Zellic

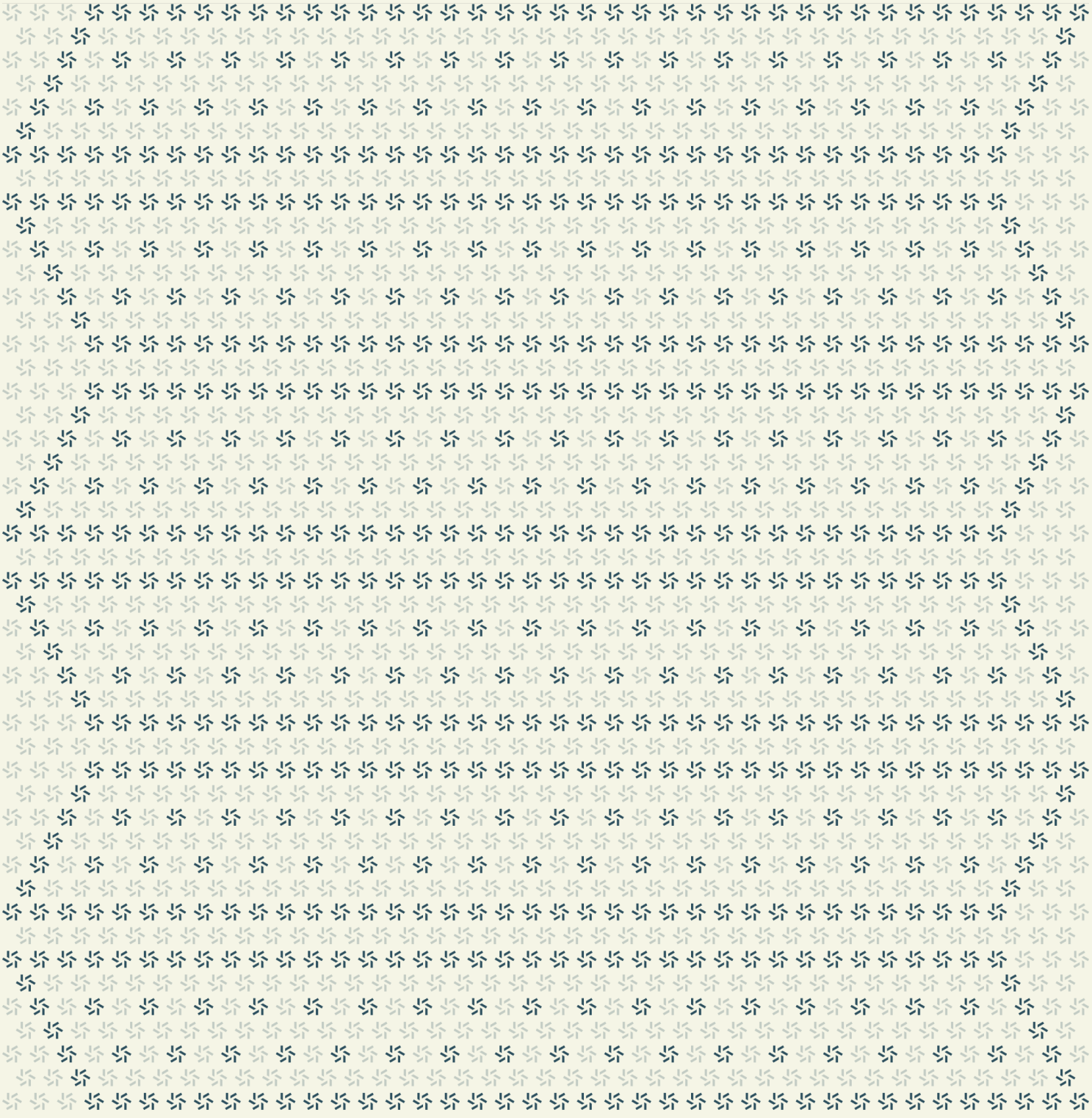
Prepared for
Dillon (COO)
Vincent (Engineer)
Nour Research

Prepared by
Filippo Cremonese
Zellic

February 22, 2025

Nous Psyche Mining Pool

Solana Program 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 Nour Research from February 22nd to February 23rd, 2025. During this engagement, Zellic reviewed Nous Psyche Mining Pool'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 primary questions:

- Is accounting of staked and claimed assets accurate?
- Is there any way for an unauthorized user to receive more reward than expected?
- Is there any ways to extract collateral from the pool without being the pool's delegate authority?

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

1.4. Results

During our assessment on the scoped Nous Psyche Mining Pool programs, we discovered one finding, which was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Nour Research 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 Nous Psyche Mining Pool

Nour Research contributed the following description of Nous Psyche Mining Pool:

> Psyche's Mining Pool smart contract is a public facing contract to raise funds for the psyche's training runs, while allowing retroactive reward to users who contributed to the training of a psyche's model.

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 programs.

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 programs 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:

Nous Psyche Mining Pool Programs

Type	Rust
Platform	Solana
Target	psyche
Repository	https://github.com/NousResearch/psyche ↗
Version	`0966d98cdbc22567bc9ccac330b89a73b88585c2`
Programs	`architectures/decentralized/solana-mining-pool/programs/solana-mining-pool/src/*`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of three person-days. The assessment was conducted by one consultant 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 ↗

The following consultant was engaged to conduct the assessment:

Filippo Cremonese
Engineer
fcremo@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

February 22, 2025	Start of primary review period
February 23, 2025	End of primary review period

3. Detailed Findings

3.1. Missing length check for pool metadata field

Target	Solana Mining Pool		
Category	Coding Mistakes	**Severity**	Low
Likelihood	N/A	**Impact**	Informational

Description

Pool accounts contain a `metadata` field of type `PoolMetadata`.

```
pub struct PoolMetadata {
    pub length: u16,
    pub bytes: [u8; PoolMetadata::BYTES],
}
```

This field is not used by on-chain code, and is meant to contain metadata used by offchain clients. The metadata is an arbitrary array of 400 bytes. The struct defining the metadata also has a `length` field, which is supposed to encode the length used in the fixed-size byte array. This length is not checked, and could assume values greater than the maximum length of 400.

Impact

There is no impact on on-chain programs, since they do not use the pool metadata field; the lack of this check could cause issues in off-chain clients that parse on-chain data.

Recommendations

Ensure that the `length` field does not contain a value greater than the maximum allowed length when processing `pool_create` and `pool_update` instructions.

Remediation

This issue has been acknowledged by Nour Research, and a fix was implemented in commit `3933bf1d` ↗.

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. Test suite

The testsuite of the project covers most of the main functionality of the program. It does not cover all possible negative test cases. Given the simplicity of the program, and the reliance on Rust types provided by Anchor to enforce basic checks, this is not considered a major issue.

The testsuite could be improved by making it more modular, adding unit tests and checks that test the behavior of individual instructions. This would increase the robustness and maintainability of the testsuite.

5. Threat Model

As time permitted, we analyzed each instruction in the program and created a written threat model for the most critical instructions. A threat model documents the high level functionality of a given instruction, the inputs it receives, and the accounts it operates on as well as the main checks performed on them; it gives an overview of the attack surface of the programs and of the level of control an attacker has over the inputs of critical instructions.

For brevity, system accounts and well known program accounts have not been included in the list of accounts received by an instruction; the instructions that receive these accounts make use of Anchor types which automatically ensure that the public key of the account is correct.

Discriminant checks, ownership checks and rent checks are not discussed for each individual account; unless otherwise stated, the program uses Anchor types which perform the necessary checks automatically.

Not all instructions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that an instruction is safe.

5.1. Mining pool program

Instruction: `pool_create`

This instruction allows to create a pool. The instruction is unpermissioned, as no special permission is needed to create a pool.

The pool is initialized using the following default settings:

- `max_deposit_collateral_amount`: 0
- `claiming_enabled`: false
- `redeemable_mint`: Not configured (zero public key)

The asset that will be used for pool claims is not set at the time of the creation of the pool, but at a later date using the `pool_claimable` instruction, which also permanently enables claims.

Input structure

```
pub struct PoolCreateParams {
    pub index: u64,
    pub metadata: PoolMetadata,
}
```

- `index`: index of the pool, used to disambiguate them and derive the pool PDA
- `metadata`: metadata for the pool; can be an arbitrary byte array

Accounts

- `payer`: Account that pays for account creation fees
    - Signer: Yes
- `authority`: Account that is set as authority for the pool
    - Signer: Yes
- `pool`: Account for the newly created pool
    - Init: Yes, the account is created by the instruction and cannot be already initialized
    - PDA: Yes, with seed `Pool::SEEDS_PREFIX`, `index`
- `pool_collateral`: spl-associated-token account associated with the pool, also initialized by this instruction
    - Init: Yes, the account is created by the instruction and cannot be already initialized
    - PDA: Yes, seed checked by Anchor, according to the spl-associated-token program
    - Rent: Checked (responsibility of spl-associated-token program)
    - Other checks: Must be a token account for the token minted by `collateral_mint`
- `collateral_mint`: Mint account of the pool collateral

Test coverage

The basic operation of this instruction is covered by the test suite.

Instruction: `pool_extract`

This instruction can be invoked by the pool authority to request a withdrawal of the pool collateral to an account controlled by the pool authority.

Input structure

```
pub struct PoolExtractParams {
    pub collateral_amount: u64,
}
```

- `collateral_amount`: Amount of collateral to be withdrawn

Accounts

- `authority`: Authority of the pool
    - Signer: Yes
- `authority_collateral`: Token account of the authority account
    - Other checks: Must be the account for the correct mint and not have delegated the authority to another account
- `pool`: Pool on which the instruction operates
    - PDA: Not explicitly checked (not a security issue)
    - Other checks: Ensures that the pool authority is the `authority` account
- `pool_collateral`: Pool collateral associated token account
    - PDA: Automatically checked by Anchor according to spl-associated-token derivation rules
    - Other checks: Must be controlled by the `pool` account

CPI

The spl-token program is invoked to perform a transfer of the requested amount from the pool's token account to the pool authority token account.

Test coverage

The testsuite covers the intended operation of this instruction, ensuring the correct authority can invoke the instruction; test robustness could be improved by checks that ensure token balances and pool accounting balances assume expected values.

The testsuite also ensures that an attempt to extract pool assets from the wrong authority fails.

Instruction: `pool_update`

This instruction can be used by the pool authority to update the `max_deposit_collateral_amount` and `metadata` parameters of a pool.

Note that the `max_deposit_collateral_amount` value can be updated to a value lower than the current one, or even lower than the current total amount of deposit assets. We have confirmed that this is intended behavior, meant to allow to stop deposits.

Input structure

```
pub struct PoolUpdateParams {
    pub max_deposit_collateral_amount: Option<u64>,
    pub metadata: Option<PoolMetadata>,
}
```

- `max_deposit_collateral_amount`: new value for the maximum collateral amount that can be deposited in the pool
- `metadata`: new value for the pool metadata

Accounts

- `authority`: Pool authority
    - Signer: Yes
- `pool`: Account of the pool to be updated
    - Other checks: Ensures that the pool authority is the `authority` signer account

Test coverage

The testsuite covers the functionality updating the `max_deposit_collateral_amount` value. It could be improved by adding a more explicit check on the `max_deposit_collateral_amount` stored in the pool account, and by also testing the functionality to update the pool `metadata`.

The testsuite does not contain negative testcases, such as attempts to use the incorrect authority.

Instruction: `pool_claimable`

This instruction can be used by the pool authority to enable claims for a pool, allowing to distribute rewards to the pool participants; this is a one-way operation: the pool can be switched to the claimable state only once.

This operation also specifies the asset that can be redeemed.

Accounts

- `authority`: Pool authority
    - Signer: Yes
- `pool`: Pool to switch to the claimable state

    - PDA: Not explicitly checked (not a security issue)
    - Other checks: Ensures that the pool authority is the `authority` account
- `redeemable_mint`: Mint of the redeemable asset.
    - Other checks: Must be an spl-token mint account

Test coverage

The testsuite covers the positive behavior of this instruction. It does not contain negative testcases, such as attempts to use the incorrect authority.

Instruction: `lender_create`

This instruction allows a user to create a lender account representing their position in a pool. Each public key can have a single lender position for any given pool.

Accounts

- `payer`: Payer of rent fees for the created `lender` account
    - Signer: Yes
- `user`: User associated with and controlling the created `lender` account
    - Signer: Yes
- `pool`: Pool associated with the `lender` account
- `lender`: Account storing the lender position
    - Init: Yes, the account is created by the instruction and cannot be already initialized
    - PDA: Yes, with seed `Lender::SEEDS_PREFIX`, `pool.key()`, `user.key()`

Test coverage

The testsuite covers this instruction, as it creates two lender accounts in the `memnet_full_cycle` scenario.

Instruction: `lender_deposit`

This instruction allows a lender to deposit collateral into a pool.

The deposit is allowed only if it does not increase the total amount of deposited collateral above the `max_deposit_collateral_amount` configured for the pool.

The processor for the instruction issues a CPI call to the spl-token program to transfer the

collateral and updates the accounting of the pool total deposited collateral and of the deposited collateral in the lender position.

Input structure

```
pub struct LenderDepositParams {
    pub collateral_amount: u64,
}
```

- `collateral_amount`: specifies the amount of collateral to be deposited

Accounts

- `user`: Identifies the user making the deposit and authorizes the token transfer
  - Signer: Yes
- `user_collateral`: Token account of the user performing the deposit
  - Other checks: Must be a token account for the pool collateral, owned by the depositor
- `pool`: Pool where the collateral is deposited
- `pool_collateral`: Token account of the pool
  - PDA: Yes, must be the unique spl-associated-token account associated with the `pool` account
- `lender`: Account representing the lender position associated with `pool` for `user`
  - PDA: Yes, with seeds `Lender::SEEDS_PREFIX`, `pool.key()`, `user.key()`, and fixed bump stored inside the `lender` account itself

CPI

The spl-token program is invoked to transfer the amount being deposited from the user token account to the pool token account.

Test coverage

The testsuite covers this instruction; the test scenarios perform multiple deposits.

The testsuite checks that a deposit fails the deposit cap is not respected. Corner cases are also checked, including the case of a newly initialized pool with an unset deposit cap (which defaults to zero), and the case where the total deposited is exactly the deposit cap, ensuring no off-by-one error occurs.

The testsuite could be improved by adding more explicit checks of the user balances and pool internal accounting.

Instruction: `lender_claim`

This instruction allows a lender to claim rewards from a pool. The instruction can only be used on pools for which claiming has been enabled.

The amount claimable by the lender is calculated as a share of the total claimable amount (the sum of the previously claimed assets and the assets still in the pool token account), proportional to the user deposited collateral over the total collateral deposited by all users. Lenders are only allowed to claim the remaining part of their claimable share.

In summary, the lenders are

The processor for this instruction updates the total claimed by the user, as well as the total claimed by all the users for the given pool, and invokes the spl-token program to perform the transfer of the claimed amount.

Input structure

```
pub struct LenderClaimParams {
    pub redeemable_amount: u64,
}
```

- `redeemable_amount`: Amount to be redeemed

Accounts

- `user`: Identifies the user performing the claim
  - Signer: Yes
- `user_redeemable`: Token account where the claimed assets are deposited
  - Other checks: Must be a token account for the claimable asset, owned by `user`, and without a delegate
- `pool`: The pool from which the user wants to redeem
  - Other checks: The `redeemable_mint` account must match the pool configuration
- `pool_redeemable`: spl-associated-token account for the redeemable asset, associated with `pool`
  - PDA: Yes, must be the unique spl-associated-token account associated with the `pool` account

- `redeemable_mint`: Mint for the redeemable asset
- `lender`: Account representing the lender position associated with `pool` for `user`
    - PDA: Yes, with seeds `Lender::SEEDS_PREFIX`, `pool.key()`, `user.key()`, and fixed bump stored inside the `lender` account itself

CPI

The spl-token program is invoked to transfer the amount being claimed from the pool redeemable asset token account to the user token account.

Test coverage

The basic functionality of this instruction is covered by the testsuite, in a scenario that simulates two users, claiming two different times, right after claimable assets are deposited to the pool claimable asset token account.

The testsuite also ensures that attempts to claim more than the amount of assets that should be claimable by the user fail.

6. Assessment Results

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

During our assessment on the scoped Nous Psyche Mining Pool programs, 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.