Zellic

Pyth2Wormhole

Smart Contract Security Assessment

April 27, 2022

Prepared for:

Pyth Data Foundation

Prepared by:

Filippo Cremonese and Jasraj Bedi

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please email us at hello@zellic.io or contact us on Telegram at https://t.me/zellic_io.

1 Introduction

1.1 About Pyth2Wormhole

Pyth is a first party financial oracle with real-time market data on-chain. It aims to bring valuable financial market data to DeFi applications and the general public. Being native to solana, the prices can only be read by clients on the same network. Pyth2Wormhole leverages the cross-chain arbitrary messaging in Wormhole to bridge the price data to other chains, such as Ethereum and Terra.

1.2 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 open-source 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. We analyze the scoped smart contract code using automated tools to quickly sieve out and catch these "shallow" bugs. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, etc. 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 manually review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents. We also thoroughly examine the specifications and designs themselves for inconsistencies, flaws, and vulnerabilities. This involves use-cases that open the opportunity for abuse, such as flawed tokenomics or share pricing, arbitrage opportunities, etc.

Complex integration risks. Several high-profile exploits have been the result of not any bug within the contract itself, but rather an unintended consequence of its interaction with the broader DeFi ecosystem. We perform a meticulous review of all of the contract's possible external interactions, and summarize the associated risks; for example: flash loan attacks, oracle price manipulation, MEV/sandwich attacks, etc.

Code maturity. We review for possible improvements in the codebase in general. We look for violations of industry best practices and guidelines, or code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, etc.

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; we assign it on a case-by-case basis based on our professional judgment and experience. As one would expect, both the severity and likelihood of an issue affect its impact; for instance, a highly severe issue's impact may be attenuated by a very low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Similarly, Zellic organizes its reports such that the most important findings come first in the document, rather than 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 numerous soft factors, such as our clients' threat model, their business needs, project timelines, etc. We aim to provide useful and actionable advice to our partners that consider their long-term goals, rather than simply a list of security issues at present.

1.3 Scope

The engagement involved a review of the following targets:

Pyth2Wormhole

Repository	https://github.com/pyth-network/pyth2wormhole
Versions	`b41708e8631c1c1f3d0173a8fe717029a93337d0`
Programs	• `solana/pyth2wormhole/program` • `terra/contracts/pyth-bridge` • `ethereum/contracts/pyth`
Type	Rust, Solidity
Platforms	Solana, EVM and Terra

1.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants, for a total of 3 person-weeks. The assessment was conducted over the course of 2 calendar

weeks.

Contact Information

The following project managers were associated with the engagement:

Jasraj Bedi, Co-Founder
jazzy@zellic.io

Stephen Tong, Co-Founder
stephen@zellic.io

The following consultants were engaged to conduct the assessment:

Filippo Cremonese, Engineer
fcremo@zellic.io

Jasraj Bedi, Co-Founder
jazzy@zellic.io

1.5 Project Timeline

The key dates of the engagement are detailed below.

April 4, 2022	Kick-off call
April 18, 2022	Start of primary review period
April 27, 2022	End of primary review period

1.6 Disclaimer

This assessment does not provide any warranties on finding all possible issues within its scope; i.e., the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees on any additional code added to the assessed project after our assessment has concluded. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program. Finally, this assessment report should not be considered financial or investment advice.

2 Executive Summary

Zellic conducted an audit for Pyth Data Foundation from April 18th to April 27th, 2022 on the scoped contracts and discovered 4 findings. Despite the overall good code quality, one critical severity issue was found.

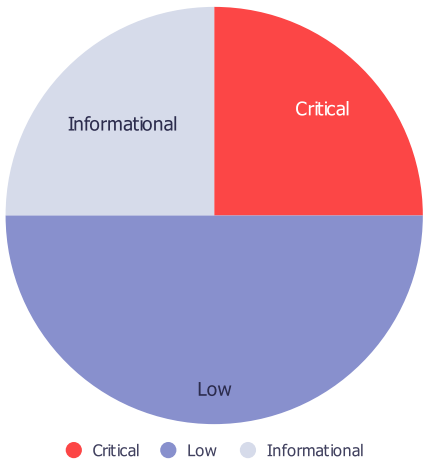
Two of the remaining issues are deemed low severity, and the last finding is reported as informational. Additionally, Zellic recorded its notes and observations from the audit for Pyth Data Foundation's benefit at the end of the document.

Zellic thoroughly reviewed the Pyth2Wormhole codebase to find protocol-breaking bugs as defined by the documentation, or any technical issues outlined in the Methodology section of this document. Specifically, taking into account Pyth2Wormhole's threat model and discussions with the team, the audit was focused heavily on issues that could cause major disruptions on the target chains, such as publishing fake or stale pricing data.

Our general overview of the code is that it was well-organized and structured. The Ethereum and Terra codebases are paired with comprehensive testsuites covering the majority of the functions. The documentation was adequate, although it could be improved to provide more safety notices to end users. The code was easy to comprehend, and in most cases, intuitive.

Breakdown of Finding Impacts

Impact Level	Count
Critical	1
High	0
Medium	0
Low	2
Informational	1



3 Detailed Findings

3.1 Possible usage of stale price information

- **Target**: Solana attester, Ethereum and Terra consumer contracts
- **Category**: Business Logic
- **Severity**: Critical
- **Likelihood**: High
- **Impact**: Critical

Description

The following four separate issues when chained together lead to a critical outcome:

`attest` performs insufficient sanity checks

The `attest` function (from `attest.rs`) of the Solana attester contract does not enforce any restriction on the publication timestamp of the price being attested. Therefore, it could be leveraged to publish out of date pricing information when the prices have not been updated for a while.

Ethereum contract performs insufficient sanity checks

The Ethereum contract consuming price attestations does not perform any sanity check on the price publication timestamp. A last-resort check is performed in `queryPriceFeed` on the price *attestation* timestamp. This check is not particularly effective as the attestation timestamp represents when the attestation program attested the price information through Wormhole, not when the price itself was published.

Terra contract performs insufficient sanity checks

Similar to the ethereum contract, the terra contract does not perform any validation against the price publication timestamp. A check is performed in the `query_price_info` method against the attestation timestamp but as stated previously, it is not sufficient to determine the liveliness of the pricing data, but merely the liveness of the stream of pricing information.

Developer documentation misses important safety notice

The documentation does not recommend the user to check the publication timestamp when retrieving a price, significantly increasing the likelihood of an unsafe usage of the API. In addition, users cannot retrieve publication timestamp from `IPyth` interface but instead have to use `queryPriceFeed`, which is not a part of `IPyth`.

Impact

Stale price accounts can be passed to the attester program and reach Pyth users on other blockchain platforms. After discussion with the Pyth team, this category of publishing stale pricing information is considered critical. Pyth users are unlikely to have implemented sanity checks that prevent them from using outdated information since there's no recommendation to do so in Pyth documentation, and would therefore use the stale data.

Recommendation

Regarding the attester program:

- refuse to attest outdated prices, for instance by checking the `publish_time` field of the `PriceAttestation` struct

Regarding the Ethereum smart contract:

- If possible, add sanity checks on the price publication timestamp by default to all public facing functions
- Otherwise, expand `IPyth` to expose the information required to implement those sanity checks, and clearly document the need for it

Regarding the Terra smart contract:

- Implement sanity checks on the price publication timestamp by default for all public facing functions

Remediation

The finding has been acknowledged by Pyth Data Foundation. Their official response is reproduced below:

> Pyth Data Association acknowledges the finding and developed a patch for this issue:
> https://github.com/pyth-network/pyth2wormhole/pull/194

> https://github.com/pyth-network/pyth2wormhole/pull/196

3.2 `IPyth` interface and implementation do not follow the recommended best practices

- **Target**: Pyth2Wormhole ethereum contract
- **Category**: Code Maturity
- **Severity**: Low
- **Likelihood**: N/A
- **Impact**: Low

Description

The documentation for the `IPyth` public interface suggest the following best practices:

- Use products with at least 3 active publishers
- Check the status of the product
- Use the confidence interval to protect your users from price uncertainty

The first recommendation cannot be followed using only the functions exposed by `IPyth`, and the documentation does not elaborate on what additional functions should be used.

`IPyth` exposes the following three functions:

```
function getCurrentPrice(bytes32 id) external view returns (PythStructs.
    Price memory price);
function getEmaPrice(bytes32 id) external view returns (PythStructs.
    Price memory price);
function getPrevPriceUnsafe(bytes32 id) external view returns (
    PythStructs.Price memory price, uint64 publishTime);
```

`PythStructs.Price` does not contain information about how many publishers contributed to the given price.

A user could still call `queryPriceFeed` (a public function which is not part of `IPyth`). This function returns an instance of `PythStructs.PriceFeed`, a struct that contains fields that can hold the required information.

However, internally the contract does not copy this information from the price attestation.

```
function newPriceInfo(PythInternalStructs.PriceAttestation memory pa)
    private view returns (PythInternalStructs.PriceInfo memory info) {
  info.attestationTime = pa.timestamp;
  // [code shortened for brevity]
```

```
  // These aren't sent in the wire format yet
  info.priceFeed.numPublishers = 0;
  info.priceFeed.maxNumPublishers = 0;
  return info;
}
```

This comment appears to be incorrect with respect to the attestation program reviewed by Zellic. The `attest` function creates instances of the `PriceAttestation` struct using `PriceAttestation::from_pyth_price_bytes`, which does set the `num_publishers` and `max_num_publishers` fields.

Impact

Consumers of Pyth data on Ethereum might not follow the documented best practices and use unreliable price information.

Recommendation

- Modify the `IPyth` interface to provide a way for Pyth users to read how many publishers were aggregated to compute a given price.
- Modify `newPriceInfo` to read from the price attestation the number of publishers that contributed to the price

Remediation

The finding has been acknowledged by Pyth Data Foundation. Their official response is reproduced below:

> Pyth Data Association acknowledges the finding, but doesn't believe it has security implications. However, we may deploy a bug fix to address it.

3.3 Limited test-suite and code coverage

- **Target**: Pyth2Wormhole attester contract
- **Category**: Code Maturity
- **Likelihood**: N/A
- **Severity**: Low
- **Impact**: Low

Description

Pyth Solana attester has only one test for the contract main function, `attest` (located in `pyth2wormhole/client/tests/test_attest.rs`).

Impact

A comprehensive testsuite covering all functionality is very effective in discovering existing bugs and prevent future ones.

Recommendation

We highly recommend Pyth to develop a comprehensive test-suite with maximum code coverage.

Remediation

The finding has been acknowledged by Pyth Data Foundation. Their official response is reproduced below:

> Pyth Data Association acknowledges the finding, but doesn't believe it has security implications. However, we may deploy a bug fix to address it.

3.4 Missing access control on initializer function

- **Target**: Pyth2Wormhole ethereum contract
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Informational
- **Impact**: Informational

Description

Contract `Pyth` (`Pyth.sol`) contains a public initializer function without any modifiers enforcing access control:

```
function initialize(
    address wormhole,
    uint16 pyth2WormholeChainId,
    bytes32 pyth2WormholeEmitter
) virtual public {
  setWormhole(wormhole);
  setPyth2WormholeChainId(pyth2WormholeChainId);
  setPyth2WormholeEmitter(pyth2WormholeEmitter);
}
```

An attacker could call this function and set the Wormhole address, Chain ID and Emitter address to any arbitrary value.

At present, the function cannot be called by an attacker since it is overridden by `PythUpgradable::initialize`, a function with the same prototype that does perform the appropriate access control checks. However, having such a dangerous function exposes Pyth to an unneeded risk of it being inadvertently made reachable by an attacker with a future code refactoring.

Impact

If the function was to be exposed e.g. in a future code refactor, an attacker could call it and trivially take over the contract by setting arbitrary values for the Wormhole and Emitter contracts. This would allow them to submit arbitrary price attestations.

Recommendation

We recommend one of the following remediations:

- Apply `initializer` modifier to `Pyth::initialize`
- Rename `Pyth::initialize` and mark it as private

Remediation

The finding has been acknowledged by Pyth Data Foundation. Their official response is reproduced below:

> Pyth Data Association acknowledges the finding, but doesn't believe it has security implications. However, we may deploy a bug fix to address it.

4 Discussion

The purpose of this section is to document miscellaneous observations the we made during the assessment. Those observations do not have a security impact, and are expressed with the intent of increasing code quality, efficiency and readability.

4.1 Extraneous test condition in terra test code

The `test_verify_vaa_sender_fail_wrong_emitter_address` test in the terra codebase is meant to only test that `verify_vaa_sender` fails if the `emitter_address` is incorrect. This first tests the intended condition, but then tests that `verify_vaa_sender` fails if the `emitter_chain` is incorrect. This is a useful invariant to validate but is already exercised by a separate test `test_verify_vaa_sender_fail_wrong_emitter_chain` already and should be removed from `test_verify_vaa_sender_fail_wrong_emitter_address`.

4.2 `latestPriceInfo` does not check if price ID exists

`latestPriceInfo` (from `PythGetters.sol`) does not check whether the price ID exists in the `_state.latestPriceInfo` map and therefore returns a zero-initialized struct if the key does not exist.

There are two usages of `latestPriceInfo` (both in `Pyth.sol`):

- One usage in `queryPriceFeed`, where the return value is explicitly checked to ensure the price ID was valid
- One usage in `updatePriceBatchFromVm`, where no check is performed

It's unclear whether this edge case of `latestPriceInfo` was considered when writing `updatePriceBatchFromVm`.

```
function updatePriceBatchFromVm(bytes memory encodedVm) public returns (
    PythInternalStructs.BatchPriceAttestation memory bpa) {
    // [shortened for brevity ...]
    PythInternalStructs.BatchPriceAttestation memory batch =
    parseBatchPriceAttestation(vm.payload);

    for (uint i = 0; i < batch.attestations.length; i++) {
        PythInternalStructs.PriceAttestation memory attestation = batch.
    attestations[i];
```

```
        PythInternalStructs.PriceInfo memory latestPrice =
    latestPriceInfo(attestation.priceId);

        if(attestation.timestamp > latestPrice.attestationTime) {
            setLatestPriceInfo(attestation.priceId, newPriceInfo(
    attestation));
        }
    }

    return batch;
}
```

If `latestPrice` is a zero-initialized struct, the `attestationTime` field has zero value, and the function still behaves correctly treating the attestation as new information to be added to the map.

We strongly suggest putting a notice in `latestPriceInfo` documentation, explaining the behaviour of the function and instructing the caller on how to check whether the given price ID existed in the map.

Alternatively, we encourage Pyth to consider implementing a safer getter, for instance returning a tuple `(PriceInfo result, bool error)`. This significantly reduces the likelihood of a caller ignoring the possibility of the price ID not existing in the map due to the additional parameter.

4.3 Extra check in `attest`

Function `attest` (from `attest.rs`) performs a redundant check.

The function expects as input an account structure of type `Attest`:

```
#[derive(FromAccounts, ToInstruction)]
pub struct Attest<'b> {
    // Payer also used for wormhole
    pub payer: Mut<Signer<Info<'b>>>,
    pub system_program: Info<'b>,
    pub config: P2WConfigAccount<'b, { AccountState::Initialized }>,
    // ... more fields
```

The `config` member is a type alias for solitaire `Derive` type:

```
pub type P2WConfigAccount<'b, const IsInitialized: AccountState> =
    Derive<Data<'b, Pyth2WormholeConfig, { IsInitialized }>,
    “pyth2wormhole-config”>;`
```

At the beginning of the function a check is performed to ensure the `config` account public key is a program derived address (PDA).

```
pub fn attest(ctx: &ExecutionContext, accs: &mut Attest, data: AttestData
    ) -> SoliResult<()> {
    accs.config.verify_derivation(ctx.program_id, None)?;
    // ...
```

This check is redundant, as the `Peel` trait for `Derived` already ensures that the account public key corresponds to the expected one.

4.4 Hardcoded magic value

`Pyth.sol` contains two instances of the hardcoded literal `0x50325748`.

We suggest creating a constant literal such as

```
// Comment explaining what this magic value is
uint32 constant PYTH_MAGIC = uint32(bytes4(“P2WH”));
```