Zellic

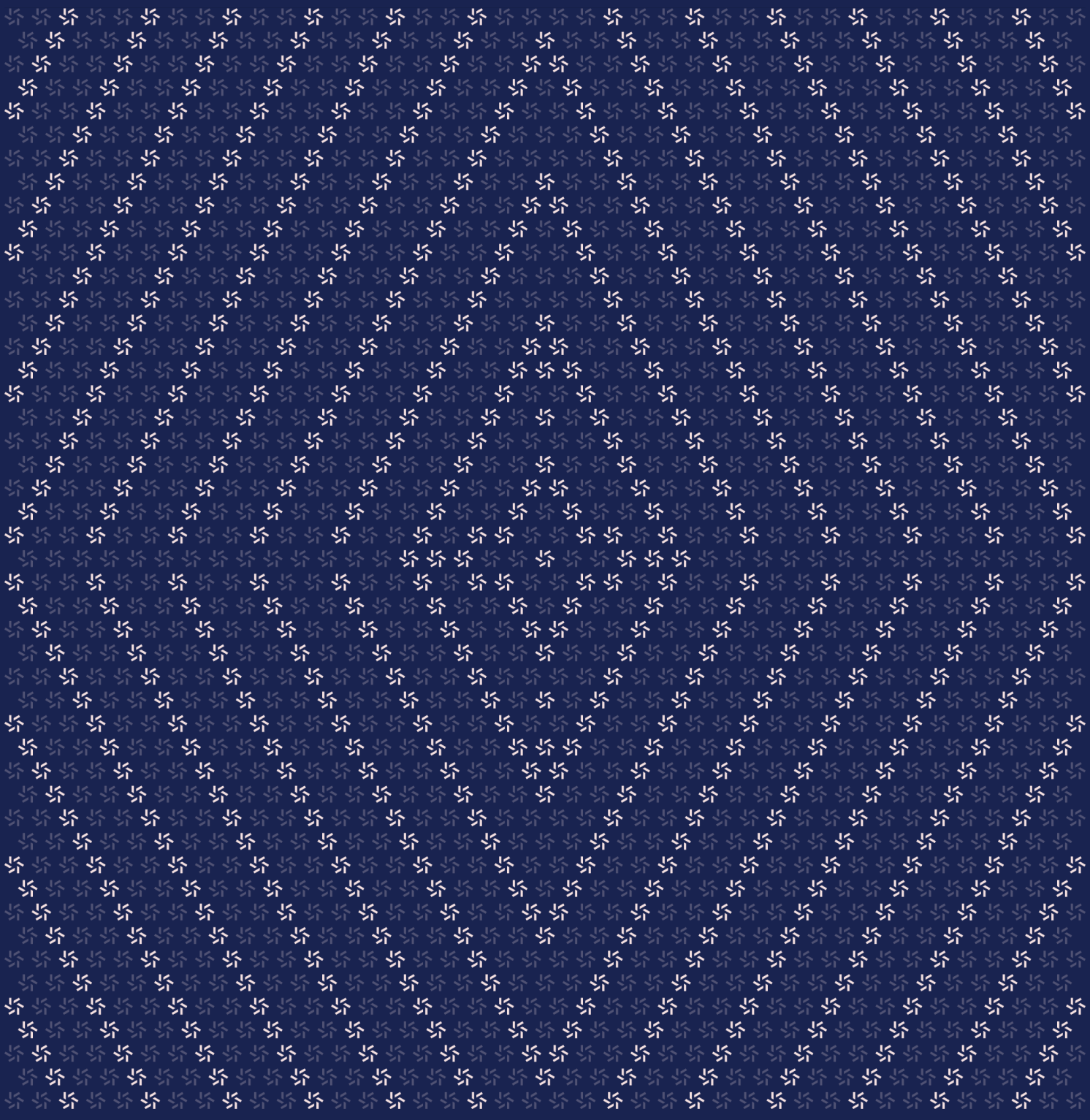
Prepared for
Matthew
Kite Labs

Prepared by
Chongyu Lv
Ulrich Myhre
Zellic

October 21, 2025

Carina Smart Contracts

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 Kite Labs from October 15th to October 17th, 2025. During this engagement, Zellic reviewed Carina smart contracts' 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:

- Is the custom ERC-1155 implementation secure?
- Are there any ways of withdrawing or vesting more funds than intended?
- Are access controls implemented effectively to prevent unauthorized operations?
- Are there any vulnerabilities that could result in the loss of user funds?

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 the assessment on the scoped Carina smart contracts, we discovered one finding, which was informational in nature.

Breakdown of Finding Impacts

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

2. Introduction

2.1. About Carina Smart Contracts

Kite Labs contributed the following description of the Carina smart contracts:

> Protocol Carina is a decentralized order settlement smart contract in Carina, an intent-based aggregation system.

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:

Carina Smart Contracts

Type	Solidity
Platform	EVM-compatible
Target	carina-sc
Repository	https://github.com/carina-finance/carina-sc ↗
Version	`6ff2ad112b6b62acb054411b8181d9414970b08b`
Programs	`src/**.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 4 person-days. The assessment was conducted by two consultants over the course of three calendar days.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

Pedro Moura
Engagement Manager
pedro@zellic.io

The following consultants were engaged to conduct the assessment:

Chongyu Lv
Engineer
chongyu@zellic.io

Ulrich Myhre
Engineer
unblvr@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

October 15, 2025	Kick-off call
October 17, 2025	Start of primary review period
October 17, 2025	End of primary review period

3. Detailed Findings

3.1. Withdrawal-fee griefing

Target	NativeTokenFlow		
Category	Optimization	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

By design, the NativeTokenFlow contract lets any user call into `wrapAllNativeToken()`, which takes the entire native balance of the contract and deposits it into a wrapped token counterpart. The caller pays the gas fees for this one-time deposit, but during a call to `cancelOrdersIgnoreInvalidCancellation(Orders[])` with multiple orders, the caller will pay the gas for one withdrawal per cancellation refund.

This opens up for a very minor griefing attack where one user can keep wrapping native tokens quite cheaply, but this makes the price for mass cancellations proportionally higher. It is possible to circumvent this by creating a new order just to fund the contract, cancel the required orders, and then cancel the new order. But this creates awkward use cases for users that want to be frugal with their gas.

Impact

Canceling multiple orders in a single call can be as costly as canceling orders one by one. For particularly large order cancellations, the required gas could reach transaction gas limits and fail. If the ERC-20 takes a fee for wrapping tokens, this attack can be quite devastating, but during the audit, we noticed no plans to use such a token.

If a fee exists, a malicious user could create hundreds of very small orders, call `wrapAllNativeToken()`, and then refund all their orders. In a normal situation, the user would pay the gas for this, but great care must be taken to avoid any token with a fee.

Recommendations

Since this is by design, it is not easy to fix without changing the design. One solution could be to not allow any user to deposit but require a more permissioned role like owner or a new depositor role that has this access.

Remediation

This issue has been acknowledged by Kite Labs, and a fix was implemented in commit 18a45298 ↗.

4. 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.

4.1. Module: NativeTokenFlow.sol

Function: `createOrder(Order order)`

This function creates a new order to exchange native token for another ERC-20 token. This will transfer the specified amount of native token from the sender to the contract.

Inputs

- `order`
  - **Control**: Almost full control. The `tokenIn` field is hardcoded to always be `WRAPPED_NATIVE_TOKEN`.
  - **Constraints**: `order.amountIn` must be nonzero and equal `msg.value`. The `order.validTo` field must not be in the past. The hash of the entire order struct cannot already exist in the `nativeTokenOrders` mapping, and the receiver must be a valid address.
  - **Impact**: Receives native tokens and creates a new order that stores metadata about the exchange order (native to ERC-20 token).

The `bytes16 quoteId` field is not part of the order digest and is meant for off-chain analysis. There is no uniqueness check on this field, and it only serves to be emitted in the `OrderCreated` event at the end. Do note that it is possible to create a separate order with the same `quoteId`, so event filtering must make certain to verify the order as well.

Branches and code coverage

Intended branches

- ☑ An order can be created (no fee).
- ☑ An order can be created (with a fee).

Negative behavior

- ☑ An order with zero `amountIn` reverts.
- ☑ An order with `amountIn` less than `msg.value` reverts.
- ☑ An order with `amountIn` greater than `msg.value` reverts.

- ☑ An order with `validTo` in the past reverts.
- ☑ An order with the receiver set to the zero address reverts.
- ☑ An order that already exists reverts.

Function: `_cancelOrder(Order order, bool revertIfInvalidCancellation)`

This function cancels an existing order. Only the owner of the order can cancel it before it expires. This function can be reached in two ways, either through the external function `cancelOrdersIgnoreInvalidCancellation()`, which takes an array of orders to cancel, or through `cancelOrder()`, which takes a single order to cancel. Both of these outer functions are non-reentrant.

The owner will be refunded for their order, and wrapped tokens will be unwrapped in order to accomplish this.

Inputs

- `order`
  - **Control**: Almost full control — `order.tokenIn` is set to `WRAPPED_NATIVE_TOKEN` inside `toSettlementOrder()`. However, the original struct is still used in some places.
  - **Constraints**: There are no direct constraints, but the hash of the settlement order must map to an order with status "created"; otherwise, the function returns early. If the order's `validto` field is in the future, only the owner can cancel it.
  - **Impact**: The order to cancel.
- `revertIfInvalidCancellation`
  - **Control**: Indirectly controllable by picking which external function to run.
  - **Constraints**: Boolean. Only true or false.
  - **Impact**: Decides if an error during cancellation should just return or revert. Do note that it can *still revert* if the refund to the owner is unsuccessful.

Branches and code coverage

Intended branches

- ☑ An owner can cancel an order that is not expired.
- ☑ An owner can cancel an order that is expired.
- ☑ Non-owners can cancel an order that is expired.

Negative behavior

- ☑ Settled orders revert on cancellation.

- ☑ Orders that are both settled and expired revert on cancellation.
- ☑ Nonexisting orders revert on cancellation.
- ☑ Canceled orders cannot be canceled.
- ☑ Non-owners cannot cancel nonexpired orders.
- ☑ Cancellation reverts if native-token transfer fails.

4.2. Module: Settlement.sol

Function: `computeTradeExecution(ISettlement.Trade trade, RecoveredOrder recoveredOrder, uint256 tokenInPrice, uint256 tokenOutPrice, TransferHelper.TransferFromData inTransfer, TransferHelper.TransferData outTransfer, TransferHelper.TransferData feeTransfer)`

This function is responsible for calculating the specific execution details of a single transaction. It is a crucial function in the Settlement contract's transaction-settlement process. It calculates the actual execution amount of the transaction based on the provided clearing price and verifies all transaction constraints.

Inputs

- `trade`
  - **Control**: Full control.
  - **Constraints**: Each trade must have a valid user signature, `tokenInIndex` and `tokenOutIndex` must be within the range of the `tokens` array, `validTo` must be greater than or equal to `block.timestamp`, `feeAmount` must be less than `amountIn`, the order cannot have been filled or canceled, and the actual output amount must be greater than or equal to `minAmountOut`.
  - **Impact**: Structure representing a trade to be executed in a settlement.
- `recoveredOrder`
  - **Control**: Generated by the internal function `recoverOrderFromTrade` and not directly controlled by the outside world. However, its content comes from the `trade` parameter and is indirectly affected by the solver.
  - **Constraints**: `order.validTo` must be greater than or equal to `block.timestamp`, `order.feeAmount` must be less than `order.amountIn`, the order ID value in the `orderFilledAmount` map must be 0, and `owner` must be a valid address recovered through signature verification.
  - **Impact**: Recovered trade data containing the extracted order and owner information.
- `tokenInPrice`
  - **Control**: Full control.

- **Constraints**: None.
- **Impact**: The price of the order's sell token.

- `tokenOutPrice`
  - **Control**: Full control.
  - **Constraints**: None.
  - **Impact**: The price of the order's buy token.
- `inTransfer`
  - **Control**: Full control.
  - **Constraints**: None.
  - **Impact**: Memory location for the computed executed sell-amount transfer.
- `outTransfer`
  - **Control**: Full control.
  - **Constraints**: None.
  - **Impact**: Memory location for the computed executed buy-amount transfer.
- `feeTransfer`
  - **Control**: Full control.
  - **Constraints**: None.
  - **Impact**: Memory location for the computed fee transfer.

Branches and code coverage

Intended branches

- ☑ This function correctly handles ERC-20 standard transfer transactions.
- ☑ This function correctly handles transactions with Permit2 signatures.

Negative behavior

- ☑ Revert if `order.validTo < block.timestamp`.
- ☑ Revert if `orderFilledAmount[orderId] != 0`.
- ☑ Revert if `order.feeAmount >= order.amountIn`.
- ☑ Revert if `actualAmountOut < order.minAmountOut`.

Function: `settle(IERC20[] tokens, uint256[] clearingPrices, Trade[] trades, Action[][Literal(value=3, unit=None)] actions)`

This function is a decentralized batch-settlement system where the authorized solver submits the optimal batch-settlement solution.

Inputs

- `tokens`
  - **Control**: Full control.
  - **Constraints**: None.
  - **Impact**: An array of ERC-20 tokens to be traded in the settlement.
- `clearingPrices`
  - **Control**: Full control.
  - **Constraints**: The length must be the same as the length of the `tokens` array, and the price value cannot be `0` (which will cause a division-by-zero error).
  - **Impact**: An array of token-clearing prices.
- `trades`
  - **Control**: Full control.
  - **Constraints**: Each trade must have a valid user signature, `tokenInIndex` and `tokenOutIndex` must be within the range of the `tokens` array, `validTo` must be greater than or equal to `block.timestamp`, `feeAmount` must be less than `amountIn`, the order cannot have been filled or canceled, and the actual output amount must be greater than or equal to `minAmountOut`.
  - **Impact**: Structure representing a trade to be executed in a settlement.
- `actions`
  - **Control**: Full control.
  - **Constraints**: `action.target` cannot be a `VAULT_RELAYER` address.
  - **Impact**: Structure representing an action to be executed during settlement.

Branches and code coverage

Intended branches

- ☑ The solver can successfully use this function to process ERC-20 standard transfer transactions.
- ☑ The solver can successfully use this function to process transactions with Permit2 signatures.
- ☑ The function correctly handles the fee and transfers it to the `feeReceiver`.
- ☑ The function can successfully process multiple transactions in batches.

Negative behavior

- ☑ Revert if the caller is not a solver.
- ☑ Revert if the order has expired.
- ☑ Revert if the order has been filled.
- ☑ Revert if the order has been canceled.
- ☑ Revert if `order.feeAmount >= order.amountIn`.

- ☑ Revert if `actualAmountOut < order.minAmountOut`.
- ☑ Revert with SigningSchemaEIP712Invalid if the signature is invalid.
- ☑ Revert if the signature is invalid.
- ☑ Revert if `action.target == address(VAULT_RELAYER)`.
- ☐ Revert if malicious tokens or contracts attempt to reenter the settlement.

4.3. Module: VaultRelayer.sol

Function: `transferFromAccounts(TransferHelper.TransferFromData[] transfers)`

This function acts as a trusted relayer, transferring tokens from multiple user accounts to the contract owner account in batches.

Inputs

- `transfers`
  - **Control**: Controlled only by the contract owner.
  - **Constraints**: None.
  - **Impact**: This parameter defines the complete contents of the batch-transfer operation, including from which user accounts (`account`), which tokens to transfer (`token`), the amount to transfer (`amount`), and the transfer method used (`transferType`).

Branches and code coverage

Intended branches

- ☑ An owner can successfully use this function to process ERC-20 transfers.
- ☑ An owner can successfully use this function to process Permit2 transfers.
- ☑ An owner can successfully use this function to process multiple transfers in batches.
- ☑ An owner can successfully call this function to mix two transfer types.

Negative behavior

- ☑ Revert if the caller of the `transferFromAccounts` function is not the owner.

5. Assessment Results

During the assessment on the scoped Carina smart contracts, we discovered one finding, which was informational in nature.

At the time of our assessment, the reviewed code was not yet deployed to any mainnet. While we have great confidence in the parts inspired by CoW Protocol, the addition of Permit2 flows adds additional attack surfaces. The codebase has very thorough and impressive test coverage that inspire confidence in that the developers have considered many avenues for branches for each individual function. The composition of these functions, however, is largely untested in this codebase, and there are no explicit end-to-end test scenarios that make sure that the entire chain from front end to contract — and back again — work as expected. We recommend creating such test flows that simulate the entire user experience, and also has extra focus on the code paths using Permit2, before deploying the code.

The tests could be manual too, but automating them will be helpful in future developments. As these types of tests are not typically found together with the contract code alone, this might already be implemented elsewhere, but we did not have access to them during the audit so this stands as a recommendation because of the advanced composition of code paths used in the real scenarios.

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.