Zellic

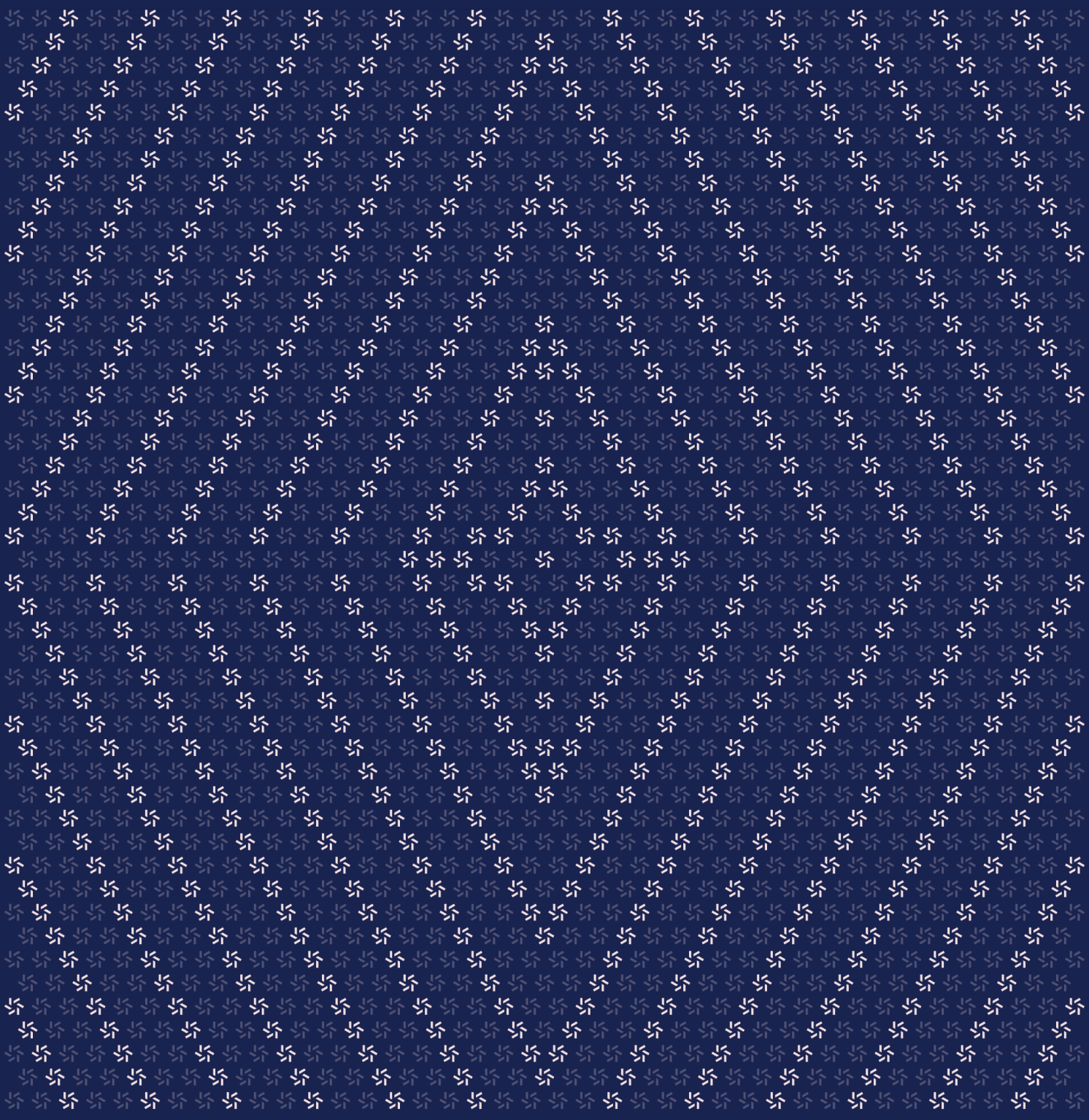
Prepared for
Royco Protocol

Prepared by
Katerina Belotskaia
Vlad Toie
Zellic

February 6, 2025

Royco CCDM

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 Royco Protocol from January 20th to February 3rd, 2025. During this engagement, Zellic reviewed Royco CCDM's code for security vulnerabilities, design issues, and general weaknesses in security posture.

It is important to note that Zellic was only asked to assess the functionality related to the Merkle flow, as the Royco Protocol team stated that the non-Merkle flow will not be used in the deployed version of the code.

We recommend assessing or explicitly removing the functionality not related to Merkle flow from the scoped contracts. This will help to reduce the attack surface and minimize the complexity of the codebase.

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:

- How does Royco implement the cross-chain bridging mechanism?
- Are there any possible edge cases when depositing/withdrawing assets in the DepositLocker?
- Could anyone leverage the LP tokens' operations? If so, what are the implications?
- Are there any actions that might lead to a lock of funds in the DepositLocker? What about in the DepositExecutor?

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
- Functionality from the scoped contracts not related to the Merkle flow

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide. During this assessment, the limitation of scoping prevented us from conducting a full review of the entire codebase, which would have included all possible attack vectors and viewpoints.

1.4. Results

During our assessment on the scoped Royco CCDM contracts, we discovered one finding, which was informational in nature.

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

Breakdown of Finding Impacts

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

2. Introduction

2.1. About Royco CCDM

Royco Protocol contributed the following description of Royco CCDM:

> Royco Protocol allows anyone to create a market for any onchain action — we call these markets Incentivized Action Markets (IAMs).

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:

Royco CCDM Contracts

Type	Solidity
Platform	EVM-compatible
Target	Merkle Bridging functionality only
Repository	https://github.com/roycoprotocol/cross-chain-deposit-module ↗
Version	`2a77bb17dbe62cf40eee8bae02fa9517408fddbc`
Programs	`DepositLocker` `DepositExecutor`

2.4. Project Overview

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

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:

Katerina Belotskaia
Engineer
kate@zellic.io ↗

Vlad Toie
Engineer
vlad@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

January 20, 2025	Kick-off call
January 20, 2025	Start of primary review period
February 3, 2025	End of primary review period

3. Detailed Findings

3.1. Funds can be permanently stuck upon multiple bridging operations

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

Description

The `depositorToWeirollWalletToWeirollWalletDepositInfo[depositor][msg.sender]` in DepositLocker accumulates during `deposit` calls; it is not reset after bridging operations.

Theoretically, the following order of operations is possible:

1. A user calls `deposit()` with the same `depositor`, whichever other params; `depositorToWeirollWalletToWeirollWalletDepositInfo[depositor][msg.sender]` will keep on increasing for that `[depositor][msg.sender]` pair.
2. Bridging will occur just as expected, and `depositorToWeirollWalletToWeirollWalletDepositInfo[depositor][msg.sender]` will not be reset.
3. Steps 1 and 2 happen many times, just as intended.
4. In the event that bridging has to be halted, users are allowed to withdraw; `depositorToWeirollWalletToWeirollWalletDepositInfo[depositor][msg.sender]` will be huge by this point, as the deposits continued accumulating throughout all the deposit epochs.

This would lead to a situation where the current bridging operation's information, stored in `marketHashToDepositorToIndividualDepositorInfo`, will be significantly less than the accrued deposits in `depositorToWeirollWalletToWeirollWalletDepositInfo[depositor][msg.sender]`, which would then lead to an underflow when calculating the amount to withdraw.

```
function withdraw() external nonReentrant {

    // ...

    IndividualDepositorInfo storage depositorInfo =
    marketHashToDepositorToIndividualDepositorInfo[targetMarketHash][depositor];
```

```
    WeirollWalletDepositInfo storage walletInfo =
    depositorToWeirollWalletToWeirollWalletDepositInfo[depositor][msg.sender];

    uint256 amountToWithdraw = walletInfo.amountDeposited;

    // THIS would underflow if `amountToWithdraw` is greater than
    `depositorInfo.totalAmountDeposited`
    depositorInfo.totalAmountDeposited -= amountToWithdraw;
```

Impact

This issue could lead to a total loss of funds for the user, as the amount to withdraw would be calculated incorrectly, leading to a permanent underflow.

Recommendations

We recommend adequately resetting the `depositorToWeirollWalletToWeirollWalletDepositInfo[depositor][msg.sender]` after each bridging operation. Alternatively, we recommend mapping the depositor wallet information to only the current bridging operation, not the entire deposit history.

Remediation

This issue has been acknowledged by Royco Protocol. The team has stated that this particular flow and the functions that are involved in it will not be used in the deployed version of the code. We therefore note this finding as Informational.

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 issue while processing LP deposits for non-Merkle flow

In `_processLpTokenDepositor`, there is a possibility that in the `if (token0_DepositAmount == 0 || token1_DepositAmount == 0) {`, only one of the `token0`s / `token1`s are =0.

In this case, if we go back in the calling function (i.e., `bridgeLpTokens`), the LP tokens are still accounted for in `lp_DepositAmounts`, but the underlying tokens will be partly or totally returned to the user.

As this potential issue was not part of the Merkle flow, and the Royco Protocol team has stated that the non-Merkle flow will not be used in the deployed version of the code, we leave this note for posterity.

4.2. Potential economic implications of moving LP positions cross-chain

The bridging mechanism for LP tokens allows users to remove liquidity from a Uniswap pool on chain A and bridge the two underlying assets to chain B. There are several potential economic implications of this particular bridging operation that users should be aware of. For example, if we are to assume that the Uniswap pool on chain A is more liquid than the Kodiak pool on chain B, the users might experience a loss in liquidity when moving their LP positions cross-chain. This is mainly because of the liquidity depths of the two pools, which will most likely differ.

Say, for example, that 1 ETHxUSDC was = 0.001 ETH / 5 USDC in Uniswap. Since the Kodiak pool will have lower liquidity of either ETH or USDC, the price will be 1 ETHxUSDC = 0.004 ETH / 5 USDC. We expect that due to liquidity differences in between the Uniswap pool and the Kodiak counterpart, the amount of LP tokens the users receive on the other side will probably vary, possibly putting them at economical disadvantage.

The Royco Protocol team agrees that this is a plausible scenario and that the users should be aware of the potential implications of moving LP positions cross-chain.

Additionally, we note several other potential implications of moving LP positions cross-chain that users should be aware of:

- **Delayed execution.** Because the bridging occurs in batches, there is a window of time during which market prices can shift (in between a user removing their liquidity for a position and when bridging occurs). This delay, even if we are to assume that liquidity depths are similar, means that by the time the deposit is executed on chain B, the prevailing price might be different from what was anticipated, further exposing users to

impermanent loss.

- **MEV and front-running potential.** Knowing that a large deposit is about to be bridged might allow sophisticated actors or bots to front-run these transactions on the destination chain. They could, for example, place orders that manipulate the pool's price right before the large deposit is executed, capturing value at the expense of the bridge participants. The team has stated that adequate slippage protection will be in place to mitigate this risk.

4.3. The accounting of `merkleDepositsInfo.totalAmountdeposited` needs to be updated

In `DepositLocker:merkleWithdraw`, the `merkleDepositsInfo.totalAmountdeposited` should be updated to reflect the decrease in the total amount deposited, based on the amount withdrawn by the user.

4.4. Usage of unsafe version of SafeTransferLib

The DepositLocker contract utilizes a vulnerable version of Solmate's SafeTransferLib. This version of the library does not verify the code size of the token address in the `safeTransferFrom` and `safeTransfer` functions. As a result, if the called token address does not have contract code, these functions will always return a success, even if the transfer has not been performed.

We recommend updating to the latest version of SafeTransferLib ↗, which includes proper code-size verification.

4.5. Custom Weiroll Wallet can be used

Weiroll Wallets, which are used in the `merkleDeposit` and `deposit` calls, are currently deployed by the RecipeMarketHub. During our assessment, it was an implied trust assumption of the DepositLocker that the Weiroll Wallets are deployed by a trusted entity, as the `onlyWeirollWallet` modifier checked whether the caller was deployed by a specific proxy EIP-1167 contract. This modifier is, however, bypassable, and as a result, anyone can deploy technically deploy a WeirollWallet manually, specifying custom parameters during the contract creation.

The Royco Protocol team has stated, however, that it is within the trust assumptions of the protocol that the Weiroll Wallets can be deployed manually by arbitrary users. We therefore leave this discussion point for posterity, should any integrators compose and implement the `onlyWeirollWallet` modifier in their own contracts.

4.6. Insufficient Test suite

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 all contracts, not just surface-level functions. It is important to test the invariants required for ensuring security. Additionally, testing cross-chain function calls and transfers is recommended to ensure the desired functionality.

Therefore, we recommend building a rigorous test suite that includes all contracts to ensure that the system operates securely and as intended.

Good test coverage has multiple effects.

- It finds bugs and design flaws early (preaudit or prerelease).
- 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.

Additionally, due to incomplete testing and challenges in integrating with the destination chain, we cannot confirm that the LayerZero integration performs as expected. This indicates that further refinements and extensive validation are necessary before relying on the cross-chain functionality. We recommend implementing further testing and validation to ensure the system operates as intended.

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: DepositExecutor.sol

Function: `executeDepositRecipes(bytes32 _sourceMarketHash, address[] calldata _weirollWallets)`

This function allows the owner of the campaign related to the provided `MarketHash` to execute the deposit scripts. The caller provided the list of the `_weirollWallets` addresses that will be executed. Tokens for the provided `_weirollWallets` should have been bridged, and `lzCompose` should have been executed at the moment of the execution of this function. If `campaign` expects two tokens, and one of them has not been bridged yet and `lzCompose` has not been executed, the related `_weirollWallet` cannot be used until both tokens have been delivered and processed.

Inputs

- `_sourceMarketHash`
  - **Control**: Full control, but the caller should be a valid owner of the campaign related to this `_sourceMarketHash`.
  - **Constraints**: The campaign related to this `_sourceMarketHash` should be verified by the `campaignVerifier` account, and the `numInputTokens` should not be zero for this campaign, and `campaign.inputTokens.length` should be equal to the `campaign.numInputTokens`.
  - **Impact**: The deposit recipes will be executed on the Weiroll Wallet contracts related to the campaign associated with the provided `_sourceMarketHash`.
- `_weirollWallets`
  - **Control**: Full control.
  - **Constraints**: Should not contain the already executed Weiroll Wallet contracts.
  - **Impact**: The addresses of the Weiroll Wallet contracts that will be executed.

Branches and code coverage

Intended branches

- The `sourceMarketHashToFirstDepositRecipeExecuted` set to `true` after the first execution.

- ☐ Test coverage

Negative behavior

- `campaign` is not verified.
  - ☐ Negative test
- `campaign.numInputTokens` is zero.
  - ☐ Negative test
- `campaign.inputTokens.length` != `campaign.numInputTokens`.
  - ☐ Negative test
- One of the `inputTokens` has not been delivered for the provided `weirollWallets`.
  - ☐ Negative test
- Allowance is not enough.
  - ☐ Negative test
- The `receiptToken` balance after execution is zero.
  - ☐ Negative test

Function call analysis

- `weirollWallet.marketHash()`
  - **What is controllable?** `weirollWallet`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the related `marketHash`, provided during the wallet creation; the `marketHash` of this wallet should be match the provided `_sourceMarketHash`.
  - **What happens if it reverts, reenters or does other unusual control flow?** No problem — pure function.
- `weirollWallet.executed()`
  - **What is controllable?** `weirollWallet`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the `bool executed` global variable — if this wallet has been already executed, returns `true`. The verification of this variable allows preventing the wallet from being executed more than once and, accordingly, the resending of tokens.
  - **What happens if it reverts, reenters or does other unusual control flow?** No problem.
- `_transferInputTokensToWeirollWallet(campaign.inputTokens, walletAccounting, _weirollWallets[i]) -> inputToken.safeTransfer(_weirollWallet, amountOfTokenDepositedIntoWallet)`
  - **What is controllable?** `weirollWallet`.

    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** The `executeDepositRecipes` function has a `nonReentrant` modifier to prevent reentrancy.
- `receiptToken.balanceOf(_weirollWallets[i])`
    - **What is controllable?** `_weirollWallets[i]`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the current `receiptToken` balance for the `_weirollWallets` account before the execution.
    - **What happens if it reverts, reenters or does other unusual control flow?** No problem.
- `weirollWallet.executeWeiroll(depositRecipe.weirollCommands, depositRecipe.weirollState);`
    - **What is controllable?** `weirollWallet`.
    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** The `executeDepositRecipes` function has a `nonReentrant` modifier to prevent reentrancy.
- `receiptToken.balanceOf(_weirollWallets[i])`
    - **What is controllable?** `_weirollWallets[i]`
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the current `receiptToken` balance for the `_weirollWallets` account after the execution, to verify that the balance changed.
    - **What happens if it reverts, reenters or does other unusual control flow?** No problem.
- `receiptToken.allowance(_weirollWallets[i], address(this))`
    - **What is controllable?** `_weirollWallets[i]`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the current `allowance` from the `_weirollWallet` to the current contract — it should be more than `type(uint96).max`.
    - **What happens if it reverts, reenters or does other unusual control flow?** No problem.
- `campaign.inputTokens[j].allowance(_weirollWallets[i], address(this))`
    - **What is controllable?** `_weirollWallets[i]`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the current `allowance` from the `_weirollWallet` to the current contract — it should be more than `type(uint96).max`.
    - **What happens if it reverts, reenters or does other unusual control flow?** No problem.

Function: `initializeCampaign(bytes32 _sourceMarketHash, uint256 _unlockTimestamp, ERC20 _receiptToken, Recipe calldata _depositRecipe)`

This function allows the campaign owner to initialize the campaign data.

Inputs

- `_sourceMarketHash`
  - **Control**: Full control.
  - **Constraints**: Should not be already initialized.
  - **Impact**: The hash of the market related to the campaign that will be initialized.
- `_unlockTimestamp`
  - **Control**: Full control.
  - **Constraints**: Cannot be set up if the deposit has been received already for this campaign. It cannot exceed the `MAX_CAMPAIGN_LOCKUP_TIME`.
  - **Impact**: Deposit can only be withdrawn after the `_unlockTimestamp` has passed.
- `_receiptToken`
  - **Control**: Full control.
  - **Constraints**: Cannot be zero address.
  - **Impact**: The receipt token, which should be received as a result of the Weiroll Wallet execution.
- `_depositRecipe`
  - **Control**: Full control.
  - **Constraints**: There are not any constraints.
  - **Impact**: The Recipe contains `weirollCommands` and `weirollState` arrays, which are used to the execute the deposit recipe.

Branches and code coverage

Intended branches

- The `campaign` has been initialized and `campaign.verified` has been reset.
  - ☑ Test coverage

Negative behavior

- The caller is not a valid `CampaignOwner`.
  - ☐ Negative test
- The current `campaign.receiptToken` is not zero.

- ☐ Negative test
- The new `_receiptToken` is zero.
  - ☐ Negative test
- `campaign.numInputTokens != 0` and `_unlockTimestamp != 0`.
  - ☐ Negative test

Function: `lzCompose(address _from, bytes32 _guid, bytes calldata _message, address, bytes calldata)`

This function allows processing the compose message transferred from the DepositLocker contract of the source chain during token-bridging process. The compose message contains the following information: `marketHash`, `ccdmNonce`, `numTokensBridged`, `srcChainTokenDecimals`, and `bridgeType`.

For the case of the LP token, bridging the `numTokensBridged` is equal to 2. Otherwise, `numTokensBridged` is 1. The `bridgeType` can be `MERKLE_DEPOSITORS` or `INDIVIDUAL_DEPOSITORS`. The current function observation only covers the `MERKLE_DEPOSITORS` type processing. Current function is allowed to be called only by the trusted `LAYER_ZERO_V2_ENDPOINT` contract; this address is immutable.

Inputs

- `_from`
  - **Control**: This address is not controlled by the caller (the `LAYER_ZERO_V2_ENDPOINT` contract), because this address defines the OFT contract, which has been initiated the `sendCompose` function execution to save the compose-message hash for subsequent usage by the `lzCompose` function. So only the address of the contract who executed `sendCompose` can be passed to this function. Also, the executor of `sendCompose` determines the receiver of the compose message, the message guid, and the message as well. But any caller can execute the `sendCompose` function to save inside the `LAYER_ZERO_V2_ENDPOINT` contract the compose message for this contract.
  - **Constraints**: This address should be added as trusted OFT to the `isValidLzV2OFT` by the owner of the contract, so during this function execution, this address is validated.
  - **Impact**: This contract determines the address of the `depositToken` contract related to this OFT.
- `_guid`
  - **Control**: This parameter is not controlled by the caller but controlled by the OFT and saved during `sendCompose` execution.
  - **Constraints**: There are not any constraints.
  - **Impact**: The `_guid` is used for the `CCDMBridgeProcessed` event.

- `_message`
    - **Control**: This parameter is not controlled by the caller but controlled by the OFT and saved during `sendCompose` execution.
    - **Constraints**: There are not any constraints. But the `_message.composeFrom` address should be equal to the expected `DEPOSIT_LOCKER` address, and `_message.srcEid` should be equal to the `SOURCE_CHAIN_LZ_EID`, so the message cannot be bridged from the other chain.
    - **Impact**: Contains the `nonce`, `srcEid`, `amountReceivedLD`, `composeFrom`, and `composeMsg`.

Branches and code coverage

Intended branches

- The Weiroll Wallet has been successfully created with the expected state.
    - ☑ Test coverage
- The `walletAccounting.merkleRoot` has been initialized with the expected `merkleRoot`.
    - ☑ Test coverage
- The `walletAccounting.totalMerkleTreeSourceAmountLeftToWithdraw` equals the expected amount.
    - ☑ Test coverage
- The `walletAccounting.tokenToTotalAmountDepositedOnDest` for the `depositToken` equals the expected amount in the case a single token is bridged.
    - ☐ Test coverage
- The `walletAccounting.tokenToTotalAmountDepositedOnDest` for `depositToken1` and `depositToken2` is equal to the expected amounts in the case LP token is bridged.
    - ☐ Test coverage
- The `CCDMBridgeProcessed` event has been emitted.
    - ☐ Test coverage
- The expected amount of native tokens has been deposited to the `WRAPPED_NATIVE_ASSET_TOKEN` contract in the case the `from` OFT contract supports native tokens.
    - ☐ Test coverage

Negative behavior

- The caller is not `LAYER_ZERO_V2_ENDPOINT`.
    - ☐ Negative test
- The `from` address is not a valid OFT.

- ☐ Negative test
- The `srcEid` is not a `SOURCE_CHAIN_LZ_EID`.
  - ☐ Negative test
- The `composeFrom` address is not `DEPOSIT_LOCKER`.
  - ☐ Negative test

Function call analysis

- `ERC20(IOFT(_from).token())`
  - **What is controllable?** `_from`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the address of the deposit token — if the same `token` address has been returned for both underlying tokens for the bridged LP token, then the `walletAccounting.tokenToTotalAmountDepositedOnDest[depositToken]` will be rewritten.
  - **What happens if it reverts, reenters or does other unusual control flow?** The `isValidLzV2OFT` contains only the addresses of the OFT tokens that support the `token()` function. In the case the returned address is zero, it means that this OFT supports native tokens.

Function: `setCampaignDepositRecipe(bytes32 _sourceMarketHash, Recipe calldata _depositRecipe)`

This function allows the campaign owner to update the Recipe data. This function can be executed even if the related campaign has been verified already, but the status of verification will be reset after that.

Inputs

- `_sourceMarketHash`
  - **Control**: Full control.
  - **Constraints**: The campaign related to the provided `_sourceMarketHash` should be initialized.
  - **Impact**: The hash of the market related to the campaign that will be updated.
- `_depositRecipe`
  - **Control**: Full control.
  - **Constraints**: There are not any constraints.
  - **Impact**: The Recipe contains `weirollCommands` and `weirollState` arrays, which are used to the execute the deposit recipe.

Branches and code coverage

Intended branches

- The `campaign.depositRecipe` has been updated, and `campaign.verified` has been reset.
  - ☐ Test coverage

Negative behavior

- The caller is not a valid `CampaignOwner`.
  - ☐ Negative test
- `campaign.receiptToken` is zero.
  - ☐ Negative test

Function: `setCampaignReceiptToken(bytes32 _sourceMarketHash, ERC20 _receiptToken)`

This function allows the campaign owner to update the `receiptToken` address. This function can be executed even if the related campaign has been verified already, but the status of verification will be reset after that. But if the deposit recipe for this market hash has been executed, the `receiptToken` cannot be updated.

Inputs

- `_sourceMarketHash`
  - **Control**: Full control.
  - **Constraints**: The campaign related to the provided `_sourceMarketHash` should be initialized.
  - **Impact**: The hash of the market related to the campaign that will be updated.
- `_receiptToken`
  - **Control**: Full control.
  - **Constraints**: Cannot be zero address.
  - **Impact**: The new receipt token, which should be received as a result of the Weiroll Wallet execution.

Branches and code coverage

Intended branches

- The `campaign.receiptToken` has been updated, and `campaign.verified` has been reset.

- ☐ Test coverage

Negative behavior

- The caller is not a valid `CampaignOwner`.
  - ☐ Negative test
- `campaign.receiptToken` is zero.
  - ☐ Negative test
- The deposit recipe for this `_sourceMarketHash` has been executed already.
  - ☐ Negative test
- The new `_receiptToken` is zero.
  - ☐ Negative test

Function: `setNewCampaignOwner(bytes32 _sourceMarketHash, address _campaignOwner)`

This function allows the campaign owner of the contract owner to set a new campaign owner.

Inputs

- `_sourceMarketHash`
  - **Control**: Full control.
  - **Constraints**: There are not any constraints.
  - **Impact**: The owner campaign related to this `_sourceMarketHash` will be updated.
- `_campaignOwner`
  - **Control**: Full control.
  - **Constraints**: There are not any constraints.
  - **Impact**: This account is granted access to perform `initializeCampaign`, `setCampaignReceiptToken`, `setCampaignDepositRecipe`, and `executeDepositRecipes` functions.

Branches and code coverage

Intended branches

- The campaign-owner address has been updated.
  - ☐ Test coverage

Negative behavior

- The caller is not a current campaign owner or contract owner.
    - ☐ Negative test

Function: `unverifyCampaign(bytes32 _sourceMarketHash)`

This function allows the campaign verifier to reset the campaign's `verified` status.

Inputs

- `_sourceMarketHash`
    - **Control**: Full control.
    - **Constraints**: There are not any constraints.
    - **Impact**: If the campaign is not verified, the execution cannot be performed.

Branches and code coverage

Intended branches

- The `verified` status has been reset.
    - ☐ Test coverage

Negative behavior

- The caller is not a valid campaign verifier.
    - ☐ Negative test

Function: `verifyCampaign(bytes32 _sourceMarketHash, bytes32 _campaignVerificationHash)`

This allows the campaign verifier to verify the campaign.

Inputs

- `_sourceMarketHash`
    - **Control**: Full control.
    - **Constraints**: There are not any constraints.
    - **Impact**: The campaign related to this `_sourceMarketHash` will be verified.
- `_campaignVerificationHash`
    - **Control**: Full control.

- **Constraints**: Should be equal to `keccak256(abi.encode(campaign.receiptToken, campaign.depositRecipe))`.
- **Impact**: Allows to check that the caller performs a validation of the expected Recipe.

Branches and code coverage

Intended branches

- The `verified` status has been set.
  - ☐ Test coverage

Negative behavior

- The caller is not a valid campaign verifier.
  - ☐ Negative test
- `campaign.receiptToken` is zero.
  - ☐ Negative test
- The `_campaignVerificationHash` does not match `keccak256(abi.encode(campaign.receiptToken, campaign.depositRecipe))`.
  - ☐ Negative test

Function: `withdrawMerkleDeposit(address _weirollWallet, uint256 _merkleDepositNonce, uint256 _amountDepositedOnSource, bytes32[] calldata _merkleProof)`

This allows withdrawing tokens from the Weiroll Wallet to the depositor address. The `walletAccounting.leafToWithdrawn[depositLeaf]` should not be set to `true`; otherwise, it means that this deposit has been withdrawn already and the function execution will be reverted.

Inputs

- `_weirollWallet`
  - **Control**: Full control.
  - **Constraints**: Only the Weiroll Wallet, which has been created as a result of the `lzCompose` function execution, can be used.
  - **Impact**: From this, `_weirollWallet` tokens will be transferred to the caller.
- `_merkleDepositNonce`
  - **Control**: Full control.

    - **Constraints**: The leaf generated using this `_merkleDepositNonce` should be a part of the Merkle tree.
    - **Impact**: The unique ID of the Merkle deposit — it is used to generate the leaf hash for the caller to validate the provided proof.
- `_amountDepositedOnSource`
    - **Control**: Full control.
    - **Constraints**: The leaf generated using this `_amountDepositedOnSource` should be a part of the Merkle tree.
    - **Impact**: The deposit amount.
- `_merkleProof`
    - **Control**: Full control.
    - **Constraints**: Should be a valid Merkle proof.
    - **Impact**: The provided `_merkleProof` is used to prove leaf membership.

Branches and code coverage

Intended branches

- Successful execution in the case of withdrawing when `weirollWallet` is executed.
    - ☑ Test coverage
- Successful execution in the case of withdrawing when `weirollWallet` is not executed yet.
    - ☑ Test coverage

Negative behavior

- Second try to withdraw after successful execution.
    - ☐ Negative test
- `msg.sender` is not a depositor.
    - ☐ Negative test
- Invalid `_merkleDepositNonce`.
    - ☐ Negative test
- Invalid `_amountDepositedOnSource`.
    - ☐ Negative test
- Invalid `_merkleProof`.
    - ☐ Negative test

Function call analysis

- `weirollWallet.marketHash()`
    - **What is controllable?** `weirollWallet`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the related `marketHash` provided during the wallet creation — the `marketHash` of this wallet should be match the provided `_sourceMarketHash`.
    - **What happens if it reverts, reenters or does other unusual control flow?** No problem — pure function.
- `weirollWallet.executed()`
    - **What is controllable?** `weirollWallet`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the `bool executed` global variable — if this wallet has been already executed, returns `true`. Allows to determine if the provided wallet has already been executed or not and process the withdrawal of tokens accordingly.
    - **What happens if it reverts, reenters or does other unusual control flow?** No problem.
- `receiptToken.safeTransferFrom(_weirollWallet, msg.sender, receiptTokensOwed)`
    - **What is controllable?** `_weirollWallet`.
    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** The `withdrawMerkleDeposit` function has the `nonReentrant` modifier to prevent reentrancy.
- `inputToken.safeTransferFrom(_weirollWallet, msg.sender, dustTokensOwed);`
    - **What is controllable?** `_weirollWallet`.
    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** The `withdrawMerkleDeposit` function has the `nonReentrant` modifier to prevent reentrancy.
- `inputToken.safeTransfer(msg.sender, amountOwed);`
    - **What is controllable?** `_weirollWallet`.
    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** The `withdrawMerkleDeposit` function has the `nonReentrant` modifier to prevent reentrancy.

5.2. Module: DepositLocker.sol

Function: `merkleBridgeSingleTokens(bytes32 _marketHash)`

This function allows users to perform Merkle Bridging operations for single token markets.

Inputs

- `msg.sender`
  - **Validation**: Checked to be the campaign owner of that specific market.
  - **Impact**: The caller of the function.
- `_marketHash`
  - **Validation**: Checked that the market is ready to be bridged in `readyToBridge`.
  - **Impact**: The market to which the deposits are being bridged.

Branches and code coverage (including function calls)

Intended branches

- Ensure that the `marketInputToken` is not a Uniswap V2 pair.
  - ☐ Test coverage
- Compose the message with the CCDM payload.
  - ☑ Test coverage
- Write the Merkle root and the deposits into the compose message.
  - ☑ Test coverage
- Normalize the deposit amount to the OFT's decimals.
  - ☑ Test coverage
- Ensure that at least one depositor was included in the bridge payload.
  - ☐ Test coverage
- Execute the bridging process via LayerZero.
  - ☐ Test coverage
- Refund any excess value sent with the transaction.
  - ☑ Test coverage
- Reset the Merkle tree and its accounting information for this market.
  - ☑ Test coverage
- Update the last CCDM bridge nonce to this nonce.

- ☐ Test coverage

Negative behavior

- Should not allow calling by a `msg.sender` that is not the campaign owner.
  - ☐ Test coverage
- Should not allow bridging LP tokens using this function.
  - ☐ Test coverage
- Should not allow bridging zero depositors (i.e., `totalAmountDeposited == 0`).
  - ☐ Test coverage

Function call analysis

- `WRAPPED_NATIVE_ASSET_TOKEN.withdraw(_amountToBridge)`
  - **External/Internal?** External.
  - **Argument control?** The amount to withdraw from the native wrapper, if the token is native.
  - **Impact**: Withdraws the amount from the native wrapper.
- `lzV2OFT.send(sendParam, messagingFee, address(this))`
  - **External/Internal?** External.
  - **Argument control?** `sendParam` (the parameters of the OFT bridge message), `messagingFee` (the fee to send), and `address(this)` (the address to send refund funds to).
  - **Impact**: Sends the OFT bridge message.
- `payable(msg.sender).call{ value: msg.value - bridgingFee }("")`
  - **External/Internal?** External.
  - **Argument control?** `msg.value - bridgingFee` (the amount to refund).
  - **Impact**: Refunds the excess value sent with the transaction.

Function: `merkleDeposit()`

This function allows users to perform deposits via the Merkle functionality.

Inputs

- `msg.sender.marketHash()`
  - **Validation**: Checked that a market input token exists for this market. This implies that the market is initialized.
  - **Impact**: The market to which the deposit is being made.

- `msg.sender.owner()`
    - **Validation**: None.
    - **Impact**: The owner of the Weiroll Wallet.
- `msg.sender.amount()`
    - **Validation**: None. Checked in `safeTransferFrom` that the amount is affordable.
    - **Impact**: The amount being deposited.

Branches and code coverage (including function calls)

Intended branches

- Should decrease the balance of `msg.sender` by `amountDeposited`.
    - ☐ Test coverage
- Should increase the balance of the contract by `amountDeposited`.
    - ☐ Test coverage
- Ensure that the market is legitimate.
    - ☐ Test coverage
- Ensure that the market is not halted.
    - ☐ Test coverage
- Initialize `merkleTree` with the default depth if it is uninitialized.
    - ☐ Test coverage
- Generate the deposit leaf based on the `depositor` and `amountDeposited`. Use a nonce so that the leaf is unique.
    - ☑ Test coverage
- Increase the nonce after it is used so that it cannot be reused.
    - ☐ Test coverage
- Push the deposit leaf into the `merkleTree`.
    - ☑ Test coverage
- Update the Merkle root and the total amount deposited into the Merkle tree.
    - ☐ Test coverage
- Update the amount this Weiroll Wallet has deposited into the currently stored Merkle tree. This is used in the case of withdrawals.
    - ☑ Test coverage
- Update the nonce after the deposit is made.
    - ☐ Test coverage

- Emit the `MerkleDepositMade` event.
    - ☐ Test coverage

Negative behavior

- Should not allow anyone other than a Weiroll Wallet to call this function. This is currently bypassable, as anyone can deploy a contract that bypasses the `onlyWeirollWallet` modifier.
    - ☐ Test coverage
- Should not allow deposits into a market that is not initialized.
    - ☐ Test coverage
- Should not allow deposits into a market that is halted.
    - ☐ Test coverage
- Should not allow reusing the same leaf — enforced through the use of nonce.
    - ☐ Test coverage

Function call analysis

- `marketInputToken.safeTransferFrom(msg.sender, address(this), amountDeposited)`
    - **External/Internal?** External.
    - **Argument control?** `msg.sender` (the Weiroll Wallet that is depositing), `address(this)` (the DepositLocker contract), and `amountDeposited` (the amount being deposited).
    - **Impact**: Transfers the `amountDeposited` from the Weiroll Wallet to the DepositLocker.

Function: `merkleWithdraw()`

This function allows users to perform withdrawals via the Merkle functionality. The withdrawals are not part of the normal flow of the contract but are instead allowed only when the market is halted.

Inputs

- `msg.sender`
    - **Validation**: None — assumed that it is a Weiroll Wallet, but it can basically be any contract.
    - **Impact**: The Weiroll Wallet that is withdrawing.
- `msg.sender.marketHash`

    - **Validation**: Checked that it is halted and that it implies that the `marketHash` exists.
    - **Impact**: The market hash that we are dealing with.
- `msg.sender.owner`
    - **Validation**: None — assumed that it is the owner of the Weiroll Wallet.
    - **Impact**: The owner of the Weiroll Wallet.

Branches and code coverage (including function calls)

Intended branches

- Ensure that the market is currently halted.
    - ☐ Test coverage
- Ensure that the caller has something to withdraw. This is enforced by the fact that the amount to withdraw is greater than zero.
    - ☐ Test coverage
- Update the `lastCcdmNonceBridgedToWeirollWalletToDepositAmount` mapping to reflect the withdrawal.
    - ☑ Test coverage
- Update the `merkleDepositsInfo.totalAmountDeposited` to reflect the withdrawal. Currently, this is not performed.
    - ☐ Test coverage
- Transfer the amount to the depositor.
    - ☐ Test coverage

Negative behavior

- Should not allow a nonstandard Weiroll Wallet to call this function. This is currently bypassable, as anyone can deploy a contract that bypasses the `onlyWeirollWallet` modifier.
    - ☐ Test coverage
- Should not allow withdrawals from a market that is not halted.
    - ☐ Test coverage
- Should not allow withdrawals if there is nothing to withdraw.
    - ☐ Test coverage
- Should not allow withdrawals if the amount to withdraw is greater than the total amount deposited.
    - ☐ Test coverage
- Should not allow altering the initial parameters of the Weiroll Wallet — currently not

enforced, and basically the initial parameters (such as the market hash, depositor, etc.) can be changed by the caller.

- ☐ Test coverage

Function call analysis

- `marketInputToken.safeTransfer(depositor, amountToWithdraw)`
  - **External/Internal?** External.
  - **Argument control?** `depositor` (the owner of the Weiroll Wallet) and `amountToWithdraw` (the amount being withdrawn).
  - **Impact**: Transfers the `amountToWithdraw` from the DepositLocker to the owner of the Weiroll Wallet.

6. Assessment Results

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

During our assessment on the scoped Royco CCDM contracts, we discovered one finding, which was 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.