Security Audit - Pye Fi

conducted by Neodyme AG

Lead Auditor:	Jasper Slusallek
Second Auditor:	Nico Gründel
Administrative Lead:	Thomas Lambertz

June 05, 2025

Table of Contents

Appendices

1 | Executive Summary

Neodyme audited **Pye Finance's** on-chain liquid staking bonds program from February 2025 until April 2025.

The scope of this audit included both implementation security and a conceptual review including analysis of economic attack vectors.

Despite two prior audits, **8 security-relevant** and **5 informational** issues were found (as measured by Neodyme's Rating Classification). The number of findings identified throughout the audit, grouped by severity, can be seen in Figure 1.

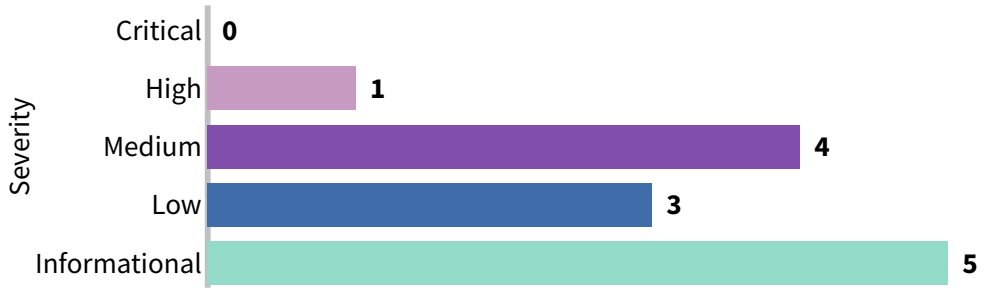

Figure 1: Overview of Findings

The auditors reported all findings to the Pye Finance developers, who addressed them promptly. The security fixes were verified for completeness by Neodyme. In addition to these findings, Neodyme delivered the Pye Finance team a list of eleven nit-picks and additional notes that are not part of this report.

2 | Introduction

From February 2025 until April 2025, Pye Finance engaged Neodyme to do a detailed security analysis of their Pye Bonds Project. Two senior security researchers from Neodyme conducted independent full audits of the contract between the 24th of February 2025 and the 30th of April 2025.

Both auditors have a long track record of finding critical and other high-impact vulnerabilities in Solana programs, as well as in Solana's core code itself. They have audited several other stake pools including Marinade and SPL-Stake, both of which Pye Bonds interacts with. They are also familiar with the intricacies of the Solana stake program and its many edge cases. The latter proved valuable, as it directly lead to finding ND-PYE-MD-01, where a little-known functionality in the stake program could cause bricked instructions in the Pye Bonds program.

Summary of Findings

All findings identified during the auditors were sent to the developers for discussion and remediation. In total, the audit revealed

0 critical • **1** high-severity • **4** medium-severity • **3** low-severity • **5** informational

issues. They are detailed in Section 5.

The auditors found that although some improvements would improve readability of the code, the structure of the code was overall clear and easy to understand. The business logic was kept simple and precise.

3 | Scope

The contract audit's scope comprised of three major components:

- Primarily, the **Implementation** security of the contract's source code
- Additionally, security of the **overall design**
- Additionally, resilience against **economical attacks**

Neodyme considers the source code, located at https://github.com/pyefi/pye-program-library, in scope for this audit. Third-party dependencies are not in scope. The Pye contract relies on the Anchor library, the SPL stake pool, marinade, as well as the standard Metaplex Metadata program, all of which are well-established.

During the audit, minor changes and fixes were made by Pye Finance, which the auditors also reviewed.

Relevant source code revisions are:

- `19d9adad9bd62f5b0aa2c3b180fb081c2443461e` · Start of the audit
- `1048c402a9337494a55f5ab85e5bcd20f171fc71` · Last reviewed revision

4 | Project Overview

This section briefly outlines Pye Fi's functionality, design, and architecture, followed by a detailed discussion of all related authorities.

Functionality

Overview

Pye Fi's protocol enables the separation and trading of staking yield independently from staked SOL. When a user deposits tokens into the system, they receive two (newly minted) assets:

- Principal Tokens (PT), which represent ownership of the underlying staked SOL, and
- Yield Tokens (YT), which represent the future staking rewards.

This works via so-called "bonds", which have fixed issuance and maturity dates. Each bond issues its own unique PT and YT tokens. Users deposit into the bond to receive these PT and YT and can only redeem them for the respective SOL value after the maturity date.

To be more precise, at maturity, the PT and YT can be redeemed together for the total value of the stake: PT corresponding to the initial deposit amount, and YT to the staking rewards accrued by that deposit amount during the bond's duration. If slashing occurs and the total value of the bond at maturity is less than the original deposit, PT holders get back the remaining value while YT holders receive nothing. Post-maturity rewards are distributed proportionally between PT and YT holders, which more or less simulates converting to liquid SOL and staking that until the actual redemption occurs.

Bond Types

There are two primary types of bonds supported by the protocol: Solo Validator Bonds and Liquid Staking Token (LST) Bonds.

Solo Validator Bonds handle SOL staked to a single, fixed validator. Each bond maintains two internal stake accounts:

- a main stake account that holds active stake for most of the bond's duration, and
- a transient stake account that facilitates stake activation or deactivation near the bond's maturity.

These bonds directly use the Solana stake program and handle their own stake accounts. After maturity, they allow redemption either as new stake account containing the SOL, or as immediately accessible liquid SOL.

To make liquid redemption possible, the protocol includes a "global counter party", a program-controlled account which acts as a pooled liquidity provider for all solo validator bonds. This counter party holds liquid SOL and lets users trade in PT and YT to it in exchange for immediately getting liquid

SOL rather than getting an active stake account. Later, the counter party can reclaim deactivated stake from the bonds to replenish its liquidity. The counter party keeps track of its PT and YT holdings over all bonds and triggers the appropriate stake deactivations for this. The instructions for doing this are permissionless.

Note that the protocol periodically sweeps Jito tips from the stake account (received as unstaked SOL) and restakes them. However, a bug in the instruction for depositing stake accounts into a solo validator bond enabled users to steal the Jito tips instead, see ND-PYE-HI-01.

LST Bonds differ from Solo Validator Bonds in that they accept LSTs such as mSOL or SPL stake pool tokens instead of native SOL. The bond specifies which LST it operates on during creation. The yield represented by the YT in this case is the accrual in value of the LST between deposit and maturity.

Currently, only liquid staking tokens from the official SPL stake pool deployment as well as from the single- and multi-validator Sanctum pools are supported.

On deposit, the protocol calculates the token's value in lamports, locking in this value for PT issuance. At maturity, the current value is recalculated, and YT receives the difference. Note that this requires introspecting state accounts of the respective foreign stake-pool program.

PT and YT can be redeemed for LST after maturity. Note that this means that PT will normally be redeemed for less LST than was deposited, as they represent a fixed SOL value, not a fixed LST value. The delta in LST will represent the yield, which will be paid out to the YT.

Fees

Finally, the protocol includes two fee types. A **deposit fee** is applied to LST bonds at time of deposit. Solo validator bonds do not incur this fee, instead they have a **counter party fee** when users opt for liquid redemption through the global counter party rather than waiting for stake deactivation.

The deposit fee is collected in the protocol fee wallet, while counter party fees simply remain with the counter party (whose liquidity would normally be controlled by the same entity).

Additional protocol revenue comes from the fact that SPL stake-pool allows a referrer for deposits, which will get a kickback in LST from the fee charged by the stake pool. Pye sets this referrer to the protocol fee wallet.

On-Chain Data and Accounts

Data-Storing Accounts

The on-chain program needs to keep track of several things:

- **global settings**; this includes the counter party fee and deposit fee, the identity of the global admin, as well as the protocol fee wallet where fees will end up. After our review, this was expanded by the halt admin as well as the halt flags, which are each used to signal whether certain functionality is halted or not.
- **for each bond**:
  - the **issuance, issuance close** and **maturity** timestamp of the bond; deposits should only be possible between the issuance and issuance close timestamps, and the bond will mature and enable payouts after the maturity timestamp
  - a boolean flag indicating **whether maturity has been handled** (i.e. the rewards calculated and the payouts stored in the redemption cache)
  - the **redemption cache**, which is populated only after maturity and stores how much is owed to PT and YT
  - its **PT and YT mint** addresses
  - **reward commission** values defining fees on rewards, though these are currently not implemented.
- additionally, **for solo validator bonds**:
  - **what validator** the bond is for (identified by the validator's vote account)
  - the **main and transient stake account** adresses
  - the **stake of the transient stake account**, for accounting purposes
  - boolean flags indicating whether the **stake account is entirely unstaked** and whether it has been **fully withdrawn**
- additionally, **for LST bonds**:
  - **what stake pool** the bond is for (identified by stake pool's main state account as well as the LST program's address)
  - the program's **vault** address where the deposited LST are held

This data is represented on-chain as follows.

Globally, the protocol has one `GlobalSettings` account, located at the fixed and unique PDA that is seeded by `[b"global_settings"]`. It stores the global fee settings as described above, along with the halt admin and halt flags. They can only be changed by the global admin authority, whose address is also stored in this account.

For each Solo Validator Bond, the protocol has a `SoloValidatorBond` account, whose PDA is seeded by `[b"bond", validator_vote_account_address, issuance_timestamp, maturity_timestamp]`. It stores all the information that is stored for every bond, along with the information specific to Solo Validator Bonds, both as listed above. Its data is not meant to be changed directly by any authority,

and indeed all its values are either static or are accounting or state variables that are programmatically changed under certain conditions.

For each LST Bond, the protocol has an `LSTBond` account where the address is the PDA derived by `[b"bond", stake_pool_state_account_address, issuance_timestamp, maturity_timestamp]`.

Note that because bonds can be created permissionlessly, and because there can only be one bond for each underlying validator/stake pool and issuance-maturity combination, issues would arise if bonds could be misconfigured. One such issue was uncovered during the audit with ND-PYE-LO-01, where an attacker could misconfigure bond by supplying misleading or malicious token metadata information.

Other PDAs

Apart from the data-storing accounts in the last section, the following accounts are used by the program:

- For each bond, the **token mint accounts for PT and YT** are located at the PDAs derived via the seeds `[b"principal"]` and `[b"yield"]`, respectively.
- The **global counter party account**, which is at the PDA seeded by `[b"counter_party"]`. It holds the counter party's liquid SOL funds.
    - Additionally, **the ATAs of the counter party** for each PT and YT mint are used to store the PT and YT that users redeem with the counter party for its liquid SOL.
- For each LST Bond, the **lst vault** is located at the ATA of the bond account for the bond's LST mint (and for the LST's token program). It holds the LST that users deposit into the bond.
- For each Solo Validator Bond, the **main stake account** is created at the PDA seeded by `[b"stake"]`. It is the stake account that usually holds all activated stake of the bond.
- Additionally, each Solo Validator Bond has a **transient stake account** for activating stake before maturity or deactivating it afterwards. This is not located at a PDA, rather the account used for it is supplied by the user and its location then stored in the bond account and checked on every usage.
- Finally, the global settings account references an external **protocol fee wallet**, which is where protocol fees are sent to. Specifically, PT and YT fees are sent to its ATAs on each deposit into a Solo Validator Bond. Furthermore, its LST ATA is also passed as a referrer when depositing via an SPL stake-pool deployment. Fees for Solo Validator Bonds, which are charged at redemption via the counter party, instead simply remain in the global counter party.

Instructions

The contract has a total of 25 instructions, which we briefly summarize here.

We divide them into categories based on which component it is associated with. Specifically, we group them into LST Bond instructions, Solo Validator Bond instructions, and global instructions.

Global Instructions

Instruction	Category	Summary
InitGlobalSettings	Permissionless	Initializes the global settings account and populates it with the supplied data. Fee parameters are verified. The account is unique due to a fixed seed, hence this will fail if called again later.
UpdateFeeSettings	Global Admin Only	Updates the fees settings in the global settings account. Parameters are verified to be within the hard-coded allowed range.
UpdateHaltAdmin	Global Admin Only	Updates the halt admin.
UpdateHaltStatus	Halt Admin Only	Updates the halt flags in the global settings account, enabling or disabling the corresponding functionality.
NativeTransfer	Permissionless	Does a native SOL transfer. Used for self-CPI in other instructions, makes logs clearer.

LST Bond Instructions

Instruction	Category	Summary
InitializeStakePoolBond	Permissionless	Initializes an LST bond account and populates it with the supplied data. The account is unique for each combination of underlying stake pool, issuance date and maturity date. Note that this, along with the fact that this instruction is permissionless, allows attackers to DOS specific “canonical” combinations via misconfigured bond accounts. See finding ND-PYE-LO-01.
Deposit	User	Simply transfers LST from the user to the bond’s vault and mints PT and YT according to the current lamport value of those LST (as calculated via reading the current rate from the external LST contract)

Instruction	Category	Summary
DepositSol	User	Similar to deposit, but the deposit is routed through an spl-stake-pool deployment. That is, the user makes SOL available which is deposited into the stake pool, and the resulting LST are then transferred to the bond's vault, thus continuing as in Deposit. Note that spl-stake-pool allows a referrer, which in this case is set to an ATA of the protocol fee wallet – a cut of the LST fee will be sent there, generating protocol revenue.
DepositSolMsol	User	Similar to DepositSol, but instead of an spl-stake-pool deployment, the deposit is routed through marinade. The resulting mSOL go into the bond's vault.
RedeemPt	User (any PT holder)	Burns the given PT and calculates their lamport value. LST with an equivalent lamport value are then sent to the user from the bond's vault. Can only be called after maturity and after the redemption cache of the bond is set.
RedeemYt	User (any YT holder)	Very similar to RedeemPT, namely: Burns the given YT and calculates their lamport value. LST with an equivalent lamport value are then sent to the user from the bond's vault. Can only be called after maturity and after the redemption cache of the bond is set.

Instruction	Category	Summary
LstBondHandleMaturity	Permissionless	Handles the maturity of the bond (assuming the maturity timestamp has passed). Specifically, sets the redemption conversion cache if it isn't set already. This calculates the current lamport worth of all LST in the vault and calculates the amount of LST that should go to the PT. Normally, this is the number of LST such that their current lamport worth is exactly the total lamport worth of all deposits (i.e. how much PT put in), and all remaining LST go to the YT. If slashing occured – meaning the total lamport worth now is less than the sum of deposits – this is instead distributed evenly among PT, and YT receive nothing.

Solo Validator Bond Instructions

Instruction	Category	Summary
InitializeSoloValidatorBond	Permissionless	Initializes an LST bond account and populates it with the supplied data. The account is unique for each combination of underlying stake pool, issuance date and maturity date. As with InitializeStakePoolBond this leads to a low-severity DoS, see finding ND-PYE-LO-01.
SoloValidatorDepositSol	User	Takes the SOL deposit from the user and puts it into the bond's two-stake-account stake system. Mints a corresponding amount of PT and YT. The calculation of YT assumes that the deposited stake is activated in the next epoch, which is true in almost all cases.

Instruction	Category	Summary
SoloValidatorDepositStake	User	Takes an active stake account and merges it into the bond's main stake account. Mints PT and YT corresponding to the amount of delegated lamports on the stake account. This assumes that merging is possible, which, if the main stake account is already activated, means that the deposited stake account must be delegated to the bond's corresponding validator. Any undelegated lamports on the deposited stake account are refunded to the user after the stake accounts are merged and are therefore not treated as deposited for the purposes of calculating PT and YT amounts.
SoloValidatorDelegateTips	Permissionless	Sweeps any undelegated lamports from the bond's main stake account and puts them into the bond's two-stake-account stake system. These undelegated lamports are most commonly Jito Tips, hence this instruction is an essential function for being able to compound these additional rewards. Cannot be called after the bond's maturity has been handled (Auditor's note: if this were possible, it would create a DOS vector.)

Instruction	Category	Summary
SoloValidatorHandleMaturity	Permissionless	Handles the maturity of the bond (assuming the maturity timestamp has passed). Specifically, sets the redemption conversion cache if it isn't set already. This calculates the total lamports currently owned by the bond in its stake system as well as its liquid reserve. If the total is less than the sum of deposits, PT get everything while YT receive nothing. Otherwise, PT receive exactly their deposit back while the additional rewards are split between all YT.
SoloValidatorRedeemPtForSol	User (any PT holder)	Sends the given PT to the global counter party and pays out the corresponding SOL from the global counter party's SOL reserve, minus the counter party fee. Can obviously only be called after the bond's maturity has been handled. Also updates internal accounting to make sure future payouts are also correctly handled.
SoloValidatorRedeemYtForSol	User (any YT holder)	Same as SoloValidatorRedeemPtForSol, but for YT instead of PT.
SoloValidatorRedeemPtForStake	User (any PT holder)	Same as SoloValidatorRedeemPtForSol, but the user is paid out via a stake account that is split of from the main bond stake account instead of via the global counterparty.
SoloValidatorRedeemYtForStake	User (any YT holder)	Same as SoloValidatorRedeemPtForStake, but for YT instead of PT.

Instruction	Category	Summary
CounterPartyRedeemPt	Permissionless	Burns a specified amount of the corresponding bond's PT in the global counter party's possesion and pays the global counter party from the bond's liquid reserve. Note that UpdateLiquidReserve keeps the bond's liquid reserve liquid enough to fulfill these requests. See that instruction's description below.
CounterPartyRedeemYt	Permissionless	Same as CounterPartyRedeemYT, but for YT instead of PT
UpdateLiquidReserve	Permissionless	Calculates the amount of lamports owed to the global counter party if it were to redeem all its PT and YT. Then calculate how much SOL is still needed in the bond's liquid reserve in order to fulfill those redeem operations. That amount is then unstaked from the bond's stake system. Only callable when bond maturity has been handled.
CounterPartyWithdrawSol	Global Admin Only	Withdraws SOL from the global counter party's account to the global admin's account. Note that the SOL held in the global counter party account is both liquidity for liquid redeems as well as fees that have been generated by said liquid redeems in any of the solo validator bonds. The global admin has authority over all these funds.

Authority Structure

The contract has three privileged authorities or authority types: The upgrade authority, the global admin and the halt admin. Each of these entities has different powers, and the potential effects of each of them being abused are very different.

In this section, we discuss what powers each of the authorities has, and what the worst-case effect of a compromise of each of them would be.

Upgrade Authority

As with any contract, the upgrade authority has complete control over the program and the funds it controls. This includes any funds in PDAs, stake accounts, or simply token accounts controlled by the contract. A compromise is therefore catastrophic.

The upgrade authority is hence one of the most critical components of the security of the protocol. By maliciously upgrading the contract, the upgrade authority can irreversibly transfer control of all funds in the global counter party, in all of the bonds' staking systems and all of the LST vaults. It cannot touch the funds in the protocol fee authority, as that is controlled by the developers, not the contract.

Global Admin

The global admin has control over the fee parameters, including both the amount of fees collected and where any collected fees are sent. More importantly, it controls all funds currently in the global counter party's possession, as the global admin can call CounterPartyWithdrawSol.

A compromise of this authority would thus mean that all the liquidity provided by the protocol would be drained. This does not, however, affect user funds which are stored in the LST or the stake accounts of the individual bonds.

Finally, note that the global admin also controls the identity of the halt authority, and thus that a compromise of the global admin also means that the halt admin should be viewed as compromised.

Halt Admin

The halt admin has only one power: to halt the contract, or to re-enable it. It is set by the global admin.

Obviously, its compromise would open the contract up to a denial-of-service. However, there should be no further effects.

5 | Findings

This section outlines all of our findings.

They are classified into one of five severity levels, detailed in Appendix C. In addition to these findings, Neodyme delivered the Pye Finance team a list of nit-picks and additional notes which are not part of this report.

All findings are listed in Table 4 and further described in the following sections.

Identifier	Name	Severity	Status
ND-PYE-HI-01	Jito tips can be stolen	HIGH	Fixed
ND-PYE-MD-01	Delinquency-triggered stake deactivation is not handled and will lead to bricked instructions	MEDIUM	Fixed
ND-PYE-MD-02	Incorrect token program usage means some users will be unable to redeem	MEDIUM	Fixed
ND-PYE-MD-03	SoloValidator maturity handling can be DoSed, locking funds	MEDIUM	Fixed
ND-PYE-MD-04	Linear YT calculation skews payout function, making later deposits more profitable	MEDIUM	Acknowledged
ND-PYE-LO-01	Canonical bonds can be DoSed via misconfiguration	LOW	Acknowledged
ND-PYE-LO-02	Epoch end date estimation can be significantly skewed	LOW	Fixed
ND-PYE-LO-03	Issuing is possible before `bond.issuance_ts`	LOW	Fixed
ND-PYE-IN-01	No global pause functionality	INFORMATIONAL	Fixed
ND-PYE-IN-02	Incorrect calculation in UpdateLiquidReserve	INFORMATIONAL	Fixed
ND-PYE-IN-03	Bond domains should be separated better	INFORMATIONAL	Acknowledged
ND-PYE-IN-04	DepositSolMsol can be called with a non-marinade LST bond	INFORMATIONAL	Fixed
ND-PYE-IN-05	Minimum delegation is not 1 SOL	INFORMATIONAL	Fixed

Table 4: Findings

[ND-PYE-HI-01] Jito tips can be stolen

Severity	Impact	Affected Component	Status
HIGH	Limited Loss of Funds	Stake Management in Solo Validator Bonds	Fixed

Description

Jito tips are sent to stake accounts as additional unstaked SOL. Normally, for solo validator bonds, they will be swept up via the `DelegateTips` instructions, which withdraws any excess lamports and stakes them.

However, there is a way for the user to obtain the jito tips deposited to the bond's main stake account instead. Let us look at `DepositStake`. It does the following:

1. Transfers the authorities of the to-be-deposited stake account from the user (attacker) to the bond
2. Merges the stake account into the bond's main stake account
3. Withdraws any excess lamports on the main stake account **to the user** [1]
4. Continues with the usual deposit logic by calculating and minting PT/YT amounts

Note that step 3 was most likely inserted to return the stake account rent to the user. However, it also sends any previously undelegated lamports on the main stake account account to the user. This includes all jito tips paid to the main stake account.

The attacker can thus steal all jito tips paid to the bond – likely a significant amount – as long as they deposit any stake account between the jito payout and the call to `DelegateTips`. This is likely a very relaxed window of time.

To mitigate this, we would suggest storing the amount of undelegated lamports on the main stake account before the merge and only paying the user the difference.

Location

[1] https://github.com/pyefi/pye-program-library/blob/1048c402a9337494a55f5ab85e5bcd20f171fc71/programs/bonds/src/instructions/solo_validator/deposit_stake.rs#L241

Relevant Code

```
let post_bond_stake_lamports = ctx.accounts.stake_account.lamports();
    // Handle withdrawing any extra SOL that may have been transferred over
let excess_lamports = post_bond_stake_lamports
    .checked_sub(bond_stake_state.delegation.stake)
    .and_then(|amount| amount.checked_sub(bond_stake_meta.rent_exempt_reserve))
    .ok_or(ErrorCode::ArithmeticOverflow)?;
if excess_lamports > 0 {
```

```
    stake_withdraw(
        ctx.accounts.stake_account.to_account_info(),
        ctx.accounts.bond.to_account_info(),
        ctx.accounts.owner.to_account_info(),
        ctx.accounts.clock.to_account_info(),
        ctx.accounts.stake_history.to_account_info(),
        Some(&[solo_validator_bond_signer_seeds!(ctx.accounts.bond)]),
        excess_lamports,
    )?;
}
```

Remediation

The Pye Fi developers remedied this by changing the logic such that only the increase in unstaked lamports in the merge target is refunded to the user. The fix was authored in PR 70 and verified for correctness by the auditors.

[ND-PYE-MD-01] Delinquency-triggered stake deactivation is not handled and will lead to bricked instructions

Severity	Impact	Affected Component	Status
MEDIUM	Accidental DoS	Stake Management in Solo Validator Bonds	Fixed

Description

The stake program has a little-known permissionless instruction which allows anyone to deactivate a stake account that is staked to a validator that has been delinquent for at least 5 epochs [1]

This becomes relevant for solo validator bonds because the contract never expects to handle a main stake account that is `Inactive` (i.e. no effective stake, no activating stake and no deactivating stake), and will error.

In particular, the `stake_sol!` instruction used in deposits and to delegate tips errors immediately if either the main or the transient stake account is `Inactive` [2]. Hence all deposits and all re-staking of rewards is bricked.

Additionally, if the counter party has enough PT and YT to trigger full deactivation of the main stake account, the contract will error, as it will try deactivating an already deactivated stake account [3].

Locations

[1] https://github.com/solana-program/stake/blob/bcec951fda5f2a30b1f4a058706d2e9ed23a8429/program/src/processor.rs#L982

[2] https://github.com/pyefi/pye-program-library/blob/19d9adad9bd62f5b0aa2c3b180fb081c2443461e/programs/bonds/src/macros.rs#L412

[3] https://github.com/solana-program/stake/blob/bcec951fda5f2a30b1f4a058706d2e9ed23a8429/interface/src/state.rs#L987

Relevant Code

```
macro_rules! stake_sol {
    ($ctx:expr, $staked_amount:expr, $transient_stake_account:expr, $owner:expr) => {

        let clock = Clock::get()?;
        let bond_stake_status = get_stake_status(
            &$ctx.accounts.stake_account,
            clock.epoch,
            &$ctx.accounts.stake_history,
        )?;
```

```
10          msg!("bss {:?}", bond_stake_status);
11          match bond_stake_status {
12              StakeStatus::Empty => { /* ... */ }
13              StakeStatus::Activating => { /* ... */ }
14              StakeStatus::FullyActive => { /* ... */ }
15               // TODO: REVIEW: Is there an edge case where the stake is forced
    unstaked on a cluster reset and it's in an inactive state?
16              _ => return Err(ErrorCode::UnsupportedStakeState.into()),
17          }
18      };
19  }
```

Remediation

The Pye Fi team remediated this by adding handlers for the case that the stake account is inactive in PR 72.

The auditors remarked that there were some inconsistencies between different cases after the fix, which were subsequently ironed out in PR 87. The final changes were fully reviewed by the auditors as well.

[ND-PYE-MD-02] Incorrect token program usage means some users will be unable to redeem

Severity	Impact	Affected Component	Status
MEDIUM	Accidental DoS Locking User Funds	Redemption in LST Bonds	Fixed

Description

In RedeemPT [1] and RedeemYT [2] for LST bonds, in the `transfer_lst_to_owner` helper function, you use the wrong token program to send the LST. You use `ctx.token_program`, when it should be `ctx.lst_token_program`.

Impact: If the token programs for PT/YT and LSTs differ, redeeming PT will be impossible. This breaks intended functionality of the contract.

Note that the LST bonds' Deposit instruction also uses the incorrect token program for its LST transfer [3]. This means that depositing LSTs will be impossible for differing LST and PT/YT programs.

Locations

[1] https://github.com/pyefi/pye-program-library/blob/1048c402a9337494a55f5ab85e5bcd20f171fc71/programs/bonds/src/instructions/redeem_pt.rs#L87

[2] https://github.com/pyefi/pye-program-library/blob/1048c402a9337494a55f5ab85e5bcd20f171fc71/programs/bonds/src/instructions/redeem_yt.rs#L97

[3] https://github.com/pyefi/pye-program-library/blob/1048c402a9337494a55f5ab85e5bcd20f171fc71/programs/bonds/src/instructions/deposit.rs#L31

Relevant Code

Affected File: path/to/file

```
pub fn transfer_lst_to_owner(&self, amount: u64) -> Result<()> {
    let cpi_accounts = /*...*/
    let cpi_ctx = CpiContext {
        accounts: /* ... */,
        remaining_accounts: /* ... */,
        program: self.token_program.to_account_info(), // <--- !
        signer_seeds: /* ... */,
    };
    token_interface::transfer_checked(cpi_ctx, amount, self.lst_mint.decimals)
}
```

Remediation

The token program used for the transfer was corrected in PR 73. Neodyme verified the fix.

[ND-PYE-MD-03] SoloValidator maturity handling can be DoSed, locking funds

Severity	Impact	Affected Component	Status
MEDIUM	DoS, Locking Funds	Maturity Handling of Solo Validator Bonds	Fixed

Description

The `SoloValidatorHandleMaturity` instruction will error if the transient stake account isn't currently `Empty` or `FullyActive` [1]. In particular, it will error if the stake account is currently activating.

This can be abused by an attacker by keeping the transient stake account in an activating state. Directly after the epoch boundary, they can deposit 1 lamport into the main stake account themselves. They can then call `DelegateTips`, which sweeps the undelegated lamport(s) from the main stake account and re-stakes them using the transient stake account. This will merge the old transient stake account that is now `FullyActivated` into the main account and then delegate all the excess lamports with a new, now-activating transient stake account.

This way, the transient stake account will always be activating and HandleMaturity can never be called. The bond maintainer would have to manage to front-run the attacker's `DelegateTips` call to resolve this situation.

Note that `DelegateTips` cannot be called after maturity is handled, but this is not a problem since we are DoSing just that.

Location

[1] https://github.com/pyefi/pye-program-library/blob/1048c402a9337494a55f5ab85e5bcd20f171fc71/programs/bonds/src/instructions/solo_validator/handle_maturity.rs#L77

Relevant Code

```
// Handle existing transient account
match transient_stake_status {
    StakeStatus::Empty => { /* ... */ }
    StakeStatus::FullyActive => { /* ... */ }
    _ => return Err(ErrorCode::UnsupportedStakeState.into()),
}
```

Remediation

The Pye Fi team addressed this by disallowing DelegateTips after the bond's maturity in PR 74. Neodyme verified the fix.

[ND-PYE-MD-04] Linear YT calculation skews payout function, making later deposits more profitable

Severity	Impact	Affected Component	Status
MEDIUM	Misaligned incentives	Core Business Logic	Acknowledged

Description

Generally, the idea of the protocol is that YT represent a share of the staking rewards of the underlying SOL. Users who deposit later should receive proportionally less YT, since the tokens they deposited generate less rewards.

However, this decrease in YT payout is modeled as a linear function. For a deposit of x SOL, the user always receives:

- X PT [1] and
- $x * \frac{\text{bond.maturity\_ts} - \text{now}}{\text{bond.maturity\_ts} - \text{bond.issuance\_ts}}$ YT [2]

This linear formula does not track compound interest. That, in turn, has a measurable effect.

To see this, let's go through a simple example. We create a bond that goes from t=0 to t=200 (we use epochs here for simplicity, so this bond would run a bit over a year). We assume staking rewards per epoch of 0.044%, which would result in 8.3% APY.

We have two depositors, A and B.

A deposits 1000 SOL at t=0. They hence receive

- 1000 PT and
- 1000 YT.

Meanwhile, B stakes their SOL natively until t=100 and only then deposits it into Pye. They have accumulated `1.00044^100 * 1000 ~ 1045 SOL` and hence receive

- 1045 PT and
- 1045 / 2 = 522.5 YT

Note that we are ignoring the fact that we may need to unstake the SOL for one epoch to deposit them, however this is both negligible and we can ignore this by depositing LST instead of native SOL, eliminating the unstaking period.

We now wait until t=200. The total rewards that the protocol accumulated are `(1.00044^200 - 1) * 1000 + (1.00044^100 - 1) * 1045 ~ 138.96`.

Both A and B now trade in their PT and YT. We have that:

- A receives 1000 SOL for their PT and `1000/1522.5 * 138.96 = 91.27` SOL for their YT.
- B receives 1045 SOL for their PT and `522.5/1522.5 * 138.96 = 47.66` SOL for their YT.

Hence A received a total of 1091.27 SOL, while B received a total of 1092.66 SOL. The return of simply natively staking would have been `1.00044^200 * 1000 = 1091.97 SOL`. Hence, in effect, B has "stolen" 0.7 SOL from A's rewards.

In general, the formulas work out as follows when abstracting the total bond duration `M` in epochs, along with the epoch `D` at which B deposits:

- After D epochs, B has accumulated `f(D) = 1.00044^D * 1000 SOL`
- After M epochs, the total rewards accumulated by the protocol are `r(M,D) = (1.00044^M - 1) * 1000 + (1.00044^(M-D) - 1) f(D)`
- When redeeming right after maturity, A receives $1000 + \frac{1000}{1000+f(D)*\frac{M-D}{M}} * r(M, D)$
- B receives $f(D) + \frac{f(D)*\frac{M-D}{M}}{1000+f(D)*\frac{M-D}{M}} * r(M, D)$

It is obvious that the difference gets exponentially worse as M grows. For example, for an M=2000 epoch bond (around 11 years) and D=1300, A receives 2265 SOL at the end, while B receives 2556 SOL, meaning B has "stolen" 145.5 SOL from A.

Location

[1] https://github.com/pyefi/pye-program-library/blob/1048c402a9337494a55f5ab85e5bcd20f171fc71/programs/bonds/src/state/solo_validator_bond.rs#L249

[2] https://github.com/pyefi/pye-program-library/blob/1048c402a9337494a55f5ab85e5bcd20f171fc71/programs/bonds/src/instructions/solo_validator/redeem_yt_for_sol.rs#L108

Mitigation Suggestion

The protocol should be modified to either integrate compounding interest into its calculation, or bond duration should be restricted to total durations where the effect of compounding interest is negligible.

Remediation

The Pye Fi developers stated that the addition of `issuance_close` date – which was added during the audit – mitigates this issue. They also stated that the planned issuance window for canonical bonds was around 1 week or less, which makes the compounding factor is negligible.

[ND-PYE-LO-01] Canonical bonds can be DoSed via misconfiguration

Severity	Impact	Affected Component	Status
LOW	DoS of bond creation	Bond Creation	Acknowledged

Description

For LST bonds, there can only be one bond per tuple of the form (stake_pool_account, issuance_ts, maturity_ts) [1]. The same is true for solo validators when replacing the first entry by the validator vote account [2].

It is likely that there will be periodical bonds issued at fixed time intervals. By interpolating the relevant timestamps of the next bonds issued for a stake pool or validator, an attacker can DoS the not-yet-created bonds by misconfiguring them.

Note that while most configuration data is defaulted to canonical values, the attacker still has certain freedoms in misconfiguring the bond.

As a particular example, the attacker can choose which token program to use for PT and YT. By for example choosing token22 when the old token program is expected, they can break interoperability of the tokens with many other protocols.

Additionally, the attacker can choose the metadata arguments, meaning token name, token symbol and URI.

Note that the metadata update authority is the bond account and that there are no IXs that expose updating it. However if there were, who would have the authority to do so? The bond creator is the attacker, so the functionality to update it would have to be set to a protocol authority.

Location

[1] https://github.com/pyefi/pye-program-library/blob/1048c402a9337494a55f5ab85e5bcd20f171fc71/programs/bonds/src/instructions/initialize_stake_pool_bond.rs#L24

[2] https://github.com/pyefi/pye-program-library/blob/1048c402a9337494a55f5ab85e5bcd20f171fc71/programs/bonds/src/instructions/initialize_solo_validator_bond.rs#L22

Remediation

The developers stated the following in this respect:

```
Because issuance_ts and maturity_ts are both part of the seeds, [the] adversary would have to create all combinations of issuance and maturity for a given issuance date and maturity date to block it. The stakeholders have no problem shifting issuance and maturity dates. So I don't think changing the seeds at this moment is worth while giving the break to backwards compatibility, but we'll look further.
```

The auditors understand this reasoning. The developers also changed the code such that the bond creator is now stored in the bond account, which makes filtering for canonical bonds easier. This was done in PR 83.

[ND-PYE-LO-02] Epoch end date estimation can be significantly skewed

Severity	Impact	Affected Component	Status
LOW	Incorrect YT payout	Business Logic in Solo Validator Bonds	Fixed

Description

Consider the function `est_next_epoch_start_ts`. It estimates the timestamp of the next epoch boundary based on either:

- a constant 400ms per slot if less than 10 slots have passed this epoch
- the average ms per slot in this epoch so far if at least 10 slots have passed

The result of the function is used to calculate when deposits into a solo validator bond start accumulating rewards, i.e. to account for the activation period.

However, this approach is flawed for several reasons:

- Slot length is not uniform, especially around epoch boundaries when additional calculations for reward distribution and other things takes place (although it should be noted that this discrepancy in slot length has recently gotten smaller)
- The chain may experience difficulties or even go down, meaning no slots will be produced while the timestamp will still advance. This would mean that if the chain goes down for, say, 11 hours in the 10th slot and the function is called in the 11th slot, the function will estimate an average of one hour per slot. This would mean that the protocol assumes that the stake will only be activated in `slots_per_epoch * 1h = 432000h`, or about 50 years.
- Validators can subtly manipulate the time per slot in their leader slots in order to either have their own stake accumulate rewards quicker or to troll other depositors.

As a mitigation, we would recommend bounding the result of the function both above and below by agreeable values.

Remediation

The Pye Fi developers remediated this by clamping acceptable slot time values to a range of 350 to 550 in PR 75. This follows the suggested remediation.

[ND-PYE-LO-03] Issuing is possible before `bond.issuance_ts`

Severity	Impact	Affected Component	Status
LOW	Incorrect Functionality	Deposits	Fixed

Description

For both solo validator and LST bonds, `issuance_ts` is not restricted in their initialization instruction. It can be set to a date far into the future, yet deposits will still be allowed before `issuance_ts`.

Note that deposits before `issuance_ts` lead to counterintuitive amounts of YT, since the are calculated with a factor of `(bond.maturity_ts - now) / (bond.maturity_ts - bond.issuance_ts)`, which will be greater than 1. While not leading to exploitable behavior, this is highly counterintuitive.

We believe the intended functionality was – or indeed that the correct behavior should be – to enable deposits only after `bond.issuance_ts`.

Remediation

The developers fixed this with PR 76.

[ND-PYE-IN-01] No global pause functionality

Severity	Impact	Affected Component	Status
INFORMATIONAL	Missing Exploit Mitigation	All	Fixed

Description

Currently, the contract has no functionality to pause all operations.

This may of particular interest if an exploit is discovered that affects the global counter party, as it may hold significant funds. Having a pause functionality would enable the maintainer to suspend operations until a fix is developed and an upgrade to the contract is done.

Remediation

The developers added a halt authority and the corresponding instruction to halt or re-enable the program in PR 83. The changes were reviewed by Neodyme.

[ND-PYE-IN-02] Incorrect calculation in UpdateLiquidReserve

Severity	Impact	Affected Component	Status
INFORMATIONAL	Inaccurate accounting	Business Logic of Solo Validator Bonds	Fixed

Description

In UpdateLiquidReserve, the calculation of `source_stake_after_split` is incorrect. This variable is supposed to hold the size of the delegation after `lamports_to_unstake` lamports are split into a new stake account. Currently however, the calculation is done as:

```
// handle case where this may error because the bond stake account (aka source stake
//  account), ends up below the minimum delegation amount if unstaked. At this point, the
//  entire bond stake account should be unstaked and moved to the liquid reserve.
let stake_account_lamports = ctx.accounts.stake_account.lamports();
let source_stake_after_split = stake_account_lamports
    .saturating_sub(lamports_to_unstake)
    .saturating_sub(stake_rent);
```

However, `stake_account_lamports` may also contain unstaked excess lamports.

This variable is later used to determined whether the main stake account has any delegation remaining. If so, `bond.completely_unstaked` is set to true.

Hence, if `lamports_to_unstake` is exactly the size of the delegation but there are more than `minimum_stake_account_amt` excess lamports on the stake account, everything will be split into the transient stake account, so the main account is entirely empty. However, `completely_unstaked` incorrectly remains as false.

Location

https://github.com/pyefi/pye-program-library/blob/1048c402a9337494a55f5ab85e5bcd20f171fc71/programs/bonds/src/instructions/counter_party/update_liquid_reserve.rs#L212

Remediation

The developers remedied this by changing `source_stake_after_split` in such a way that it only calculates the difference in delegated stake. This was done in PR 77. Neodyme reviewed the fix.

[ND-PYE-IN-03] Bond domains should be separated better

Severity	Impact	Affected Component	Status
INFORMATIONAL	Negligible	PDA Seeds	Acknowledged

Description

We would recommend separating the seed domain of solo validator bonds from that of LST bonds. Right now, both prefixes are `b"bond"` followed by data fields that are equal in structure.

Though a collision would require a fair amount of effort and only DoS the creation of a very specific bond, separating the domains requires no effort and prevents this from happening. It is also cleaner design.

Remediation

The developer stated that while they agree with the rationale on cleaner design, they do not see a risk of collision since `SoloValidatorBond` requires the `validator_vote_account` in the seed while `StakePoolBond` requires `stake_pool` in the seed, and there's validation on these accounts in the initialization.

The auditors agree that a collision here would be difficult to impossible to achieve, and that a collision also gives negligible to no gain to the attacker.

[ND-PYE-IN-04] DepositSolMsol can be called with a non-marinade LST bond

Severity	Impact	Affected Component	Status
INFORMATIONAL	Negligible	Deposit in LST Bonds	Fixed

Description

In `DepositSolMsol`, note that there is no check that the bond that was passed into this instruction is actually an LST bond tied to marinade.

We can call this instruction with a bond that is tied to a different LST program, along with that other LST program as `marinade_program`. The contract will attempt calling `Deposit` on it, which will fail unless the LST program is upgraded to include such an instruction with the according argument and account layout.

Remediation

This was remedied in PRs 78 and 87. Neodyme verified the fix.

[ND-PYE-IN-05] Minimum delegation is not 1 SOL

Severity	Impact	Affected Component	Status
INFORMATIONAL	Unnecessary restriction due to incorrect assumption	Redemption and Deposit in Solo Validator Bonds	Fixed

Description

In RedeemYtForStake, a minimum withdrawal of 1 SOL is enforced [1]. However, this is not enforced in RedeemPtForSol. Indeed, there does not appear to be a reason for this restriction.

Similarly, here [2] 1 SOL minimum stake is enforced while arguing that this is because the stake program does not allow smaller stakes.

This is incorrect. There is currently no limit as to how small a delegation can be. A Solana feature gate to introduce a 1 SOL minimum has existed for some time [3] but has not been activated [4], nor is it likely to be in the near future [5].

Location

[1] https://github.com/pyefi/pye-program-library/blob/1048c402a9337494a55f5ab85e5bcd20f171fc71/programs/bonds/src/instructions/solo_validator/redeem_yt_for_stake.rs#L152

[2] https://github.com/pyefi/pye-program-library/blob/1048c402a9337494a55f5ab85e5bcd20f171fc71/programs/bonds/src/instructions/solo_validator/deposit_sol.rs#L145

[3] https://github.com/anza-xyz/solana-sdk/blob/0f9989d0f2f6aa5328124084e20f587446bda79b/feature-set/src/lib.rs#L396

[4] https://explorer.solana.com/address/9onWzzvCzNC2jfhxxeqRgs5q7nFAAKpCUvkj6T6GJK9i

[5] https://github.com/anza-xyz/agave/pull/5754#discussion_r2039843254

Remediation

The developers remedied this by using the network's advertised minimum stake, instead of the hardcoded 1 SOL. This fix was done in PR 80.

A | About Neodyme

Security is difficult.

To understand and break complex things, you need a certain type of people. People who thrive in complexity, who love to play around with code, and who don't stop exploring until they fully understand every aspect of it. That's us.

Our team never outsources audits. Having found over 80 High or Critical bugs in Solana's core code itself, we believe that Neodyme hosts the most qualified auditors for Solana programs. We've also found and disclosed critical vulnerabilities in many of Solana's top projects and have responsibly disclosed issues that could have resulted in the theft of over $10B in TVL on the Solana blockchain.

All of our team members have a background in competitive hacking. During such hacking competitions, called CTFs, we competed and collaborated while finding vulnerabilities, breaking encryption, reverse engineering complicated algorithms, and much more. Through the years, many of our team members have won national and international hacking competitions, and keep ranking highly among some of the hardest CTF events worldwide. In 2020, some of our members started experimenting with validators and became active members in the early Solana community. With the prospect of an interesting technical challenge and bug bounties, they quickly encouraged others from our CTF team to look for security issues in Solana. The result was so successful that after reporting several bugs, in 2021, the Solana Foundation contracted us for source code auditing. As a result, Neodyme was born.

B | Methodology

We are not checklist auditors.

In fact, we pride ourselves on that. We adapt our approach to each audit, investing considerable time into understanding the program upfront and exploring its expected behavior, edge cases, invariants, and ways in which the latter could be violated.

We use our uniquely deep knowledge of Solana internals, and our years-long experience in auditing Solana programs to find bugs that others miss. We often extend our audit to cover off-chain components in order to see how users could be tricked or the contract affected by bugs in those components.

Nonetheless, we also have a list of common vulnerability classes, which we always exhaustively look for. We provide a sample of this list here.

Select Common Vulnerabilities

Our most common findings are still specific to Solana itself. Among these are vulnerabilities such as the ones listed below:

- Insufficient validation, such as:
  - Missing ownership checks
  - Missing signer checks
  - Signed invocation of unverified programs
  - Account confusions
  - Missing freeze authority checks
  - Insufficient SPL account verification
  - Dangerous user-controlled bumps
  - Insufficient Anchor account linkage
- Account reinitialization vulnerabilities
- Account creation DoS
- Redeployment with cross-instance confusion
- Missing rent exemption assertion
- Casting truncation
- Arithmetic over- or underflows
- Numerical precision and rounding errors
- Anchor pitfalls, such as accounts not being reloaded
- Non-unique seeds
- Issues arising from CPI recursion
- Log truncation vulnerabilities
- Vulnerabilities specific to integration of Token Extensions, for example unexpected external token hook calls

Apart from such Solana-specific findings, some of the most common vulnerabilities relate to the general logical structure of the contract. Specifically, such findings may be:

- Errors in business logic
- Mismatches between contract logic and project specifications
- General denial-of-service attacks
- Sybil attacks
- Incorrect usage of on-chain randomness
- Contract-specific low-level vulnerabilities, such as incorrect account memory management
- Vulnerability to economic attacks
- Allowing front-running or sandwiching attacks

Miscellaneous other findings are also routinely checked for, among them:

- Unsafe design decisions that might lead to vulnerabilities being introduced in the future
    - Additionally, any findings related to code consistency and cleanliness
- Rug pull mechanisms or hidden backdoors

Often, we also examine the authority structure of a contract, investigating their security as well as the impact on contract operations should they be compromised.

Over the years, we have found hundreds of high and critical severity findings, many of which are highly nontrivial and do not fall into the strict categories above. This is why our approach has always gone way beyond simply checking for common vulnerabilities. We believe that the only way to truly secure a program is a deep and tailored exploration that covers all aspects of a program, from small low-level bugs to complex logical vulnerabilities.

C | Vulnerability Severity Rating

We use the following guideline to classify the severity of vulnerabilities. Note that we assess each vulnerability on an individual basis and may deviate from these guidelines in cases where it is well-founded. In such cases, we always provide an explanation.

Severity	Description
CRITICAL	Vulnerabilities that will likely cause loss of funds. An attacker can trigger them with little or no preparation, or they are expected to happen accidentally. Effects are difficult to undo after they are detected.
HIGH	Bugs that can be used to set up loss of funds in a more limited capacity, or to render the contract unusable.
MEDIUM	Bugs that do not cause direct loss of funds but that may lead to other exploitable mechanisms, or that could be exploited to render the contract partially unusable.
LOW	Bugs that do not have a significant immediate impact and could be fixed easily after detection.
INFORMATIONAL	Bugs or inconsistencies that have little to no security impact, but are still noteworthy.

Additionally, we often provide the client with a list of nit-picks, i.e. findings whose severity lies below Informational. In general, these findings are not part of the report.

Neodyme AG

Dirnismaning 55
Halle 13
85748 Garching
Germany

E-Mail: contact@neodyme.io

https://neodyme.io