Zellic

Pyth Network

Smart Contract Patch Review

July 24, 2023

Prepared for:

Pyth Data Association

Prepared by:

Filippo Cremonese and Raj Agarwal

Zellic Inc.

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

We conducted a review of a patch to the Pyth Network that introduced accumulators.

Accumulators are a new approach of generating verified prices for consumption in various networks. With accumulators, price updates are included in a Merkle tree, allowing to verify the authenticity of every individual price on chain without the need to verify every price included in the tree. This change increases Pyth scalability and brings gas savings and reliability benefits.

1.1 Scope

The engagement involved a review of the following targets:

Pyth Network

Repository	https://github.com/pyth-network/pyth-crosschain
Versions	https://github.com/pyth-network/pyth-crosschain/pull/904
	https://github.com/pyth-network/pyth-crosschain/pull/931
	https://github.com/pyth-network/pyth-crosschain/pull/932
Type	CosmWasm, Aptos and Sui
Platform	Multi

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Filippo Cremonese, Engineer
fcremo@zellic.io

Raj Agarwal, Engineer
raj@zellic.io

1.2 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 additional code added to the assessed project after the audit version of 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 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.

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.

2 Smart Contract Patch Review

In this engagement we reviewed three pull requests (#904, #931, and #932), which added support for price updates using accumulators on the Aptos, CosmWasm, and Sui platforms, respectively.

Accumulators provide an efficient way to perform a price update by selecting an arbitrary amount of price information from a wide set of (signed and verified) price updates, which can potentially include all the prices published by Pyth. This allows Pyth to provide its users with a single and compact signed VAA that summarizes a wide set of price updates and can be used to publish any number of updates the user needs.

No security vulnerabilities were found during the patch review. For more details, please refer to the discussion section of the report.

3 Discussion

3.1 Merkle trees

Introduction

Accumulators are built using a key cryptographic primitive known as Merkle trees. This brief section provides a basic description, and it is not intended to be exhaustive, as several crucial details, including potential attacks, are not discussed.

Merkle trees provide the ability to compute a unique *root hash* based on the content of the tree. With this hash, it becomes possible to verify whether a specific node belongs to the tree, using a space and time complexity that grows linearly with the node's height, which translates to a logarithmic growth with the total number of nodes if the tree is balanced.

Since arbitrary data can be stored in the leaves of a Merkle tree, it is possible to use this primitive to create a compact attestation covering the entire tree's data, allowing proof for any number of leaves' presence.

Each node in the tree has a corresponding hash; the hash of leaf nodes consists of the hash of the leaves' contents, while the hash of inner nodes is computed on the concatenation of the hashes of its children. The following figure 3.11 shows an example of a simple tree with four leaves:

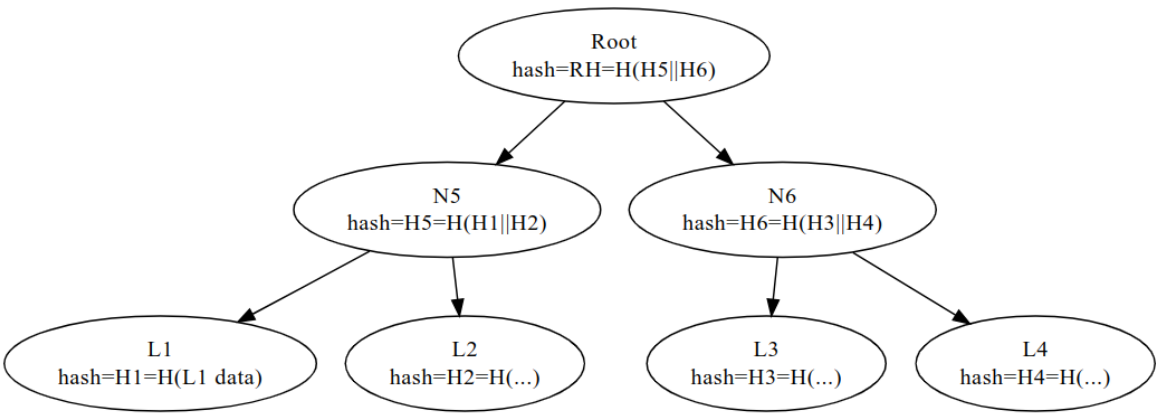

Figure 3.11: A simple merkle tree with four leaves

In order to prove that the data of a given leaf belongs to the tree, the prover needs to provide to the verifier the hashes of the siblings encountered along the path from the leaf node to the root.

For example, to prove the data of the leaf L1 is included in the above tree, the prover needs to supply to the verifier just the hash `H2` (and not the content) of leaf L2 and the hash `H6` of node N6.

In the following figure 3.12, red arrows mark nodes involved in the computation of the root hash, and red labels mark nodes whose hash is required for the proof.

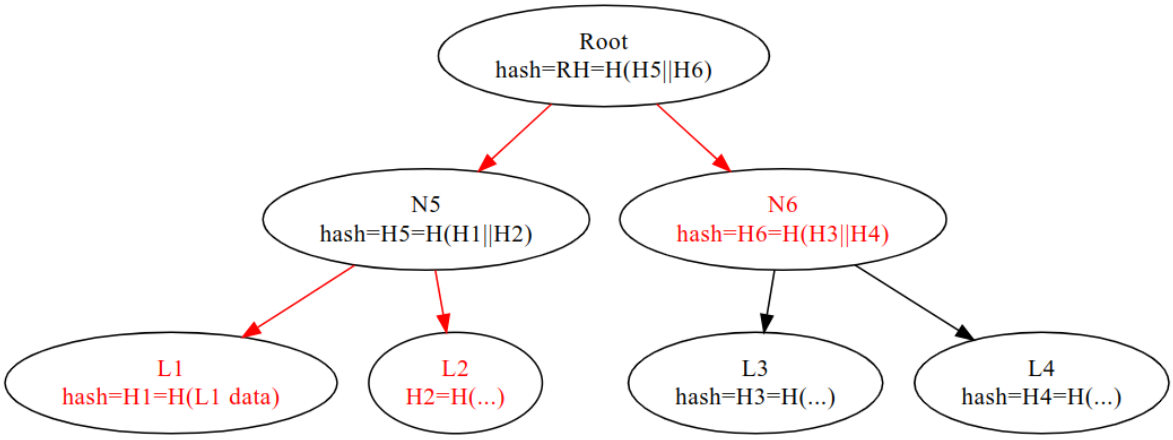

Figure 3.12: Computation of the root hash

To verify the data of L1 belongs to the tree, the verifier computes

- The hash `H1 = H(L1 data)`
- The hash `H5 = H(H1||H2)`
- The hash `RH = H(H5||H6)`

and then verifies that the computed `RH` matches the expected root hash.

Pyth Merkle tree implementation

The Pyth implementation of Merkle trees is straightforward and remains consistent across all three platforms. No issues were identified in any of the implementations. However, there are a few noteworthy aspects that we will discuss in the following subsections.

Hash truncation

We would like to highlight that the Merkle tree hashes are computed using the `keccak256` hash function, resulting in a 32-byte-long output. However, the implementation truncates this output to 20 bytes, reducing the collision resistance from 256 bits to 160 bits. While this truncation diminishes the security level slightly, we consider it adequate and do not see it as a vulnerability. The decision to truncate the hash is likely made to save 12 bytes for each hash included in a Merkle proof.

Implicit child chirality

When constructing the hash of a node, the order in which the hashes of its children are concatenated is significant. The Pyth implementation determines the order of the children by sorting their hashes before concatenating them. This allows to save more space, since the order of the children does not have to be explicitly encoded in the Merkle proof.

Forging protection

An important protection against malleability is implemented by appending a different prefix to the material hashed for leaves and for inner nodes. This prevents attacks where the hash of an inner node could be interpreted as the content of a leaf, with potentially disastrous consequences.

3.2 Aptos code review (PR #904)

PR #904 implements accumulators on the Pyth Aptos codebase. In addition to introducing the new merkle and keccak160 modules, which implement Merkle tree and hashing functionality, relatively straightforward changes were made to the main pyth module to support updating prices using accumulators.

Specifically, `update_price_feed_from_single_vaa` was updated to recognize price updates using accumulators. If an accumulator-enabled update is found (by inspecting the first bytes of the VAA for the magic `PYTHNET_ACCUMULATOR_UPDATE_MAGIC`), then `parse_and_verify_accumulator_message` is invoked, shifting execution to a set of new functions introduced to handle accumulator updates.

We thoroughly examined the newly introduced functions and did not identify any security issues. The VAA source is correctly checked to be trusted, and we we did not find any issue allowing to manipulate accumulator update data to spoof price

information. Type confusion between different VAAs is prevented by checking a magic value in the VAA payload header.

The fee computation routine `get_update_fee` was also updated to account for accumulator updates. The function was changed to recognize an accumulator update, parse it, and count the number of price updates contained in it, to properly compute update fees.

The PR also contains some minor changes to tests unrelated to accumulators, which we did not consider security relevant.

3.3 CosmWasm code review (PR #931)

PR #931 implements accumulators on the Pyth CosmWasm codebase.

The PR introduced the MerkleRoot and Keccak160 modules in the contract. These modules implement the Merkle proof verification and hashing functionality. Additionally, minor changes were made to the codebase to support price updates using accumulators. Note that the contract still allows price updates using batch attestations for legacy support.

The handler for price feed updates (`update_price_feeds`) was updated to handle accumulator based price updates. The price feed updates are parsed by `parse_update`, which uses the VAA data to check if the header is equal to `PYTHNET_ACCUMULATOR_UPDATE_MAGIC` and then uses the accumulator implementation to validate and load the price feed. Otherwise, the VAA data is parsed directly as a batch attestation.

Moreover, the PR introduces updates to the fee computation routine to handle price updates through the accumulator. The `get_update_fee` function now calls `get_update_fee_amount`, which parses the VAA data to compute the fee required for the price update.

Another handler, `parse_price_feed_updates`, was added in the update. It implements functionality similar to `update_price_feeds`. However, the handler is currently not used anywhere in the contract. If the handler is used in a future update, the caller needs to ensure that `min_publish_time` and `max_publish_time` are properly validated. In `update_price_feeds`, this is done through `update_price_feed_if_new`. Otherwise, an attacker could potentially manipulate the oracle by submitting old price feeds.

Other minor changes and tests present in the PR were deemed to not have any security implications. All changes were closely inspected and no other security issues were identified. The program correctly validates the VAAs, ensuring that the submitted price data is correct. The accumulator implementation is sound, and no issues were found that could allow manipulation of the oracle price feed.

3.4 Sui code review (PR #932)

PR #932 implements accumulators on the Pyth Sui codebase.

The merkle_tree and pyth_accumulator modules were introduced. The former implements low-level details of the Merkle tree implementation, such as hashing and Merkle proof verification. The latter implements functionality to parse price updates and verify they belong to a given Merkle tree.

As with PR #904 implementing the equivalent functionality on Aptos, we closely analyzed the codepaths responsible for parsing and verifying VAAs and Merkle proofs, finding no security issues. The VAA source is correctly checked in all instances. Appropriate checks are performed on several fields of the VAA payload, including version checks and magic number checks (preventing type confusion).

Different from the Aptos changes, the Sui codebase introduced separate entry points for working with accumulators and for working with legacy batch price attestations. We second this design choice, as it simplifies testing and code review, limiting the amount of code that is reachable within the code of a single call.

The PR also contains other changes unrelated to accumulators. Excluding minor cleanups, a bug in `pyth::contract_upgrade::deserialize` was addressed, where the function was not stripping leading bytes from a buffer when they should have been ignored. Additionally, the price_info module was updated to allow collecting price update fees and storing them in a dynamic object field owned by the individual price info object being updated. Currently, no functionality is implemented to allow extracting the collected update fees.