Pyth Sui

Audit

Presented by:

OtterSec contact@osec.io

Robert Chen r@osec.io

James Wang james.wang@osec.io

Aleksandre Khokhiashvili khokho@osec.io

Contents

Appendices

01 | Executive Summary

Overview

Pyth Foundation engaged OtterSec to perform an assessment of the Pyth implementation in the `pyth-crosschain` program. This assessment was conducted between December 12th and December 12th, 2022. For more information on our auditing methodology, see Appendix B.

Key Findings

Over the course of this audit engagement, we produced 8 findings total.

In particular, we found issues around missing validations that allow price feed spoofing (OS-PYS-ADV-02), incorrect checks against allowed governance actions (OS-PYS-ADV-00), and the lock of coins due to an incorrect fee recipient (OS-PYS-ADV-01).

We also made recommendations around minimizing the attack surface with friend-only functions (OS-PYS-SUG-00).

Scope

The source code was delivered to us in a git repository at github.com/pyth-network/pyth-crosschain. This audit was performed against commit 178f05c.

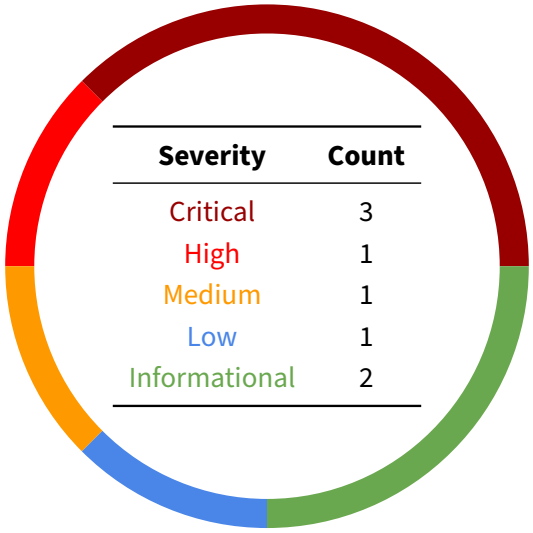
A brief description of the programs is as follows.

Name	Description
pyth-crosschain	Implements a package that allows the verified data source to create `vaa` messages for price attestations. Users and integrators are to submit those `vaa` messages to update and utilize price feeds.

02 | Findings

Overall, we reported 8 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings don't have an immediate impact but will help mitigate future vulnerabilities.



Severity	Count
Critical	3
High	1
Medium	1
Low	1
Informational	2

03 | Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-PYS-ADV-00	Critical	Resolved	`TRANSFER_FEE` not included in the range of allowed governance action.
OS-PYS-ADV-01	Critical	Resolved	`Coins` transferred to a package as a fee will not be collectible.
OS-PYS-ADV-02	Critical	Resolved	`PriceInfoObject` not properly validated in `pyth::get_price` allows attackers to spoof price.
OS-PYS-ADV-03	High	Resolved	The improper design may render upgrades unusable or allow unexpected user interactions during non-atomic upgrade procedures.
OS-PYS-ADV-04	Medium	Resolved	`PriceFeedUpdateEvent` should only be emitted when an actual update occurs.
OS-PYS-ADV-05	Low	Resolved	Governance action sequence numbers should only be allowed to increase monotonically.

OS-PYS-ADV-00 [crit] | Uncallable Governance Action

Description

The `pyth::governance_action` module includes the below function to construct the `GovernanceAction` structure. However, the `assert` check on the value range does not include `TRANSFER_FEE`, making it impossible to perform such an action.

governance/governance_action.move RUST

```
const CONTRACT_UPGRADE: u8 = 0;
const SET_GOVERNANCE_DATA_SOURCE: u8 = 1;
const SET_DATA_SOURCES: u8 = 2;
const SET_UPDATE_FEE: u8 = 3;
const SET_STALE_PRICE_THRESHOLD: u8 = 4;
const TRANSFER_FEE: u8 = 5;

const E_INVALID_GOVERNANCE_ACTION: u64 = 5;

struct GovernanceAction has copy, drop {
    value: u8,
}

public fun from_u8(value: u8): GovernanceAction {
    assert!(
        CONTRACT_UPGRADE <= value && value <= SET_STALE_PRICE_THRESHOLD,
        E_INVALID_GOVERNANCE_ACTION
    );
    GovernanceAction { value }
}
```

Remediation

Fix the `assert` range check to include `TRANSFER_FEE`.

governance/governance_action.move DIFF

```
  public fun from_u8(value: u8): GovernanceAction {
      assert!(
-         CONTRACT_UPGRADE <= value && value <= SET_STALE_PRICE_THRESHOLD,
+         CONTRACT_UPGRADE <= value && value <= TRANSFER_FEE,
          E_INVALID_GOVERNANCE_ACTION
      );
      GovernanceAction { value }
  }
```

Patch

Resolved in 4fe6034.

OS-PYS-ADV-01 [crit] | Invalid Fee Recipient

Description

`Sui` does not yet support retrieving objects owned by other objects. The current implementation of transferring fees for price feed updates to the `pyth` package will result in `Coins` being stuck.

pyth.move RUST

```rust
public fun update_price_feeds(
    worm_state: &WormState,
    pyth_state: &PythState,
    vaas: vector<vector<u8>>,
    price_info_objects: &mut vector<PriceInfoObject>,
    fee: Coin<SUI>,
    clock: &Clock
){
    // Version control.
    state::check_minimum_requirement<UpdatePriceFeeds>(pyth_state);

    // Charge the message update fee
    assert!(
        get_total_update_fee(pyth_state, &vaas) <= coin::value(&fee),
        E_INSUFFICIENT_FEE
    );

    // TODO: use Wormhole fee collector instead of transferring funds to deployer
    ↪  address.
    transfer::public_transfer(fee, @pyth);

    ...
}
```

Remediation

Transfer collected fees to an external receiver address instead.

pyth.move DIFF

```
- transfer::public_transfer(fee, @pyth);
+ transfer::public_transfer(fee, state::get_fee_recipient(pyth_state));
```

Patch

Resolved in 7a286f6.

OS-PYS-ADV-02 [crit] | PriceInfoObject Not Validated

Description

The `pyth::get_price` API takes a `PriceInfoObject` provided by a user, and does not validate whether this object is registered in `pyth::state` while unpacking it.

pyth.move RUST

```
public fun get_price(
    state: &PythState,
    price_info_object: &PriceInfoObject,
    clock: &Clock
): Price {
    get_price_no_older_than(
        price_info_object,
        clock,
        state::get_stale_price_threshold_secs(state)
    )
}
```

Since the `PriceInfoObject` constructer `price_info::new_price_info_object` is a public function, attackers can create an arbitrary `PriceInfoObject` to pass to the integrator and spoof price feeds.

price_info.move RUST

```
public fun new_price_info_object(
    price_info: PriceInfo,
    ctx: &mut TxContext
): PriceInfoObject {
    PriceInfoObject {
        id: object::new(ctx),
        price_info: price_info
    }
}
```

Remediation

Make `price_info::new_price_info_object` a `friend` function only.

Patch

Resolved in fa5aca0.

OS-PYS-ADV-03 [high] | Improper Upgrade Implementation

Description

Pyth's upgrade implementation relies on the `wormhole::vaa` module. Due to the design of `sui` package upgrades, on cases where `wormhole::vaa` is upgraded and `required_version` is raised, an upgrade of `pyth` would no longer be possible since it is linked to the old `wormhole` implementation.

Apart from this, the original implementation of `wormhole`'s `token_bridge` upgrade consists of two steps.

1. Upgrade package through native sui upgrade command.
2. Custom migration logic to facilitate a switch from old modules to new modules.

Due to a circular dependency between `storageId` of newly deployed package and hash of `BatchTransaction` that includes the upgrade transaction, it is generally not possible to call functions on new modules within the same `BatchTransaction` where upgrade command is included.

This limitation forces `wormhole`'s `token_bridge`, `upgrade`, and `migrate` to be performed in two different transactions and exposes a window where other transactions may be called on both new and old packages after `upgrade` occurs but before `migrate` is finished.

Remediation

After discussion with the Pyth Foundation team, split the `vaa` validation part out from the upgrade flow to prevent version-checking issues.

Rework the upgrade flow such that a version toggle that would get activated upon calling `migrate` is added. This ensures that before `migrate` occurs, only functions from the old package with APIs compatible with the pre-migrate state may be called.

Patch

Resolved in d079d4a.

OS-PYS-ADV-04 [med] | Incorrect PriceFeedUpdateEvent Emission

Description

The current implementation emits `PriceFeedUpdateEvent` even for stale price feed update transactions.

pyth.move RUST

```
if (price_info::get_price_identifier(&price_info) ==
    price_info::get_price_identifier(&update)){
    found = true;
    pyth_event::emit_price_feed_update(
        price_feed::from(price_info::get_price_feed(&update)),
        clock::timestamp_ms(clock)/1000
    );

    // Update the price info object with the new updated price info.
    if (is_fresh_update(&update, vector::borrow(price_info_objects, i))){
        price_info::update_price_info_object(
            vector::borrow_mut(price_info_objects, i),
            update
        );
    }
};
```

Remediation

Only emit events when the provided price feed is fresh.

pyth.move DIFF

```
- pyth_event::emit_price_feed_update(
-       price_feed::from(price_info::get_price_feed(&update)),
-       clock::timestamp_ms(clock)/1000
- );

  // Update the price info object with the new updated price info.
  if (is_fresh_update(&update, vector::borrow(price_info_objects, i))){
      ...
+     pyth_event::emit_price_feed_update(
+         price_feed::from(price_info::get_price_feed(&update)),
+         clock::timestamp_ms(clock)/1000
+     );
  }
```

Patch

Resolved in 7a286f6.

OS-PYS-ADV-05 [low] | Unchecked Governance Action Sequence Numbers

Description

Governance actions (`vaas`) should be executed in order. Additionally, whenever an update to the governance data source occurs, a `initial_sequence` would be included in the update message to retroactively disable all stale `vaas`.

The current implementation does not properly

1. Check the sequence numbers when performing governance actions.
2. Apply `initial_sequence` to disable old messages.

Remediation

Check sequence numbers before performing governance actions.

governance/governance.move DIFF

```
  public entry fun execute_governance_instruction(
      ...
  ) {

+     let sequence = governance_message::sequence(&receipt);
+     assert!(
+         sequence > state::get_last_executed_governance_sequence(pyth_state),
+         E_CANNOT_EXECUTE_GOVERNANCE_ACTION_WITH_OLD_SEQUENCE_NUMBER
+     );

+     state::set_last_executed_governance_sequence(
+         &latest_only,
+         pyth_state,
+         sequence
+     );

      // Get the governance action.
      let action = governance_instruction::get_action(&instruction);
      ...
  }
```

Properly handle `initial_sequence` on governance data source updates.

governance/set_governance_data_source.move DIFF

```
  public(friend) fun execute(pyth_state: &mut State, payload: vector<u8>) {
      state::check_minimum_requirement<SetGovernanceDataSource>(pyth_state);

-     // TODO - What is GovernanceDataSource initial_sequence used for?
```

```
        let GovernanceDataSource {
            emitter_chain_id,
            emitter_address,
-           initial_sequence: _initial_sequence
+           initial_sequence: initial_sequence
        } = from_byte_vec(payload);
        state::set_governance_data_source(
            pyth_state,
            data_source::new(emitter_chain_id, emitter_address)
        );
+       state::set_last_executed_governance_sequence(
+           &latest_only,
+           pyth_state,
+           initial_sequence
+       );
    }
```

Patch

Resolved in 0509f0f, 621acb6 and b76e1df.

04 | General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent antipatterns and could lead to security issues in the future.

ID	Description
OS-PYS-SUG-00	Make dangerous functions `friend` only to minimize the potential attack surface.
OS-PYS-SUG-01	Remove unused duplicate functions in the code.

OS-PYS-SUG-00 | Minimize Attack Surface

Description

Several functions for `price_info` and `data_source` modifies `pyth::state`. While no immediate threats are found, those functions could be marked as `friend` only to minimize the attack surface.

Remediation

Mark functions that create persistent objects or modify `pyth::state` friend only.

data_source.move DIFF

```
-  public fun new_data_source_registry(...){
+  public(friend) fun new_data_source_registry(...){

-  public fun add(...) {
+  public(friend) fun add(...) {

-  public fun empty(...) {
+  public(friend) fun empty(...) {

-  public fun new(...): DataSource {
+  public(friend) fun new(...): DataSource {
```

price_info.move DIFF

```
-  public fun new_price_info_registry(...) {
+  public(friend) fun new_price_info_registry(...) {

-  public fun add(...) {
+  public(friend) fun add(...) {

-  public fun new_price_info_object(...) {
+  public(friend) fun new_price_info_object(...) {
```

Patch

Resolved in fa5aca0.

OS-PYS-SUG-01 | Remove Duplicate Function

Description

An unused function `state::register_price_feed`, which is a duplicate of `state::register_price_info_object`, exists in code.

Remediation

Remove the redundant function.

state.move DIFF

```
-  public(friend) fun register_price_feed(
-      s: &mut State,
-      p: PriceIdentifier,
-      id: ID
-  ){
-      price_info::add(&mut s.id, p, id);
-  }
```

Patch

Resolved in 7a286f6.

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

Critical	Vulnerabilities that immediately lead to loss of user funds with minimal preconditions Examples: • Misconfigured authority or access control validation • Improperly designed economic incentives leading to loss of funds
High	Vulnerabilities that could lead to loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions • Exploitation involving high capital requirement with respect to payout
Medium	Vulnerabilities that could lead to denial of service scenarios or degraded usability. Examples: • Malicious input that causes computational limit exhaustion • Forced exceptions in normal user flow
Low	Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants • Improved input validation

B | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the implementation of the program requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of sum, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.