Zellic

Pyth Oracle

Smart Contract Security Assessment

April 15, 2022

Prepared for:

Pyth Data Association

Prepared by:

Jasraj Bedi and Stephen Tong

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 to 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 Pyth

Pyth network 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. The network does so by incentivizing market participants (trading firms, market makers, and exchanges) to share the price data collected as part of their existing operations. The network aggregates this first-party price data and publishes it on-chain for use by either on or off-chain applications.

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 a "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:

Pyth Client

Repository	https://github.com/pyth-network/pyth-client
Versions	`2ed8fd07ccc0084129f0c164326140d268e9d922` (`v2` branch)
Programs	program/src/oracle
Type	C
Platform	Solana

1.4 Project Overview

Zellic was approached to perform an 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:

Jasraj Bedi, Co-Founder
jazzy@zellic.io

Stephen Tong, Co-Founder
stephen@zellic.io

Project Timeline

The key dates of the engagement are detailed below.

March 28, 2022	Kick-off call
March 28, 2022	Start of primary review period
April 1, 2022	Weekly progress update
April 8, 2022	End of primary review period
April 11, 2022	Closing call

1.5 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 as financial or investment advice.

2 Executive Summary

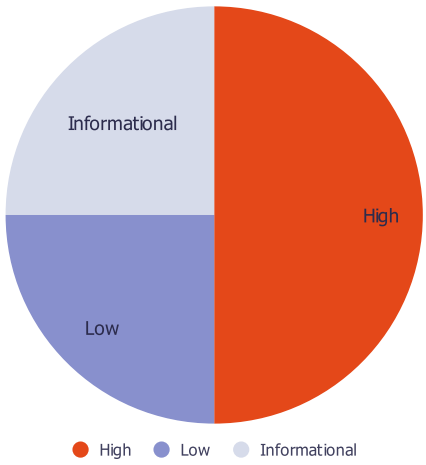
Zellic conducted an audit for Pyth Data Association from 28 March to 8 April 2022 on the scoped contracts and discovered 4 findings. Fortunately, no critical issues were found. We applaud Pyth for their attention to detail and diligence in maintaining high code quality standards. Of the 4 findings, 2 were of high impact, and 1 was of low impact. The remaining findings were informational in nature.

Pyth is an on-chain oracle aggregation network for real-time market data. Interestingly, it is written in C instead of Rust unlike the majority of programs on Solana. Each price product on the network has a list of verified publishers that are allowed to publish the market data. For this audit, we specifically focused on authorization and authentication flaws, both general to solana and specific to Pyth network that might lead to invalid or unauthorized price updates and would be of the highest impact.

Our general assessment of the code is that it is very well-written and maintained. The test suite covers nearly all of the main functionality, and the project has implemented generative fuzz testing for the pd math library. The documentation was clear, concise, and thorough. We hope Pyth continues their commitment to high code quality.

Breakdown of Finding Impacts

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



3 Detailed Findings

3.1 Out-of-bounds write in update test instruction

- **Target**: oracle.c:`upd_test`
- **Severity**: High
- **Impact**: High
- **Category**: Coding Mistakes
- **Likelihood**: High

Description

Pyth exposes two instructions associated with test price accounts - one to initialize them, and another to update the account's pricing information. The update instruction sets pricing, status, and confidence intervals for the test account's price components using a loop that copies over values from the instruction data to the account. This loop iterates a number of times as specified by the caller, incrementing a variable used to index into the price components.

```
for( uint32_t i=0; i != cmd->num_; ++i ) {
    pc_price_comp_t *ptr = &px->comp_[i];
    ptr->latest_.status_ = PC_STATUS_TRADING;
    ptr->latest_.price_  = cmd->price_[i];
    ptr->latest_.conf_   = cmd->conf_[i];
    ptr->latest_.pub_slot_ = slot + (uint64_t)cmd->slot_diff_[i];
}

upd_aggregate( px, slot+1 );
```

The supplied number of iterations this loop should run is not bound in any way. This allows a caller to index past the end of the array, which has a fixed size of 32, and can allow an attacker to manipulate memory outside of the pricing components.

Impact

Memory corruption is a critical violation of program integrity and safety guarantees. The ability to write out-of-bounds can violate the integrity of data structures and invariants in the contract. Thankfully, this instruction validates that only two accounts can be supplied in invocation which does help reduce the impact, but this behavior is still dangerous and may result in an attack that could result in price manipulation.

The `num_` variable is also referenced in `upd_aggregate`, which eventually leads an out-

of-bound stack write that can be potentially leveraged to redirect control flow.

Recommendations

The `upd_test` instruction should validate that `cmd->num_` is equal to or less than `PC_COMP_SIZE`. This will preventing the out-of-bound indexing and write behavior.

Remediation

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

> Pyth Data Association acknowledges the finding and a security fix for this issue will be deployed on or before April 18th.

3.2 Lack of rent exemption enforcement

- **Target**: oracle.c
- **Severity**: High
- **Impact**: High
- **Category**: Business Logic
- **Likelihood**: Low

Description

To support the validators that maintain account state, Solana imposes rent on accounts. Every so often, if an account does not have more than the minimum required lamports to qualify as "rent exempt", an amount of lamports are collected as rent. If an account's balance hits 0, the data for the account is forgotten, effectively resetting the account. Thus, it is possible to *reinitialize accounts which have run out of lamports*.

Pyth uses accounts created and supplied by the caller to store data. Pyth does not require that these accounts maintain a balance large enough to qualify as "rent exempt". This means that a caller can supply an account with too few lamports, initialize it as a particular account type, and, after rent has drained the account, use the account as if it were brand new.

This type of confusion can be found everywhere in the code as rent is not enforced for any accounts supplied by the user.

Impact

The lack of rent exemption checks can result in invariants in the code breaking which can impact clients interacting with the state of these accounts or the contract itself.

For example, product accounts can only be placed into a map if they haven't been initialized yet. This step, using the `add_product` instruction, requires the product account to be initialized but the data field empty. This *should* only be true if the product account has never been used before, but because this account can be wiped out due to rent we can actually add this product account to multiple maps resulting in the product's `prod_` field pointing to an incorrect map.

Recommendations

Pyth should either:

1. Use Program Derived Accounts (PDA) to manage state and delegate signing authority in a way similar to Solana's Token accounts (with an `owner` or `authority` field on the PDA). These accounts should be created with a minimum "rent exempt" qualifying balance.

2. Require all accounts supplied by the user to be rent exempt. It should be sufficient to update both `valid_signable_account` and `valid_writable_account` with this check to get the desired mitigation in place.

Remediation

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

> Pyth Data Association acknowledges the finding and a security fix for this issue will be deployed on or before April 18th.

3.3 Inefficient publisher deletion algorithm results in excessive costs

- **Target**: oracle.c:`del_publisher`
- **Severity**: Low
- **Impact**: Low
- **Category**: Optimization
- **Likelihood**: High

Description

The `del_publisher` instruction allows a caller to remove a publisher from a price account. To do this, the instruction first loops through the publishers on the price account's `comp_` array. After identifying the index of `comp_` with the publisher account, an inner loop runs which shifts all of the accounts down.

```
static uint64_t del_publisher( SolParameters *prm, SolAccountInfo *ka )
{
...
  // try to remove publisher
  for(uint32_t i=0; i != sptr->num_; ++i ) {
    pc_price_comp_t *iptr = &sptr->comp_[i];
    if ( pc_pub_key_equal( &iptr->pub_, &cptr->pub_ ) ) {
      for( unsigned j=i+1; j != sptr->num_; ++j ) {
        pc_price_comp_t *jptr = &sptr->comp_[j];
        iptr[0] = jptr[0];
        iptr = jptr;
      }
      --sptr->num_;
      sol_memset( iptr, 0, sizeof( pc_price_comp_t ) );
      // update size of account
      sptr->size_ = sizeof( pc_price_t ) - sizeof( sptr->comp_ ) +
        sptr->num_ * sizeof( pc_price_comp_t );
      return SUCCESS;
    }
  }
}
```

This is an inefficient way to remove the publisher account from the price account.

Impact

This can result in excessive fees for removing a publisher than would otherwise be necessary. It also increases code complexity, which may leads to bugs in the future.

Recommendations

A more efficient solution would be to replace the publisher account with the last publisher account and then clear out the final publisher entry.

A reference implementation is supplied.

```
for(uint32_t i = 0; i != sptr->num_; ++i ) {
    pc_price_comp_t *iptr = &sptr->comp_[i];
    // identify the targeted publisher entry
    if ( pc_pub_key_equal( &iptr->pub_, &cptr->pub_ ) ) {
        // select the last publisher entry
        pc_price_comp_t *substitute_ptr = &sptr->comp_[sptr->num_ - 1];
        // swap the current publisher entry with the last one - it's okay
    if this is the same entry
        iptr[0] = substitute_ptr[0];
        // clear out the last publisher
        sol_memset(substitute_ptr, 0, sizeof( pc_price_comp_t ));

        // reduce the number of publishers by one
        --sptr->num_;
        // recalculate size
        sptr->size_ = sizeof( pc_price_t ) - sizeof( sptr->comp_ )
            + sptr->num_ * sizeof( pc_price_comp_t );

        return SUCCESS;
    }
}
```

Remediation

The finding has been acknowledged by Pyth Data Association. 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 Unclear variable names can be potentially confusing

- **Target**: Multiple functions
- **Severity**: n/a
- **Impact**: Informational
- **Category**: Code Maturity
- **Likelihood**: n/a

Description

Several functions use short, abbreviated variables names such as `kptr`, `pptr`, `fptr`, and `prm`.

Impact

The finding does not have a direct security impact, but we believe it may lead to future bugs. Although this practice is common in C code, we nevertheless recommend against it. Variable names like these can lead to developer confusion, and ultimately, bugs. These variable names may also lead to the inadvertent misuse of the incorrect variable in place of the correct one. These simple coding mistakes are easy-to-make, hard-to-catch, and often critical in nature. It also makes the code more difficult for auditors and external developers to read, understand, and extend.

Recommendations

Given the security-critical and high-assurance nature of the project, we recommend using longer, clearer, and more specific variable names in the future. For instance, `pc_price_t *pptr` could be instead named `pc_price_t *price_ptr`. Of course, customary variable names such as `i` for loop indices or `n` for count variables are still fine.

Remediation

The finding has been acknowledged by Pyth Data Association. 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

In this section, we discuss miscellaneous interesting observations during the audit that are noteworthy and merit some consideration.

Pyth's test suite includes a battery of fuzz tests. Currently, the fuzz testing covers the `pd` Pyth decimal library. We believe that generative tests, like fuzzing, are an excellent way to improve the security of a project. Thus, we developed a new fuzzing harness for Pyth's sorting algorithm implementation to augment Pyth's test suite.

The fuzzing harness is a straightforward C program which reads test cases from stdin, and calls Pyth's sorting library on it. To best reflect real-world conditions, our harness invokes the sorting library with the same parameters and instantiation as is used in Pyth. The harness tests sorting inputs of variable size and variable values. To validate correctness, the harness compares Pyth sort's results against libc's `qsort` as a ground truth. To validate memory safety, we compiled the harness with clang's AddressSanitizer (ASAN) and MemorySanitizer (MSAN).

We ran the harness in two regimes: (1) "dumb", black-box fuzzing and (2) coverage-guided gray-box fuzzing. For black-box fuzzing, we simply piped `/dev/urandom` into the harness for each execution. We ran the harness under this regime for 12 hours on a 32-core machine. For coverage-guided fuzzing, we built and ran the harness under the AFL++ fuzzer. Under this regime, we ran 16 instances of the fuzzer for 12 hours on a 32-core machine. The results of the fuzz testing did not yield any memory safety or correctness violations.

To facilitate the integration of this new fuzz test into the development lifecycle as part of continuous integration, we also integrated the harness with LLVM libfuzzer.

As part of this engagement, we provided the fuzzing framework to Pyth Data Association, and we recommend integrating fuzz tests with the existing test suite. Fuzzers expand branch, path, and state space coverage and decrease the likelihood of bugs. This is because fuzzers regularly catch corner cases that human programmers fail to consider when writing unit tests. Another benefit of fuzz testing is that developers can stress test business-critical invariants that are specific to the application. This benefits developer confidence, making the development lifecycle both faster and more secure.