Zellic

Prepared for
Pyth Data Association

Prepared by
Jasraj Bedi
Filippo Cremonese
Zellic

January 17, 2025

Pyth Lazer

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 Pyth Data Association from December 11th to December 12th, 2024. During this engagement, Zellic reviewed Pyth Lazer'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:

- Is the price feed signature verification functionality implemented correctly and securely?
- Is the administrative functionality implemented correctly and securely?

1.3. Non-goals and Limitations

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

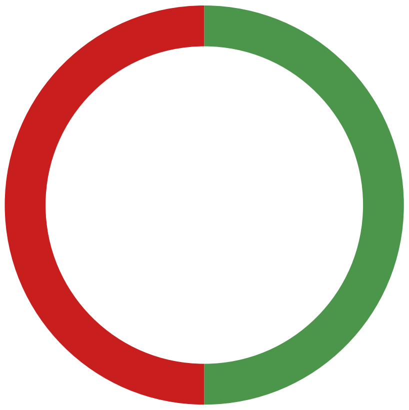
- Front-end components
- Back-end components
- Infrastructure relating to the project
- Key custody

1.4. Results

During our assessment on the scoped Pyth Lazer programs, we discovered two findings. One critical issue was found. One was of low impact.

Breakdown of Finding Impacts

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



2. Introduction

2.1. About Pyth Lazer

Pyth Data Association contributed the following description of Pyth Lazer:

> Lazer is a blazing-fast oracle with focus on latency, and efficiency. It streams price updates in a flexible format for on-chain consumption.

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:

Pyth Lazer Programs

Type	Rust
Platform	Solana

Target	Pyth monorepo
Repository	https://github.com/pyth-network/pyth-crosschain ↗
Version	`2faddf96bc22e1aedc1f72f929cb1a64b520db6a`
Programs	`lib.rs` `signature.rs`

2.4. Project Overview

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

Jasraj Bedi
Co-founder
jazzy@zellic.io ↗

Filippo Cremonese
Engineer
fcremo@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

December 11, 2024	Start of primary review period
December 12, 2024	End of primary review period

3. Detailed Findings

3.1. Signature bypass

Target	Lazer		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	Low	**Impact**	Critical

Description

`verify_message` incorrectly assumes that the layout of the calldata passed to the `VerifyMessage` instruction is as follows:

```
 Offset  Size  Handler argument name/Significance
      0     8  ANCHOR HEADER
      8     4  message_data length (4+64+32+2+psz)
     12        message_data
     12     4    SOLANA_FORMAT_MAGIC
     16    64    signature
     80    32    pubkey
    112     2    payload size (psz)
    114   psz    payload
114+psz     2  ed25519_instruction_index
116+psz     1  signature_index
117+psz     2  message_offset
```

Under this assumption, `message_offset` has a value of 12, referencing the offset of the data of the `message_data` vector in the raw encoding of the calldata for the `VerifyMessage` instruction.

The `verify_message` function performs several checks on the data contained in `message_data` using offsets computed assuming the storage layout outlined above.

However, if the calldata layout and `message_offset` are adjusted in the following way:

```
       Offset  Size  Handler argument name/Significance
            0     8  ANCHOR HEADER
            8     4  message_data length (4+64+32+2+psz)
           12        message_data
           12     4    SOLANA_FORMAT_MAGIC
           16    64    <UNUSED>
           80    32    PK1
          112     2    payload1 size (psz1)
          114  psz1    payload1
    114+psz1     4    <UNUSED>
```

```
     118+psz1    64    valid signature for PK2
     182+psz1    32    PK2
     214+psz1     2    payload2 size (psz2)
     216+psz1   psz    payload2
216+psz1+psz2     2  ed25519_instruction_index
218+psz1+psz2     1  signature_index
219+psz1+psz2     2  message_offset (114+psz1)
```

The `verify_message` function will read the public key, payload size and payload appearing earlier in the calldata, while the ec25519 instruction will reference the latter instances.

Thus, `verify_message` will incorrectly check that the PK1 public key appears in the list of trusted signers, while the ec25519 instruction will require a signature from a different PK2 public key. Additionally, the signed payload and the payload returned by `verify_message` also differ arbitrarily both in their content and their size.

Impact

If exploited, this issue allows to completely bypass signature verification of a price feed update, allowing to forge a price feed update with an arbitrary payload and that appears to originate from any address in the set of trusted signers.

The exploitability of this issue is conditional on the program invoking the `VerifyMessage` instruction. The attacker needs to have control over the `message_offset` provided to `VerifyMessage`. The official integration examples provided by Pyth Data Association did not allow the end user to specify the offset provided to the `VerifyMessage` instruction. These preconditions lower the likelyhood of the vulnerability. However, Lazer integrations are not required to use the examples provided by Pyth Data Association, and there seems to be some more advanced and reasonable scenarios where a program integrating with Lazer could give an attacker control over the offset argument.

Recommendations

It appears to be possible to entirely remove the `message_offset` argument, fixing its value to 12, which would constrain the data referenced by the ec25519 program to be aligned with the start of the `message_data` argument of the `VerifyMessage` instruction. Note that the offset used by the ec25519 program would still need to be checked to be 12 if removing the `message_offset` argument.

Remediation

This issue has been acknowledged by Pyth Data Association, and a fix was implemented in PR #2250 ↗.

The PR added a check requiring the `message_data` processed by the `verify_message` function to match the data considered by the ec25519 program.

Additionally, the `message_offset` argument indicating the start of the message data processed by the ec25519 program was removed.

The code could not be simplified further by requiring a stricter layout for the data processed by the ec25519 program, as Pyth Data Association indicated this flexibility is a requirement.

3.2. Migration applicable to already migrated storage account

Target	Lazer		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

`migrate_from_0_1_0` allows the `top_authority` to migrate the `storage` account from the older `StorageV010` model to the new `Storage` data model. This resizes the account, migrates the `trusted_signers`, and sets the `treasury` based on a value passed in by the `top_authority`. The migration requires a few conditions to be true, but one of the most important bits is that the anchor signature is equivalent to the `Storage` account's signature. Notably, this means that the migration does *not* change the anchor signature, because the name of the type for the account has not changed.

As the preliminary checks that allow the migration to go forward are true both before and after the migration, nothing prevents the `top_authority` from rerunning the migration again.

Impact

The `top_authority` could reset the `treasury` and/or putting the `trusted_signers` in a potentially corrupted state. This could be done by causing the `num_trusted_signers` to be set to an invalid value, breaking the signature validation logic.

Recommendations

As Anchor's discriminant is primarily dependant on the name of the type of the account, we recommend changing the name of the account from `Storage` to a a distinct name such as `StorageV2`. The migration step would validate that the account had a discriminant expected for `Storage`, but would eventually write `StorageV2`'s discriminant to the account. This would prevent subsequent migrations as the discriminant of the account would change.

Remediation

This issue has been acknowledged by Pyth Data Association, and a fix was implemented in PR #2250 ↗.

The PR added a check requiring the size of the `Storage` account to match the size of the legacy layout. Since the legacy and new layouts differ in size, this prevents the migration from being performed more than once.

Pyth Data Association subsequently removed the migration feature entirely in commit `800bb1aa` ↗.

4. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the programs 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. Program: Lazer

`initialize`

This instruction can be used to create and initialize the singleton `storage` account. The instruction handler function does not enforce access control, and therefore it could technically be invoked by any caller. However, the `storage` account being created has a fixed seed. For this reason, `initialize` can only be called once. While this does not directly prevent someone to race to call the instruction and initialize the `storage` account, this action would be detected and could be addressed by simply redeploying and reinitializing a new instance of the program.

The `storage` account `top_authority` and `treasury` fields are initialized to public key values given as arguments to the `initialize` instruction, and `single_update_fee_in_lamports` is initialized to 1.

Arguments

- `top_authority`: public key value used to initialize the `top_authority` field in the newly created `storage` account
- `treasury`: public key value used to initialize the `treasury` field in the newly created `storage` account

Accounts

- `payer`: account paying for the fees needed to create the `storage` account
    - Anchor flags: signer, mut
- `storage`: new account being initialized
    - Anchor flags: init
    - Payer: `payer`
    - Seed: `STORAGE_SEED` (fixed constant)
    - Space: manually specified

Tests

Happy path behavior is implicitly tested by the testsuite setup routines.

`migrate_from_0_1_0`

This instruction can be used to migrate the singleton `storage` account in-place from the legacy layout to the current storage layout.

The instruction handler takes the `storage` account as a raw `AccountInfo`, manually verifies the discriminant and deserializes it according to the legacy layout.

It then ensures that the `top_authority` signer account matches the `top_authority` field of the deserialized storage account.

The storage account is reallocated, increasing its size to account for the additional space needed by the newer `Storage` account type.

Finally, the function reinitializes the account, by instantiating an instance of the new `Storage` account type and serializing it manually into the `storage` account.

This function does not prevent to migrate an already migrated `storage` account, allowing to corrupt its data. Refer to finding 3.2. ↗ for a more in depth discussion.

Arguments

- `treasury`: public key value used to initialize the `treasury` field of the migrated `storage` account

Accounts

- `top_authority`: Signer account giving authorizing the migration
  - Anchor flags: signer
  - Checks: must match the `top_authority` field of the `storage` account when decoded using the legacy layout
- `storage`: storage account to be migrated; taken as a raw `AccountInfo`
  - Anchor flags: mut
  - Seed: `STORAGE_SEED` (fixed constant)

Tests

Positive cases

- ✓ A legacy Storage account can be migrated successfully, with the new Storage account layout correctly reflecting the previous configuration

Negative cases

- [x] Rejects attempts to migrate the Storage account multiple times
  - Note: this test was added as part of the remediations
- [ ] Rejects attempts to migrate the Storage account by an unauthorized account

`update`

This instruction allows to manage the set of trusted signers. It can only be invoked with a signature from the `top_authority` account.

Arguments

- `trusted_signer`: Public key value of the trusted signer to be added/removed/modified
- `expires_at`: Expiration date to be associated to the trusted signer; if zero, signifies the trusted signer should be removed from the signer set

Accounts

- `top_authority`: Signer account giving authorizing the migration
  - Anchor flags: signer
  - Checks: must match the `top_authority` field of the `storage` account
- `storage`: storage account to be migrated; taken as a raw `AccountInfo`
  - Anchor flags: mut
  - Seed: `STORAGE_SEED` (fixed constant)
  - Notes: the storage account must have been migrated, deserialization would fail for a legacy-layout account due to mismatching size

Tests

Positive cases

- [x] Allows the authorized administrator address to add a trusted signer
  - Note: this is implicitly tested by the test setup procedures
- [ ] Allows the authorized administrator address to remove a trusted signer
- [ ] Allows the authorized administrator address to update a trusted signer

Negative cases

- [ ] Rejects calls from unauthorized signers

`verify_message`

This instruction can be used to verify the legitimacy of a price feed update.

The instruction handler collects the required fee amount specified by the `single_update_fee_in_lamports` of the `storage` account, transferring it from the `payer` account to the `treasury` account.

It then verifies that the message data was correctly signed by ensuring that the transaction contains an instruction invoking the ed25519 system program which ensures a price feed update was validly signed by one of the currently active trusted signers.

Verification of the ed25519 instruction requires several checks, including:

- ensuring that the instruction indices used to retrieve the signature, public key, and signed message by the ed25519 instruction refer to the current `verify_message` instruction
- ensuring that the ed25519 instruction index is lower than the current `verify_message` instruction
- (implicitly) ensuring that at least one signature is present
- ensuring that the signed data starts with the magic value `SOLANA_FORMAT_MAGIC_LE` (`2182742457`) to disambiguate messages that may have been signed for other purposes
- ensuring the signer pubkey appears in the set of trusted signers
- ensuring the signer trust has not expired
- ensuring that the offsets for the signature, signer pubkey, and payload provided to the ed25519 instruction are consistent with the offsets used by `verify_message`
- ensuring that the payload length is consistent

Note that the offset checks are implemented incorrectly, allowing a complete bypass of the signature verification checks. Refer to finding 3.1. ↗ for a more in-depth discussion of the issue.

Arguments

- `message_data`: signed message that is being verified
- `ed25519_instruction_index`: index of the `ed25519_program` instruction within the transaction. This instruction must precede the current instruction
- `signature_index`: index of the signature within the inputs to the `ed25519_program`
- `message_offset`: offset of the signed message within the input data for the current instruction

Accounts

- `payer`: account paying for the price feed verification fees
  - Anchor flags: signer, mut
- `storage`: account storing Lazer configuration
  - Seed: `STORAGE_SEED` (fixed constant)

    - Notes: the storage account must have been migrated, deserialization would fail for a legacy-layout account due to mismatching size
- `treasury`: Treasury token account used to collect fees
    - Checks: Must match the `treasury` field of the `storage` account
- `instructions_sysvar`: Sysvar account used for instruction introspection
    - Notes: this is taken as a raw AccountInfo; the account ID is checked by the Solana SDK functions that receive it as an argument

Tests

Positive cases

- [x] Allows to verify a payload signed by an authorized signer

Negative cases

- [ ] Rejects payloads with an invalid signature (invocation of a program different than ec25519)
- [ ] Rejects payloads with an invalid signature (not originating from a trusted signer)
- [ ] Rejects payloads with an invalid signature (originating from an expired trusted signer)
- [ ] Rejects invalid payloads (not starting with the correct magic)
- [x] Rejects all combinations of invalid offsets

- Note: this test was added as part of the remediations

5. Assessment Results

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

During our assessment on the scoped Pyth Lazer programs, we discovered two findings. One critical issue was found. One was of low impact.

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.