Pyth Governance

Audit

Presented by:

OtterSec contact@osec.io

Robert Chen notdeghost@osec.io

Kevin Chow kchow@osec.io

Table of Contents

01 | Executive Summary

Overview

Pyth Data Association engaged OtterSec to perform an assessment of the `pyth-governance` program prior to deployment. This assessment was conducted between April 30th and May 20th, with a focus on the following:

- Ensuring the integrity of programs at an implementation and design level
- Analyzing the program attack surface and providing meaningful recommendations to mitigate future vulnerabilities

All issues were communicated to the team prior to the delivery of the report to speed up remediation. After delivering our audit report, we worked closely with the Pyth Data Association team to streamline patches and confirm remediation.

Key Findings

The following is a summary of the major findings in this audit.

- 8 findings total
- 1 vulnerability which could lead to loss of funds
    - OS-PYG-ADV-00: Max Voter Weight manipulation

We also observed the following:

- The code base quality was high with solid design
- The team was very knowledgeable and responsive to our feedback
- The most serious vulnerabilities were related to `spl-governance`

As part of this audit, we also provided proofs of concept for each vulnerability to prove exploitability and enable simple regression testing. These scripts can be found at https://osec.io/pocs/pyth-governance. For a full list, see Appendix C.

02 | Scope

We received the program from Pyth Data Association and began the audit on April 30. The source code was delivered to us in a git repository at https://github.com/pyth-network/governance. This audit was performed against commit bd7b33.

The scope of the staking implementation was specific to governance. Parts of this program are also used for future features related to product staking for publishers and delegators.

A brief description of the program is as follows. A full list of program files and hashes can be found in Appendix A.

Name	Description
`staking`	The central program for governance of Pyth. The functionality of this program includes: - Staking and creating positions for Pyth tokens for voting power - Warm-up/cooldown rules for unstaking, closing positions, and withdrawal - Managing the voter weight add-in for `spl-governance`

03 | 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 onchain program–that, in other words, there is no way to steal tokens or deny service, ignoring any Solana specific quirks such as account ownership issues. An example of a design vulnerability would be an onchain oracle which could be manipulated by flash loans or large deposits.

On the other hand, auditing the implementation of the program requires a deep understanding of Solana's execution model. Some common implementation vulnerabilities include account ownership issues, arithmetic overflows, and rounding bugs. For a non-exhaustive list of security issues we check for, see Appendix B. We identified an important issue here which would allow a staker to lower the minimum threshold to pass a proposal (OS-PYG-ADV-00).

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 any target in a team of two. This allows us to share thoughts and collaborate, picking up on details that the other missed.

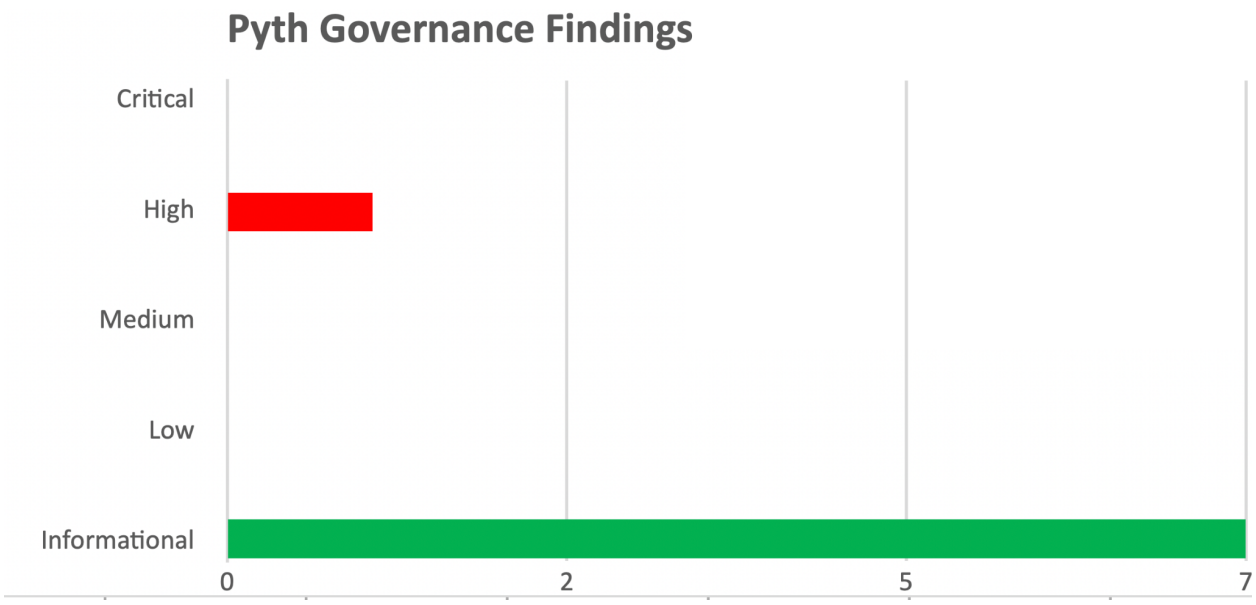
While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.

04 | Findings

Overall, we report 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.

The below chart displays the findings by severity.



Proofs of Concept

For each vulnerability we created a proof of concept to enable easy regression testing. We recommend integrating these as part of a comprehensive test suite. The proof of concept directory structure can be found in Appendix C.

These proofs of concept can be found at https://osec.io/pocs/pyth-governance.

To run a POC:

```
./run.sh <directory name>
```

For example,

```
./run.sh os-pyg-adv-00
```

Each proof of concept comes with its own patch file which modifies the existing test framework to demonstrate the relevant vulnerability. We also recommend integrating these patches into the test suite to prevent regressions.

05 | Vulnerabilities

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

Rating criterion can be found in Appendix D.

ID	Severity	Status	Description
OS-PYG-ADV-00	**High**	**Resolved**	Using the total supply of Pyth mint in voter weight calculation can be easily manipulated

OS-PYG-ADV-00 [High] [Resolved]: Max Voter Weight Manipulation

Description

The maximum voter weight is resolved as the supply of the governing community mint (or the total number of token votes). The minimum vote threshold or minimum votes needed to pass a proposal is $Threshold * Mint\ Supply$.

governance/src/state/proposal.rs

```
let numerator = (yes_vote_threshold_percentage as u128)
        .checked_mul(max_vote_weight as u128)
        .unwrap();
```

In `pyth-governance`, the voter weight add-in defines a staker's vote weight as:

$$\frac{Staker\ Locked\ Tokens}{Total\ Locked\ Tokens} * Mint\ Supply$$

src/utils/voter_weight.rs

```
let voter_weight: u64 = ((raw_voter_weight as u128) * (total_supply
as u128))
        .checked_div(current_locked as u128)
        .unwrap_or(0_u128)
        .try_into()
        .map_err(|_| ErrorCode::GenericOverflow)?;
```

Therefore, the threshold to pass a proposal resolves to:

$$\frac{Staker\ Locked\ Tokens}{Total\ Locked\ Tokens} * Mint\ Supply > Threshold * Mint\ Supply$$

Unfortunately, the mint supply can be manipulated such that it is higher during voting but lower doing proposal settlement and finalization (via burning). The same tokens used to vote can be burned before finalization.

Proof of Concept

Here we assume that an attacker has `33.3%` of token supply and that the other `66.6%` is already locked up–In practice we can relax this assumption. We also assume the threshold to pass a proposal is `50%` of the majority vote.

To replicate the vulnerability, we follow the steps outlined below:

1. Lock up `33`% of token supply [**Now locking**].
2. Wait until the next epoch to lock [**Now locked**]. Now initiate an unlock on this position [**Now pre-unlocking**].
3. Near the end of the pre-unlocking epoch boundary, update voter weight for staker, create proposal and cast vote.
4. Now wait for another epoch [**Unlocking**]. The proposal ends during this epoch, close to the epoch boundary.
5. Now wait until the epoch boundary when we reach **Unlocked**.
6. Withdraw tokens, burn `33%,` and now finalize the vote. The proposal passes as `33/66` is now `50%`.

The amount we need to burn is:

$$Burn\ Amount = Mint\ Supply * (1 - 2 * \frac{Staker\ Locked\ Tokens}{Total\ Locked\ Tokens})$$

While this may seem unrealistic, it only makes sense to burn tokens when the staker has between `33%` and `49%` of the total locked voter weight. At `33%,` as demonstrated in our proof of concept, there is a chance to reach a `50%` majority if the staker can burn `33%` of the total supply; and at `49%, 2%` of the total supply.

Additionally, this would allow an attacker to have whatever authority governance has, along with whatever instructions are added to the proposal (including minting more tokens). An attacker may evaluate a proposal opportunity as net profitable compared to the burn loss.

To create a proposal with transactions and finalize it toward a resolution (Success or Defeated), the proposal must use the deny vote option.

Remediation

We recommend deriving voter weight with a large constant, rather than using mint supply. Note this also requires adjusting the denominator of vote resolution and specifying a max vote weight add-in with the same constant.

voter_weight.rs

```
let voter_weight: u64 = ((raw_voter_weight as u128) * (large_constant
as u128))
        .checked_div(current_locked as u128)
        .unwrap_or(0_u128)
        .try_into()
        .map_err(|_| ErrorCode::GenericOverflow)?;
```

lib.rs

```
    pub fn update_max_voter_weight(ctx:
Context<UpdateMaxVoterWeight>) -> Result<()> {
        let config = &ctx.accounts.config;
        let max_voter_record = &mut ctx.accounts.max_voter_record;

        max_voter_record.realm = config.pyth_governance_realm;
        max_voter_record.governing_token_mint =
config.pyth_token_mint;
        max_voter_record.max_voter_weight = MAX_VOTER_WEIGHT;
        max_voter_record.max_voter_weight_expiry = None; // never
expires
        Ok(())
    }
```

Patch

Pyth Data Association acknowledges the finding and developed a patch for this issue: #170

06 | General Findings

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

ID	Description
OS-PYG-SUG-00	Use of zero-copy with unsupported data types can cause undefined behavior and incompatible layout between different builds with the same compiler.
OS-PYG-SUG-01	Frozen vesting of tokens possible.
OS-PYG-SUG-02	A future update of `spl-governance` may make proposal `max_voting_time` future incompatible.
OS-PYG-SUG-03	For a risk validation check to be functional, the token account must be reloaded after CPI transfer.
OS-PYG-SUG-04	The global config for `pyth-governance` can be instantiated by anyone.
OS-PYG-SUG-05	The proposal account when updating voter weight is not validated.
OS-PYG-SUG-06	For consistency, a check for a zero amount in `close_position` should be added.

OS-PYG-SUG-00 [Resolved]: Unstable Use of Zero-Copy with non-POD Data

Description

The `PositionData` account is deserialized via Anchor's zero-copy mechanism, which requires the structures serialized to have a stable layout, not contain padding, and accept any bit-pattern as a valid value in order to maintain correctness and avoid undefined behavior.

However, `PositionData` contained Rust enums which were not annotated with a specific `repr` -. This includes both the standard library's `Option<T>` on elements of the `positions` array, `unlocking_start`, and the `TargetWithParameters` enum.

src/state/positions.rs

```
pub struct Position {
    pub amount:                u64,
    pub activation_epoch:      u64,
    pub unlocking_start:       Option<u64>,
    pub target_with_parameters: TargetWithParameters,
```

src/state/positions.rs

```
pub enum TargetWithParameters {
    VOTING,
    STAKING {
        product:   Pubkey,
        publisher: Publisher,
    },
}
```

Since these enums are not `repr(C)`, Rust makes absolutely no guarantees about the layout of these structures in memory—and by extension, their layout when serialized into the on-chain account. It is explicitly allowed for these layouts to change between compilations with the same compiler version, which makes them unsuitable for permanent storage. The presence of padding bytes in the structure due to the alignment of types used also leads to uninitialized bytes being exposed in the account data, and can potentially lead to undefined behavior.

Finally, there is no guarantee from Rust that the all-zeros initial state of the account is a valid value for the Rust structure. This could give rise to undefined behavior when initializing the account.

Remediation

Avoid zero-copying any structure that does not fulfill the conditions listed in the safety section of Anchor's zero-copy docs.

We recommend serializing and deserializing the individual `Positions` in a `PositionData` on-demand using a general purpose serializer such as Borsh.

Patch

Pyth Data Association acknowledges the finding and developed a patch for this issue: #164

OS-PYG-SUG-01 [Resolved]: Frozen Vesting Possible

Description

With very large `num_periods` or `period_duration` in periodic vesting, it is possible to specify a schedule that never vests and results in tokens being non-withdrawable.

src/state/vesting.rs

```
let time_passed: u64 = current_time
                .checked_sub(start_date)
                .unwrap()
                .try_into()
                .unwrap();
let periods_passed = time_passed / period_duration;
// Definitely round this one down
if periods_passed >= num_periods {
    0
} else {
```

Remediation

It may also be good practice to limit how far in the future the `num_periods` `period_duration` can be (for example, one year), to prevent tokens from vesting for impractically long periods.

Patch

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

OS-PYG-SUG-02 [Resolved]: Proposal `max_voting_time` Future Incompatibility

Description

The `spl-governance` program has a `max_voting_time` attribute per proposal that exists, but is not implemented yet. It is intended to allow additional optionality to set a `max_voting_time` greater than the `max_voting_time` in the governance that it belongs to. The `pyth-governance` program hinges on `max_voting_time` being less than or equal to one epoch.

governance/src/state/proposal.rs

```
/// Max voting time for the proposal if different from parent
Governance  (only higher value possible)
/// Note: This field is not used in the current version
pub max_voting_time: Option<u32>,
```

Remediation

While this property is already implicitly enforced by only updating voter weight during a proposal's starting epoch and the next epoch, we recommend adding a check to the program such that a proposal's `max_voting_time` cannot be greater than one epoch (such as in `update_voter_weight`), in case this is implemented in the future.

Patch

Pyth Data Association acknowledges the finding and developed a patch for this issue: #182

OS-PYG-SUG-03 [Resolved]: TokenAccount Reload

Description

In the `withdraw_stake` function, risk validation is attempted after a transfer, but the custody token account needs to be reloaded to reflect its true balance.

src/lib.rs

```
transfer(
           CpiContext::from(&*ctx.accounts).with_signer(&[&[
               AUTHORITY_SEED.as_bytes(),
               ctx.accounts.stake_account_positions.key().as_ref(),
               &[stake_account_metadata.authority_bump],
           ]]),
           amount,
       )?;

     if utils::risk::validate(
           stake_account_positions,
           stake_account_custody.amount,
           unvested_balance,
           current_epoch,
           config.unlocking_duration,
       )
       .is_err()
       {
           Return
Err(error!(ErrorCode::InsufficientWithdrawableBalance));
       }
```

Remediation

To make this validation functional, add a reload call after the transfer CPI.

```
ctx.accounts.stake_account_custody.reload()?;
```

Patch

Pyth Data Association acknowledges the finding and developed a patch for this issue: #156

OS-PYG-SUG-04 [Resolved]: Config Instantiation Permission

Description

The global configuration has constant seeds and can only be instantiated once. However, there are no permission restrictions, meaning anyone can instantiate this configuration.

src/context.rs

```
pub struct InitConfig<'info> {
    // Native payer
    #[account(mut)]
    pub payer:          Signer<'info>,
    #[account(
        init,
        seeds = [CONFIG_SEED.as_bytes()],
        bump,
        payer = payer,
        space = global_config::GLOBAL_CONFIG_SIZE
    )]
```

Remediation

We recommend requiring an admin to call this function.

src/context.rs

```
pub struct InitConfig<'info> {
    // Native payer
    #[account(mut, address = admin.key)]
    pub payer:          Signer<'info>,
    #[account(
        init,
        seeds = [CONFIG_SEED.as_bytes()],
        bump,
        payer = payer,
        space = global_config::GLOBAL_CONFIG_SIZE
    )]
```

Patch

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

OS-PYG-SUG-05 [Resolved]: Proposal Account Validation

Description

When calling `update_voter_weight`, a proposal can be passed in as a remaining account, but the account owner is neither checked against the `spl-governance` program ID nor checked for type.

src/lib.rs

```
let proposal_account = &ctx.remaining_accounts[0];
let proposal_data: ProposalV2 =
    try_from_slice_unchecked(&proposal_account.data.borrow())?;
let proposal_start = proposal_data
    .voting_at
    .ok_or_else(|| error!(ErrorCode::ProposalNotActive))?;
```

Remediation

Add a check that runs when `update_voter_weight` is called such that the owner is checked against the `spl-governance` program ID. We also recommend considering a PDA check against the proposal seeds.

src/lib.rs

```
let proposal_account = &ctx.remaining_accounts[0];
let governance_program =
Pubkey::from_str("GovER5Lthms3bLBqWub97yVrMmEogzX7xNjdXpPPCVZw").unwr
ap();
assert_eq!(proposal_account.owner, &governance_program);
let seed = &[
       PROGRAM_AUTHORITY_SEED,
       governance.as_ref(),
       governing_token_mint.as_ref(),
       proposal_index_le_bytes,
    ];

let (proposal_addr, _bump) = Pubkey::find_program_address(seed,
&governance_program);
assert_eq!(proposal_addr, proposal_account.key());
let proposal_data: ProposalV2 =
    try_from_slice_unchecked(&proposal_account.data.borrow())?;
```

```
let proposal_start = proposal_data
    .voting_at
    .ok_or_else(|| error!(ErrorCode::ProposalNotActive))?;
```

governance/src/state/proposal.rs

```
pub fn get_proposal_address_seeds<'a>(
   governance: &'a Pubkey,
   governing_token_mint: &'a Pubkey,
   proposal_index_le_bytes: &'a [u8],
) -> [&'a [u8]; 4] {
   [
       PROGRAM_AUTHORITY_SEED,
       governance.as_ref(),
       governing_token_mint.as_ref(),
       proposal_index_le_bytes,
   ]
}
```

Patch

Pyth Data Association acknowledges the finding and developed a patch for this issue: #181

OS-PYG-SUG-06 [Resolved]: Check Non-zero close_position amount

Description

When `close_position` is called, it does not check that the amount is 0 as it does in `create_position`. This check should be added for consistency.

Remediation

Ensure that `amount` is non-zero by adding a check to `close_position`.

src/lib.rs

```
pub fn close_position(
        ctx: Context<ClosePosition>,
        index: u8,
        amount: u64,
        target_with_parameters: TargetWithParameters,
    ) -> Result<()> {
        if amount == 0 {
            return Err(error!(ErrorCode::ClosePositionWithZero));
        }
```

Patch

Pyth Data Association acknowledges the finding and developed a patch for this issue: #179

07 | Appendix

Appendix A: Program Files

Below are the files in scope for this audit and their corresponding sha256 hashes.

```
staking
  Cargo.toml                          37c01be76924151cfd32baf28847c930
  Cargo.toml                          815f2dfb6197712a703a8e1f75b03c69
  rustfmt.toml                        4583b9ea72b8efa3361cb662387aae23
  src
    constants.rs                      01ba4719c80b6fe911b091a7c05124b6
    context.rs                        58440b133403638fc95825ca4108f560
    error.rs                          9bd51e255bb9c6e00f6dd719d1007e6e
    lib.rs                            299ab45bc62a0b376c4bf3796a3d8163
    wasm.rs                           8e9a59c372515c49770a3e035c04e0be
    state
      global_config.rs                28d19bd10af14da6a8a2eb35124149d3
      mod.rs                          d17a962dec93181dc7f1b4c7274719ec
      positions.rs                    b40c14161145f50f9d47a1a8c5f0443c
      stake_account.rs                e84a54df9ed5cee565400d062fa4fce3
      target.rs                       fbe348f9ce62b2086bac52ba74737cb6
      vesting.rs                      0a520ae5c3b1255ac8e0573d2058a015
      voter_weight_record.rs          15c91ee1550c7d89768fa01dea091ddd
    utils
      clock.rs                        b663687d7a1ee2bbf57d5d23a965e9cd
      mod.rs                          97ce6888ea74b12abf11ff8a5bfdea7f
      risk.rs                         fa83bbb569e32ef3719892244f093f8a
      voter_weight.rs                 a09201b85989560e6952aaf0185577eb
```

Appendix B: Implementation Security Checklist

Unsafe arithmetic

Integer underflows or overflows	Unconstrained input sizes could lead to integer over or underflows, causing potentially unexpected behavior. Ensure that for unchecked arithmetic, all integers are properly bounded.
Rounding	Rounding should always be done against the user to avoid potentially exploitable off-by-one vulnerabilities.
Conversions	Rust `as` conversions can cause truncation if the source value does not fit into the destination type. While this is not undefined behavior, such truncation could still lead to unexpected behavior by the program.

Account Security

Account Ownership	Account ownership should be properly checked to avoid type confusion attacks. For Anchor, the safety of unchecked accounts should be clearly justified and immediately obvious.
Accounts	For non-Anchor programs, the type of the account should be explicitly validated to avoid type confusion attacks.
Signer Checks	Privileged operations should ensure that the operation is signed by the correct accounts.
PDA Seeds	PDA seeds are uniquely chosen to differentiate between different object classes, avoiding collision.

Input Validation

Timestamps	Timestamp inputs should be properly validated against the current clock time. Timestamps which are meant to be in the future should be explicitly validated as such.
Numbers	Reasonable limits should be put on numerical input data to mitigate the risk of unexpected over and underflows. Input data should be constrained to the smallest size type possible, and upcasted for unchecked arithmetic.
Strings	Strings should have reasonable size restrictions to prevent denial of service conditions.

Internal State	If there is internal state, ensure that there is explicit validation on the input account's state before engaging in any state transitions. For example, only open accounts should be eligible for closing.

Miscellaneous

Libraries	Out of date libraries should not include any publicly disclosed vulnerabilities
Clippy	`cargo clippy` is an effective linter to detect potential anti-patterns.

Appendix C: Proofs of Concept

Below are the provided proof of concept files and their sha256 hashes.

```
run.sh                                  904675ba1ca6cf997dcbe5bd87b36b4a
os-pyg-adv-00
  patch                                 169028a36f3ee2f6153cc62d1407ea98
  run.sh                                76efcfb0346537867608423b49bc7c7d
```

Appendix D: 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 which **immediately** lead to loss of user funds with minimal preconditions Examples: - Misconfigured authority/token account validation - Rounding errors on token transfers
High	Vulnerabilities which **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 which could lead to denial of service scenarios or degraded usability. Examples: - Malicious input cause computation limit exhaustion - Forced exceptions preventing normal use
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 - Uncaught Rust errors (vector out of bounds indexing)