Zellic

Prepared for
Dillon Rolnick
Vincent Brunet
Nous Research

Prepared by
Aaron Esau
Maik Robert
Zellic

July 3, 2025

Psyche Treasurer
Solana Application 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 Nous Research from July 1st to July 3rd, 2025. During this engagement, Zellic reviewed Psyche Treasurer's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

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

- Are there bugs that allow for unauthorized extraction of collateral?
- Are rewards allocated to participants correctly?

1.3. Non-goals and Limitations

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

- Front-end components
- Infrastructure relating to the project
- Key custody

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

1.4. Results

During our assessment on the scoped Psyche Treasurer programs, we discovered three findings. No critical issues were found. One finding was of high impact, one was of medium impact, and the remaining finding was informational in nature.

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	1
🟨 Medium	1
🟩 Low	0
⬜ Informational	1

2. Introduction

2.1. About Psyche Treasurer

Nous Research contributed the following description of Psyche Treasurer:

> The Psyche Treasurer provides an Incentive layer on top of the Psyche's coordinator program. It allows distributing rewards to coordinator's participants.

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.

2.3. Scope

The engagement involved a review of the following targets:

Psyche Treasurer Programs

Type	Rust
Platform	Solana

Target	psyche
Repository	https://github.com/PsycheFoundation/psyche ↗
Version	`2c6f4c8eb5b8c5bebe01ecc791b72fd836656cc4`
Programs	`architectures/decentralized/solana-treasurer/src/**.rs`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 4.5 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 ↗

The following consultants were engaged to conduct the assessment:

Aaron Esau
Engineer
aaron@zellic.io ↗

Maik Robert
Engineer
maik@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

July 1, 2025	Kick-off call
July 1, 2025	Start of primary review period
July 3, 2025	End of primary review period

3. Detailed Findings

3.1. Lack of comprehensive tests

Target	Solana-Treasurer		
Category	Protocol Risks	**Severity**	High
Likelihood	N/A	**Impact**	High

Description

There are no tests, besides the integration test at solana-tooling/tests/suites/memnet_treasurer_full_epoch.rs (which only creates a run and configures its params).

When building a complex 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.

Therefore, we recommend building a rigorous test suite that includes all programs 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.

Impact

In our opinion, there is a higher risk of bugs and unintended behavior because of the lack of tests.

Recommendations

The test coverage for this project should be expanded to include all programs and instructions, and not just surface-level functions. It is important to test the invariants required for ensuring security and also verify properties from the specification. We always recommend having both positive (expected success), negative (expected failure) and fuzz test cases (unexpected inputs handled correctly) especially for instructions that involve moving funds. Some of the instructions to focus on would be `participant_claim`, `run_top_up` and `run_update` as they are directly responsible for fund transfers.

Remediation

This issue has been acknowledged by Nous Research, and a fix was implemented in `PR #155` ↗.

3.2. Unoptimized for loop with potentially unwanted behavior

Target	src/logic/participant_claim.rs		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The handler for the `participant_claim` instruction uses a for loop to iterate over `clients`.

```
let mut participant_earned_points = 0;
    for client in context
        .accounts
        .coordinator_account
        .load()?
        .state
        .clients_state
        .clients
        .iter()
    {
        if client.id.signer == context.accounts.user.key() {
            participant_earned_points = client.earned;
        }
    }
```

It is unclear what the intended behavior of this for loop should be. The `clients` vector contains up to 256 entries, which may or may not be unique.

In the case they are unique, the for loop will iterate over all 256 possibilities even though we may have already found our target at index `0`, leading to a lot of wasted compute units because the for loop does not break.

In the case the values are not unique, and one client may be represented multiple times with different reward amounts, the for loop will only take into account the last found entry in the `Vec`. This is because `participant_earned_points` is assigned to `client.earned` instead of added, if the intended behavior is adding up all earned rewards.

Impact

The impact depends on the intended behavior of the code. Due to the unclear nature of the for loop, we cannot determine which of the following impacts the loop has. It either

1. wastes considerable compute units by unnecessarily iterating over the entire `Vec`, or
2. miscounts the earned rewards.

In either case, we consider there to be a medium impact.

Recommendations

Clear up the logic of the for loop, and either `break` if the target client was found or switch to correctly adding up all found rewards for a `client`.

Remediation

This issue has been acknowledged by Nous Research, and a fix was implemented in `PR #152` ↗.

3.3. Superfluous signers in `run_top_up` and `participant_claim`

Target	src/logic/participant_claim.rs, src/logic/run_top_up.rs		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Two instructions, `participant_claim` as well as `run_top_up`, require a `payer` account as an additional signer.

```
#[derive(Accounts)]
#[instruction(params: ParticipantClaimParams)]
pub struct ParticipantClaimAccounts<'info> {
    #[account(mut)]
    pub payer: Signer<'info>,

    #[account()]
    pub user: Signer<'info>,
...
}
```

```
#[derive(Accounts)]
#[instruction(params: RunTopUpParams)]
pub struct RunTopUpAccounts<'info> {
    #[account(mut)]
    pub payer: Signer<'info>,

    #[account()]
    pub authority: Signer<'info>,
...
}
```

This account is not actually used in either function and not required by any of the Anchor constraints.

Impact

Having additional unrequired signers could lead to issues where these are passed via CPI to potentially malicious programs. For both instructions, only known, safe programs are called via CPI, but the unrequired account should be removed regardless.

Recommendations

Remove the `payer` account from the requirements, as the account is not used in either handler.

Remediation

This issue has been acknowledged by Nous Research, and a fix was implemented in `PR #152` ↗.

4. 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.

4.1. Program: `psych_solana`

Instruction: `participant_create`

This instruction creates a new participant account for a client in a specific run.

Input parameters

```
pub struct ParticipantCreateParams {}
```

Accounts

- **`payer`**
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: Yes. However, the instruction does not mutate the account.
  - Constraints: None.
- **`user`**
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: None.

- **`run`**
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: Yes.
  - Constraints: Must be a `Run` account.
- **`participant`**
  - Signer: No.
  - Init: Yes.
  - PDA: Yes.
  - Mutable: Yes.
  - Constraints: Must be a unique participant account for the `run` and `user`.
- **`system_program`**
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be the system program account.

Instruction: `participant_claim`

This instruction distributes the earned points in the form of collateral to the participant.

Input parameters

```
pub struct ParticipantClaimParams {
    pub claim_earned_points: u64,
}
```

Accounts

- **`payer`**
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: Yes; however, the account is not mutated.
  - Constraints: None.

- **`user`**
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: None.
- **`user_collateral`**
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes; the account is mutated to transfer collateral.
    - Constraints: Must be associated with the user's collateral token account. Must not be delegated to anyone.
- **`run`**
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes; the account is muted to update the total claimed collateral amount and earned points.
    - Constraints: Must be a `Run`. Must have the correct `coordinator_account`.
- **`run_collateral`**
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes; the account is mutated to transfer collateral to the user.
    - Constraints: Mint must be `run`'s collateral mint. And `run` must be the authority.
- **`coordinator_account`**
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be a `CoordinatorAccount`. Must match the `coordinator_account` in the `run`.
- **`participant`**
    - Signer: No.
    - Init: No.
    - PDA: Yes; derived from `run` and `user`.

- Mutable: Yes; the account is mutated to update its claimed collateral amount and earned points.
- Constraints: Must be a `Participant` account.

Additional checks and behavior

The `claim_earned_points` is constrained to be less than or equal to the earned points minus the previously claimed points.

CPI

It performs a CPI to the `token` program to transfer the collateral from the `user_collateral` account to the `run_collateral` account.

Instruction: `run_create`

This instruction creates a new run in the PsycheSolana program. It initializes the run account and its associated collateral account.

Input parameters

```
pub struct RunCreateParams {
    pub index: u64,
    pub run_id: String,
    pub main_authority: Pubkey,
    pub join_authority: Pubkey,
}
```

Accounts

- **`payer`**
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: None.
- **`run`**
  - Signer: No.

  - Init: Yes (created by instruction).
  - PDA: Yes (seeds: `[Run::SEEDS_PREFIX, params.index.to_le_bytes()]`, bump).
  - Mutable: Yes.
  - Constraints: Space allocated with `Run::space_with_discriminator()`. Payer is `payer`.
- **`run_collateral`**
  - Signer: No.
  - Init: Yes (associated token account created).
  - PDA: Yes (associated token account PDA).
  - Mutable: Yes.
  - Constraints: Must be an associated token account with mint `collateral_mint` and authority `run`.
- **`collateral_mint`**
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be an SPL Token Mint account.
- **`coordinator_instance`**
  - Signer: No.
  - Init: No.
  - PDA: N/A.
  - Mutable: Yes.
  - Constraints: Checked only in CPI to coordinator program.
- **`coordinator_account`**
  - Signer: No.
  - Init: No.
  - PDA: N/A.
  - Mutable: Yes.
  - Constraints: Checked only in CPI to coordinator program.
- **`coordinator_program`**
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be the PsycheSolanaCoordinator program.
- **`associated_token_program`**

    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the SPL Associated Token program.
- **`token_program`**
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the SPL Token program.
- **`system_program`**
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the System program.

CPI

It makes an `InitCoordinatorAccounts` instruction, which initializes the coordinator accounts for the run. The CPI is made to the PsycheSolanaCoordinator program.

Instruction: `run_update`

This instruction updates the configuration and metadata of the `Run` account.

Input parameters

```
pub struct RunUpdateParams {
    pub metadata: Option<RunMetadata>,
    pub config: Option<CoordinatorConfig>,
    pub model: Option<Model>,
    pub progress: Option<CoordinatorProgress>,
    pub epoch_earning_rate: Option<u64>,
    pub epoch_slashing_rate: Option<u64>,
    pub paused: Option<bool>,
}
```

Accounts

- **`authority`**
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: None.
- **`run`**
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: No.
    - Constraints: `main_authority` must be the `authority` key, `coordinator_instance` must be the `coordinator_instance` key, and `coordinator_account` must equal the `coordinator_account` key.
- **`coordinator_instance`**
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be a `CoordinatorInstance`.
- **`coordinator_account`**
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Must be a `CoordinatorAccount`.
- **`coordinator_program`**
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the `PsycheSolanaCoordinator` program.

CPI

This instruction makes the `update`, `set_future_epoch_rates`, and `set_paused` CPI calls.

Instruction: `run_top_up`

This instruction allows the authority to top up the collateral of a run.

Input parameters

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

Accounts

- **`payer`**
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: None.
- **`authority`**
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: None.
- **`authority_collateral`**
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: The mint must match the `run.collateral_mint`, and the authority must be the `authority` key. Finally, no delegate should be set on the account.
- **`run`**
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: Yes.
  - Constraints: `main_authority` should match the `authority` key.

- **`run_collateral`**
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: Yes.
  - Constraints: Associated token account with mint `collateral_mint` and authority `run`.
- **`collateral_mint`**
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be a Mint account.
- **`token_program`**
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be the SPL Token program.

CPI

Transfers `collateral_amount` from `payer` to `authority_collateral` using the SPL Token program.

5. Assessment Results

During our assessment on the scoped Psyche Treasurer programs, we discovered three findings. No critical issues were found. One finding was of high impact, one was of medium impact, and the remaining finding was informational in nature.

5.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.