Zellic

Prepared for
Pyth Data Association

Prepared by
Sunwoo Hwang
Doyeon Park
Zellic

February 12, 2025

Lazer
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 Pyth Data Association from February 10th to February 11th, 2025. During this engagement, Zellic reviewed 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 signature-verification process secure?
- Is the parsing logic of the library correct?
- Is the signer-management process secure?

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

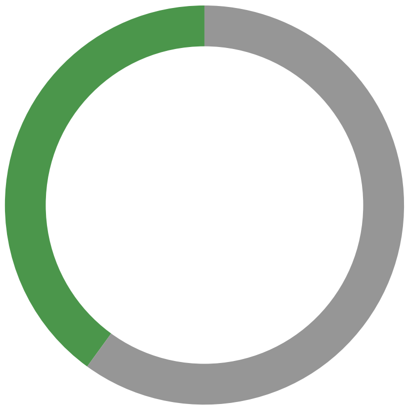
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide. During this assessment, gaps in response times as well as a lack of developer-oriented documentation and thorough end-to-end testing impacted the momentum of our auditors.

1.4. Results

During our assessment on the scoped Lazer contracts, we discovered five findings. No critical issues were found. Two findings were of low impact and the remaining findings were informational in nature.

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	0
🟨 Medium	0
🟩 Low	2
⬜ Informational	3



2. Introduction

2.1. About Lazer

Pyth Data Association contributed the following description of 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 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:

Lazer Contracts

Type	Solidity
Platform	EVM-compatible

Target	pyth-crosschain
Repository	https://github.com/pyth-network/pyth-crosschain ↗
Version	`c6a2eb91c7ed50de8553f6a64c766ff1d2e58b5a`
Programs	`PythLazer.sol` `PythLazerLib.sol`

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:

Sunwoo Hwang
Engineer
sunwoo@zellic.io ↗

Doyeon Park
Engineer
doyeon@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

February 10, 2025	Kick-off call
February 10, 2025	Start of primary review period
February 11, 2025	End of primary review period

3. Detailed Findings

3.1. Signature malleability

Target	PythLazer.sol		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The current implementation recovers the `signer` using the raw `ecrecover` function. Unlike OpenZeppelin's ECDSA library, this implementation does not include signature-malleability checks. The `ecrecover` function accepts both 27 and 28 as valid values for the `v` parameter and allows both lower and upper values for the `s` parameter. This means that multiple valid signatures can exist for the same message and signer.

```
signer = ecrecover(
    hash,
    uint8(update[68]) + 27,
    bytes32(update[4:36]),
    bytes32(update[36:68])
);
```

Impact

While this does not impact the security of the contract itself, depending on the use case in the consumer contract, accepting two different signatures from the same signer may lead to unexpected behavior (e.g., when the consumer contract manages signatures in a batch).

Recommendations

It is recommended to use OpenZeppelin's ECDSA library instead of the raw `ecrecover` function.

Remediation

This issue has been acknowledged by Pyth Data Association, and a fix was implemented in commit 49de9a23 ↗.

3.2. Improper initialization within the UUPS pattern

Target	PythLazer.sol		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

In the UUPS pattern, the upgrade functionality of the proxy is typically found within the implementation contract.

```
// PythLazer.sol::initialize()
    function initialize(address _topAuthority) public initializer {
        __Ownable_init(_topAuthority);
        __UUPSUpgradeable_init();

        verification_fee = 1 wei;
    }
```

Prior to OpenZeppelin version 4.7.3 and the Shanghai fork, a common attack technique involved frontrunning the `initialize` function of an implementation contract and using `delegatecall` within the `upgradeToAndCall` function to self-destruct the implementation. Since self-destruct has been deprecated on ETH mainnet, this specific attack vector is no longer a concern.

```
// UUPSUpgradeable.sol::upgradeToAndCall()
    function upgradeToAndCall(address newImplementation, bytes memory data)
    public payable virtual onlyProxy {
        _authorizeUpgrade(newImplementation);
        _upgradeToAndCallUUPS(newImplementation, data);
    }
```

However, the invocation of `initialize` in the implementation contract can still be frontrun. If this function contains critical logic, it may introduce another attack vector with potentially widespread implications for the protocol.

To clarify, this is not a security vulnerability but a design consideration for upgradeable contracts.

Impact

While any user could potentially take over the uninitialized implementation contract, this vulnerability is limited to the implementation contract itself.

Recommendations

In order to prevent the invocation of the `initialize` function in the implementation contract below, it is advisable ↗ to include the `_disableInitializers` function within the constructor.

```
constructor() {
        _disableInitializers();
}
```

Remediation

This issue has been acknowledged by Pyth Data Association, and a fix was implemented in commit 49de9a23 ↗.

3.3. Performing duplicate checks within the same range of `update` length

Target	PythLazer.sol		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the `verifyUpdate` function, validation is conducted to prevent excessively short `update` lengths. However, checking the length of duplicate ranges results in unnecessary gas costs.

```
// PythLazer.sol::verifyUpdate()
    function verifyUpdate(
        bytes calldata update
    ) external payable returns (bytes calldata payload, address signer) {
        // [...]
        if (update.length < 71) {
            revert("input too short");
        }
        // [...]
        uint16 payload_len = uint16(bytes2(update[69:71]));
        if (update.length < 71 + payload_len) {
            revert("input too short");
        }
        // [...]
    }
```

Impact

While this does not constitute a security vulnerability, it may lead to users incurring higher gas fees.

Recommendations

It is recommended to remove the following code snippet, which checks for duplicate ranges.

```
// PythLazer.sol::verifyUpdate()
    function verifyUpdate(
        bytes calldata update
    ) external payable returns (bytes calldata payload, address signer) {
        // [...]
        if (update.length < 71) {
            revert("input too short");
        }
        // [...]
    }
```

Remediation

This issue has been acknowledged by Pyth Data Association.

3.4. Inclusion of unnecessary `migrate` function

Target	PythLazer.sol		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `migrate` function includes the ability for administrators to set the `verification_fee` value to `1 wei`. However, since the `verification_fee` value remains unchanged throughout, it is more akin to a constant than a variable. Therefore, the `migrate` function is deemed unnecessary.

```
// PythLazer.sol::migrate()
    function migrate() public onlyOwner {
        verification_fee = 1 wei;
    }
```

Impact

While it is not a security vulnerability, the addition of unnecessary functions may lead to wasted gas costs during deployment and hinder code optimization.

Recommendations

It is recommended to remove the following code snippet.

```
// PythLazer.sol::migrate()
    function migrate() public onlyOwner {
        verification_fee = 1 wei;
    }
```

Remediation

This issue has been acknowledged by Pyth Data Association, and a fix was implemented in commit 49de9a23 ↗.

3.5. Remove unused code

Target	PythLazerLib.sol		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The following line of code imports a library; however, since this library is not utilized within PythLazerLib, the code is unnecessary.

```
// PythLazerLib.sol
import {console} from "forge-std/console.sol";
```

Impact

While it does not represent a security vulnerability, it generally hinders code readability and optimization.

Recommendations

It is recommended to remove the following code snippet.

```
// PythLazerLib.sol
import {console} from "forge-std/console.sol";
```

Remediation

This issue has been acknowledged by Pyth Data Association, and a fix was implemented in commit 49de9a23 ↗.

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: PythLazer.sol

Function: `updateTrustedSigner(address trustedSigner, uint256 expiresAt)`

The function encompasses the ability for the contract's administrator to establish both a trustworthy signer and the duration for which this trust is valid.

Inputs

- `trustedSigner`
  - **Control**: Arbitrary.
  - **Constraints**: It is necessary to convey a value of the `address` type.
  - **Impact**: The conveyed value is appended to the `pubkey` of the `trustedSigners` array.
- `expiresAt`
  - **Control**: Arbitrary.
  - **Constraints**: It is necessary to convey a value of the `uint256` type.
  - **Impact**: The conveyed value is appended to the `expiresAt` of the `trustedSigners` array.

Branches and code coverage

Intended branches

- When a `trustedSigner` value that does not exist in the `trustedSigners` array is provided, the `trustedSigner` value along with the `expiredAt` value is correctly added.
  - ☑ Test coverage
- When the `expiredAt` value is 0, the `trustedSigner` value is successfully removed from the `trustedSigners` array.
  - ☑ Test coverage
- When a `trustedSigner` value that already exists in the `trustedSigners` array is provided, the `expiredAt` value is updated successfully.
  - ☐ Test coverage

Negative behavior

- When the `expiredAt` value is 0 and the `trustedSigner` value does not exist in the `trustedSigners` array, the transaction reverts.
  - ☐ Negative test
- When the `trustedSigners` array is completely filled and an attempt is made to add a new `trustedSigner`, the transaction reverts.
  - ☐ Negative test

Function call analysis

- No external function calls found.

Function: `verifyUpdate(bytes update)`

Based on the extracted signature value from the provided `update`, the function verifies the update of the authenticated signer and extracts the contents of the bytes update to return it alongside the signer.

Inputs

- `update`
  - **Control**: Arbitrary.
  - **Constraints**: The length of the bytes type data must be at least 71 or greater.
  - **Impact**: The `signer` and `payload` are extracted from the bytes data of the provided update for verification, after which they are returned.

Branches and code coverage

Intended branches

- When including the signature of the trusted `signer` and delivering the `payload` in the form of byte data, the values are accurately extracted and returned.
  - ☑ Test coverage
- When Ether exceeding the `verification_fee` value is sent, the remaining Ether is accurately returned to the caller.
  - ☐ Test coverage

Negative behavior

- When the length of the `update` data is less than 71, the transaction reverts.

- ☐ Negative test
- If the first four bytes of the `update` do not match the `EVM_FORMAT_MAGIC` value, the transaction reverts.
  - ☐ Negative test
- If the signer value recovered from the signature is not included in the `trustedSigners` array, the transaction reverts.
  - ☐ Negative test

Function call analysis

- No external function calls found.

5. Assessment Results

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

During our assessment on the scoped Lazer contracts, we discovered five findings. No critical issues were found. Two findings were of low impact and the remaining findings were 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.