bypass via zero duration.** This feature allows the owner to disable the timelock by setting `timelockDurationSeconds` to zero. However, changing the duration itself must go through the timelock mechanism, since `setTimelockDuration` uses `onlyOwnerWithTimelock`.
5. **Excessive timelock duration.** This is prevented via the `MAX_TIMELOCK_DURATION_SECONDS` check in `setTimelockDuration`, capping the maximum delay at seven days.

5. Assessment Results

During our assessment on the scoped FAsset Redeem Composer contracts, we discovered two findings. No critical issues were found. One finding was of medium impact and the other finding was informational in nature.

5.1. 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.