Security Audit - P-Token

conducted by Neodyme AG

Lead Auditor:	Nico Gründel
Second Auditor:	Sebastian Fritsch

June 12, 2025

Table of Contents

1 | Executive Summary

Neodyme audited **Anza's** new implementation of the Solana token program, called P-Token, from April 2025 until June 2025. Part of this audit was a review of the new smart contract library Pinocchio, which is a more efficient replacement of the Solana Program library.

As P-Token is supposed to be a drop-in replacement for the current token program, Neodyme also considered whether edge-case behaviour is identical, including whether error codes are produced with the right priority.

According to Neodymes Rating Classification, **0 security relevant** and **0 informational** were found. The number of findings identified throughout the audit, grouped by severity, can be seen in Figure 1.

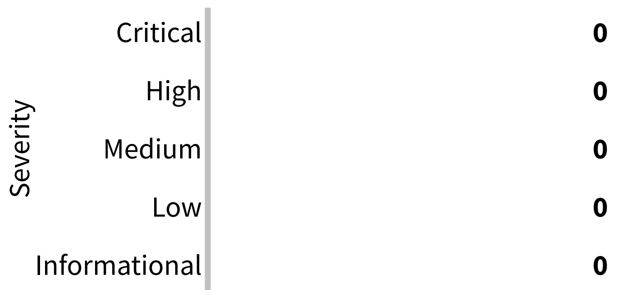


Figure 1: Overview of Findings

Neodyme delivered the Anza team a list of nit-picks and additional notes that are not part of this report. Furthermore, Neodyme implemented a new fuzzcase into Riverguard. This fuzzcase, running over multiple months, has not spotted any divergence of behaviour in any transaction executed on mainnet when replacing the old token program with P-Token. In addition, it has allowed us to precisely measure the impact of replacing SPL-Token with P-Token. We will explore this in Section 6.

2 | Introduction

From April 2025 until June 2025, Anza engaged Neodyme to do a detailed security analysis of Pinocchio and P-Token. Two senior security researchers from Neodyme conducted independent full audits of both the library and the smart contract between the 23th of April 2025 and the 12th of June 2025. Both auditors have a long track record of finding critical and other vulnerabilities in Solana programs, as well as in Solana's core code itself. Both have audited and reviewed the original token program multiple times. As the longest-standing auditing firm in the Solana ecosystem, Neodyme has helped to uncover and fix multiple footguns both in the token program and the Solana program library over the years, supporting the development of both intimately.

The audit focused on the usability of Pinocchio as a replacement for the Solana program library from a security perspective, and on ensuring the behaviour of P-Token is indistinguishable from the current token program. We present our work in the following sections.

Neodyme would like to emphasise the high quality of Anza's work. A lot of effort has gone into making sure that P-Token behaves identically to the token program, including a fuzzing harness, which found all of the issues that Neodyme flagged during the audit.

Summary of Findings

In total, the audit revealed:

0 critical • **0** high-severity • **0** medium-severity • **0** low-severity • **0** informational

issues.

3 | Scope

The contract audit's scope comprised two major components:

- Primarily, making sure that P-Token behaves exactly like the old Solana token program
- Additionally, evaluating Pinocchio for security footguns

Neodyme considers the source code, located at https://github.com/anza-xyz/pinocchio and https://github.com/solana-program/token, in scope for this audit. Third-party dependencies are not in scope.

Relevant source code revisions are:

- `4c3fabd685e30b27bbc8f5d515626177dd720a02` · P-Token
- `837535f244a8f995e11266a52471a5cd3e3e548f` · Pinocchio

4 | P-Token Overview

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

Functionality

The old token program, and now P-Token, provides the backbone of the entire Solana DeFi ecosystem. All token balances, movements, mints, burns, etc. are managed by this smart contract.

Tokens are stored in token accounts, which are controlled by an authority (owner). The type of token is represented by a mint, which itself is represented by its own account. Tokens can be minted by the relevant authority if set, they can be burned, and they can be transferred. Token accounts can be frozen by a freeze authority, which prevents tokens from being moved.

The owner of a token accounts can delegate tokens to a seperate authority, allowing that authority to move at most the delegated amount from the account. The delegation can be revoked at any time by the owner.

In addition, a simple m of n multisig implementation is provided. All authorities (including the delegate), both on the token account and the mint, can also be one of these multisig accounts instead.

In addition to the features of the old token program, P-Token provides a `batch` instruction, which allows to batch multiple token operations into a single instruction, as well as the `WithdrawExcessLamports` instruction, that allows authorities to withdraw accounts above the rent-excempt minimum from all account types.

On-Chain Data and Accounts

There are three types of accounts: `Mint`, representing a token type, `Account`, representing an account that holds tokens and `Multisig`, which can be used in place of any of the authorities. The on-chain representation of these accounts is identical to the old version of the Solana token program. The types are distinguished by the length of the accounts, which are always fixed for each type and distinct from each other.

`Mint` stores an optional mint authority, the total supply of the token, the number of decimals the token has, a flag whether the account is intialized, and an optional freeze authority.

`Account` stores the following data:

- The associated mint
- The owner
- The amount of the token that is held in the account
- Optionally, the delegate and delegated amount

- The accounts state, which can be either `Initialized` or `Frozen` (or `Uninitialized` in case the account hasn't been initialized yet)
- A flag that indicates whether a token account represents native SOL
- The amount of SOL that is required as the rent-exempt reserve when this is a native account
- An optional close authority

`Multisig` stores the amount of signers required for a valid signatures, a flag whether the account is initialized and the list of signers that are part of this multisig (at most 11).

Authorities

Mints have 2 different authorities, both are optional:

- The mint authority is able to mint arbitrary amounts of new tokens. If the mint authority is not set, no new tokens will be able to be minted from this account. Only the mint authority can update or clear this field.
- The freeze authority is able to freeze and thaw any token account associated with this mint. If this authority isn't set, token accounts associated with this mint can't be frozen or thawed. Only the freeze authority can update or clear this field.

Accounts have 3 different authorities:

- The owner has the ability to transfer, burn, close the account, delegate funds or change any authorities on a token account. The owner can only be changed by the owner itself.
- The close authority has the ability to close the account. All funds are transferred to an account of the close authorities choosing, including any SOL currently on the account. The close authority can only be changed by the owner.
- The delegate is an optional authority that is allowed to burn or transfer whatever amount the owner has delegated to it. Both the delegate and the delegated amount can only be changed by the owner.

Instructions

The contract has a total of 27 instructions, which we briefly summarize here.

Instruction	Category	Summary
InitializeAccount	Permissionless	Initializes a new `Account`. Takes the owner and rent sysvar as an account
InitializeAccount2	Permissionless	Identical to `InitializeAccount`, but parses the pubkey of the owner from the instruction data

Instruction	Category	Summary
InitializeAccount3	Permissionless	Identical to `InitializeAccount`, but parses the pubkey of the owner from the instruction data and doesn't require the rent sysvar as a passed account
InitializeMint	Permissionless	Initializes a new `Mint`. Takes the rent sysvar as an account
InitializeMint2	Permissionless	Identical to `InitializeAccount`, but doesn't require the rent sysvar as a passed account
InitializeMultisig	Permissionless	Initializes a new `Multisig`. Takes the rent sysvar as an account
InitializeMultisig2	Permissionless	Identical to `InitializeAccount`, but doesn't require the rent sysvar as a passed account
MintTo	Mint authority	Mints a certain amount of tokens to a token account belonging to this mint
MintTo2	Mint authority	Identical to `MintTo`, but checks that the mint has the expected amount of decimals
FreezeAccount	Freeze authority	Freezes a token account, preventing tokens from being transfered or burned
ThawAccount	Freeze authority	Thaws a previously frozen token account
Approve	Owner only	Sets the delegate and delegate amount on a token account
ApproveChecked	Owner only	Identical to `Approve`, but checks that the mint has the expected amount of decimals
Revoke	Owner only	Clears the delegate and delegated amount
CloseAccount	Owner or close authority	Closes a token account and transfers all tokens and SOL on the account to a different account
Transfer	Owner or delegate	Transfers tokens to a different account associated with the same mint
TransferChecked	Owner or delegate	Identical to `Transfer`, but checks that the mint has the expected amount of decimals
Burn	Owner or delegate	Burns a certain amount of tokens

Instruction	Category	Summary
BurnChecked	Owner or delegate	Identical to `Burn`, but checks that the mint has the expected amount of decimals
SyncNative	Permissionless	Syncs a native token account with the underlying SOL balance of the token account
WithdrawExcessLamports	Authority only	Withdraws any SOL above the rent exempt threshold from a token account, a mint or a multisig account. For token accounts the owner has to sign, for mints the mint authority has to sign (or if there is no mint authority, alternatively the mint account itself can sign for this operation), and for multisig m out of the n authorities have to sign. This is unique to P-Token
SetAuthority	Authority only	Can be used to set the mint or freeze authorities on a mint or the owner or close authority on an account
InitializeImmutableOwner	Permissionless	Noop that only fails if the first passed account is not an initialized token account. Only there to provide the same interface as token2022
AmountToUiAmount	Permissionless	Takes an amount of tokens and formats it corresponding to the amount of decimals on the given mint, returning the resulting string in the return data
UiAmountToAmount	Permissionless	The inverse of `AmountToUiAmount`
GetAccountDataSize	Permissionless	Returns the required size to store an `Account` in the return data
Batch	Permissioned based on executed instructions	Executes multiple instructions in a single call, saving on CUs by reducing CPI calls. This is unique to P-Token

5 | Pinocchio Overview

Pinocchio is a lightweight replacement for the Solana program library, with a focus on reducing CU usage. The overall interface remains very close to the original, however, with the exception of a new logging library and a lazy entrypoint, which allows for more control over the parsing of the data passed to the smart contract by the runtime.

Unlike in the Solana program library, there are now a multitude of unsafe-marked functions and methods. These exist when checks might not always be necessary, such as when borrowing account data. This is a good pattern, as the user of the library has to explicitly wrap the function call into an `unsafe` block, which forces them to think about what they're doing is actually safe, while allowing more experienced developers to save CUs where possible.

In order to cut down on the number of dependencies, many data types have been redefined in Pinocchio. This reduces the overall risk of supply chain attacks and reduces the size of binaries, cutting down on state bloat and rent costs. However, this is also the primary challenge in the adoption of Pinocchio, as interacting with other smart contracts that don't have an interface written for Pinocchio either requires falling back on the types in the old Solana program library or implementing the interface yourself. Pinocchio itself provides interfaces for `token`, `token2022`, `associated-token-account`, `memo` and `system`.

Syscalls, memory management, sysvars and CPI work analogously to the old Solana program library. Some interfaces have changed slightly to accommodate the new types.

Logging has been completely overhauled. The rust `format!` macro is very expensive in terms of CU, which is why it shouldn't be used in smart contracts. Pinnochio now offers an alternative logging macro, based on a Logger implementation that is able to build strings from commonly-used types without too much CU overhead.

6 | Fuzzing P-Token with Riverguard

We have implemented a new fuzzcase for Riverguard. It executes all transactions on mainnet that uses the token program in real time, and replaces its ELF with P-Token. We can then check whether all accounts are identical at the end of the transaction, as well as that all of the metadata matches.

As P-Token uses significantly fewer compute units, we can't match those, and we can't fully match the program's logs, as CU usage is part of that. In addition, transactions that originally failed because of exceeding the compute budget sometimes succeeded because using P-Token instead of the old token program resulted in the transaction requiring less CUs, for which we put an exception in place in order to not get too many false positives.

After running for multiple months, we have not found a single instance of a transaction that was executed on mainnet resulting in different account data or a different error code due to behaviour between P-Token and the old token program.

Investigating the impact on CU usage on mainnet

This fuzzcase also allowed us to very easily determine the exact amount of CUs that would be saved on mainnet in total by the replacement of the old token program with P-Token. As P-Token can be compiled both with logging and without logging, we investigated the impact of both.

From 2025-08-03 up to and including all of 2025-08-11, the replacement would have saved a total of 8.90T CUs with logging enabled, and 9.14T CUs with logging disabled. According to a dune query, the chain processed a total of 74.50T CUs in that time. This corresponds to the following savings:

Variant	CUs saved (absolute)	Relative amount of used blockspace saved
With logging	8.90T	12.0%
Without logging	9.14T	12.3%

Note that these are not the savings relative to just the blockspace used up by the token program. These are the savings relative to the entire used blockspace, including failed and vote transactions.

7 | Findings

We're happy to share that we did not find any issues with the P-Token program or with Pinocchio. Both are heavily based on already existing and established code bases. During the initial auditing phase we discovered some behavioral missmatches between P-Token and the old Solana token program, however these were already spotted by Anza's fuzzer and remediated in a pull request, so we won't be listing them here.

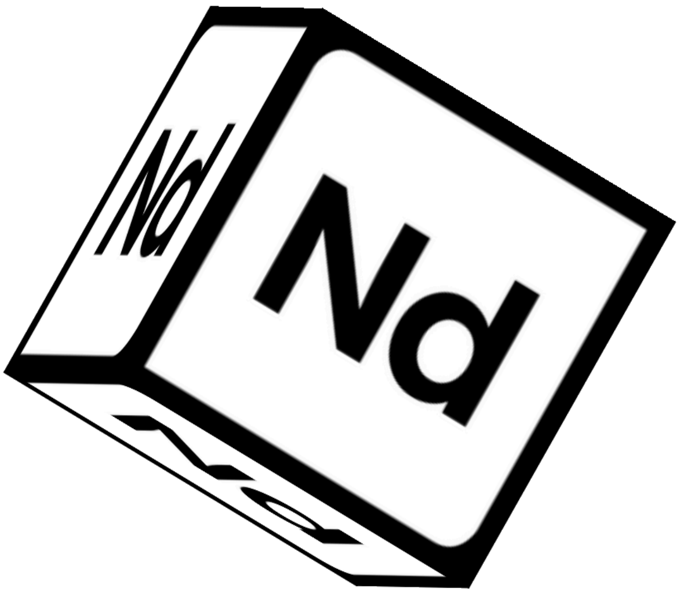
A | About Neodyme

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