Security Audit - Metaplex Token Metadata

conducted by Neodyme AG

Lead Auditor:	Sebastian Fritsch
Second Auditor:	Nico Gründel
Administrative Lead:	Thomas Lambertz

July 01, 2025

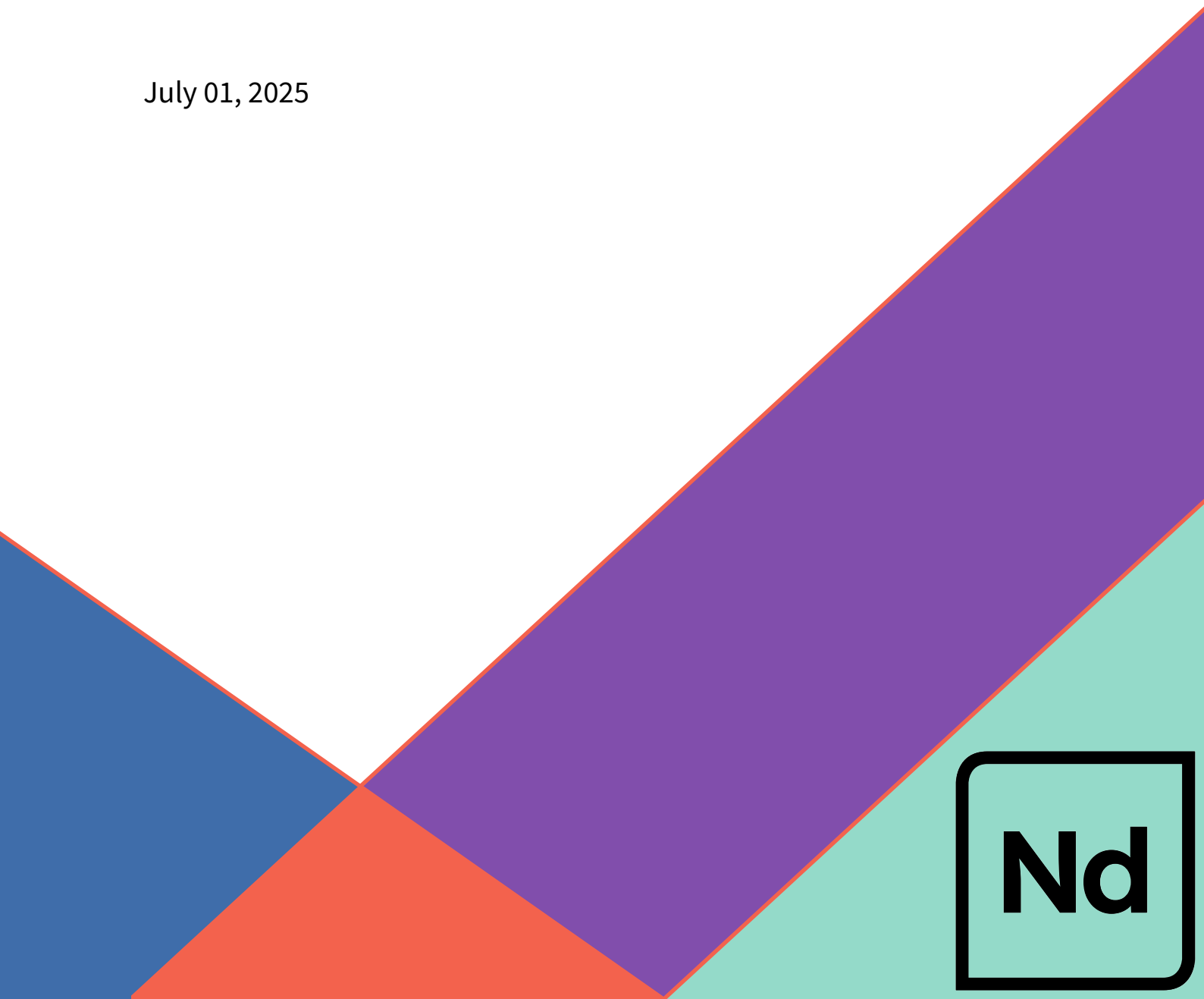

Table of Contents

Appendices

1 | Executive Summary

Neodyme audited Metaplex's on-chain **Token Metadata** program from March 2025 until July 2025. The audit was part of the Solana Foundation efforts to make the Token Metadata program immutable due to its crucial role in the Solana NFT ecosystem. Three auditing companies from the Solana ecosystem were chosen (Sec3, OtterSec and Neodyme) to audit the program, where we were doing the last pass.

Due to the threat model for Metaplex, the scope of this audit included the security of the implementation, evaluating the risks and benefits of making the program immutable and potential effects to the Solana NFT ecosystem. According to Neodymes Rating Classification, **2 security relevant** and **5 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

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

2 | Introduction

From March 2025 until July 2025, Metaplex engaged Neodyme to do a detailed security analysis of their Metaplex Token Metadata. Two senior security researchers from Neodyme conducted independent full audits of the contract between the 25th of March 2025 and the 1st of July 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.

The audit focused on the contract's technical security and an evaluation of the consequences of immutability. In the following sections, we present our findings and discuss worst-case scenarios for authority compromise and provide some general notes for considerations that may be useful in the future.

Summary of Findings

All found issues were **quickly remediated**. In total, the audit revealed:

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

issues. We further discuss all authorities in Section 4.4.

3 | Scope

The contract audit's scope comprised three major components:

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

Neodyme considers the source code, located at https://github.com/metaplex-foundation/security-mpl-token-metadata, in scope for this audit. Third-party dependencies are not in scope.

During the audit, minor changes and fixes were made by Metaplex, which the auditors also reviewed in-depth.

Relevant source code revisions are:

- `e55497a3122abda1b59a806f44ccaa1dbeaa8e41` · Start of the audit
- `4284d5d1608da887a79d64e3765efb2809ab87da` · Last reviewed revision

4 | Project Overview

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

Functionality

Metaplex's Token Metadata program is the de facto standard for adding Metadata to SPL Mints. It offers six different token standards that can be assigned to a mint:

- `Fungible`: A token mint without any restrictions
- `FungibleAsset`: A token mint, that has zero decimals
- `NonFungible`: A token mint, that has zero decimals and a supply of exactly one. The mint authority has to be owned by Metaplex.
- `ProgrammableNonFungible`: Same as `NonFungible`, but all token transfers of this token have to be through Metaplex, which can enforce further rules
- `NonFungibleEdition`: A token mint, with a supply of one and decimals of zero, that is "printed" as an edition of a `NonFungible`.
- `ProgrammableNonFungibleEdition`: A token mint, with a supply of one and decimals of zero, that is "printed" as an edition of a `ProgrammableNonFungible`.

Additionally, Metaplex has a broad feature set to improve the user experience of dealing with token metadata and NFTs. Those include

- Updating the Metadata
- Transferring assets as well as preventing the transfer of assets through unwanted channels using programmable assets
- Burning assets
- Printing editions of master editions
- Adding mints to collections and verifying their membership
- Adding verified creators to prove the validity of a NFT
- Delegating fine-grained authorities for different tasks (for the metadata itself as well as for token accounts)
- Locking assets
- Creating Escrow accounts, such that NFTs can be enabled to hold assets for themselves

On-Chain Data and Accounts

Metaplex uses several different accounts to store its on-chain information. Most importantly, the `Metadata` account serves as the foundational record of Metadata for a token mint. If the token mint is of type `NonFungible` or `ProgrammableNonFungible`, an additional `MasterEditionV2` account is created.

If the token mint is of type `NonFungibleEdition` or `ProgrammableNonFungibleEdition`, an additional `EditionV1` account is created.

In the table below, all different account types are listed and explained briefly.

AccountType	Summary
`MetadataV1`	PDA of `[program_id, mint]`, stores metadata and token standard
`MasterEditionV2`	PDA of a `[program_id, mint, EDITION]`, serves as type marker and stores the printable supply
`EditionV1`	PDA of a `[program_id, mint, EDITION]`, serves as type marker, stores the parent `MasterEdition`
`EditionMarker`	Stores which edition numbers have been printed for a `NonFungible`
`EditionMarkerV2`	Stores which edition numbers have been printed for a `ProgrammableNonFungible`
`ReservationList`	Not used anymore
`TokenRecord`	Stores delegate information and state for a Token account of a programmable asset
`UseAuthorityRecord`	PDA of `[program_id, mint, USER, user]`, stores delegated uses for `user`
`CollectionAuthorityRecord`	Delegate record to set and verify collections
`TokenOwnedEscrow`	Stores information about an escrow account for a NFT
`MetadataDelegate`	Stores pubkey and permissions of a MetadataDelegate
`HolderDelegateRecord`	Stores pubkey and permissions of a TokenDelegate

Instructions

The contract has a total of 44 instructions, which we briefly summarize here.

Instruction	Permission	Summary
Burn		
`BurnV1`	Token-Owner	Burns a specified amount of tokens and closes `Metadata` and `Edition` accounts
Close		
`CloseAccounts`	OwnerlessCloseAuthority	Closes `Metadata` and `Edition` accounts, if mint supply is zero
Collection		

Instruction	Permission	Summary
`ApproveCollectionAuthority`	Update Authority	Creates a `CollectionAuthorityRecord` for a mint
`RevokeCollectionAuthority`	Update Authority or CollectionDelegate	Closes a `CollectionAuthorityRecord`
`SetAndVerifyCollection`	Update Authority or CollectionDelegate	Adds a `Metadata` to a verified `Collection`
`SetAndVerifySized--CollectionItems`	UpdateAuthority or CollectionDelegate	Same as `SetAndVerifyCollection` but for sized collections
`SetCollectionSize`	UpdateAuthority or CollectionDelegate	Sets the collection size
`UnverifyCollection`	UpdateAuthority or CollectionDelegate	Removes the verified flag from a `Metadata` collection
`UnverifySizedCollectionItem`	UpdateAuthority or CollectionDelegate	Same as `UnverifyCollection` but for sized collections
`VerifyCollection`	UpdateAuthority or CollectionDelegate	Sets a `Metadata` collection to verified
`VerifySizedCollectionItem`	UpdateAuthority or CollectionDelegate	Same as `VerifyCollection` but for sized collections
Delegate		
`Delegate`	UpdateAuthority or TokenOwner	Creates a `MetadataDelegate`, `HolderDelegate` or `TokenDelegate` record
`Revoke`	UpdateAuthority or TokenOwner	Closes a record created via `Delegate`
Edition		
`ConvertMasterEditionV1ToV2`	Permissionless	Converts a `MasterEditionV1` to a `MasterEditionV2`
`MintNewEditionFrom--MasterEditionViaToken`	PrintDelegate or TokenOwner	Prints an `Edition` from a `MasterEdition` NFT
`CreateMasterEdition`	UpdateAuthority	Creates a `MasterEdition` account for a mint with decimals of zero and supply of one. Transfers the Mint authority
Escrow		

Instruction	Permission	Summary
`CreateEscrowAccount`	Permissionless	Creates an escrow account for a `NonFungible`
`CloseEscrowAccount`	EscrowAuthority	Closes an escrow account
`TransferOut`	EscrowAuthority	Transfers tokens out of an SPL token account that is owned by the escrow account
Fee		
`CollectFees`	FeeAuthority	Collects fees from a `Metadata` account
Freeze		
`FreezeDelegatedAccount`	TokenDelegate	Freezes a SPL Token account
`ThawDelegatedAccount`	TokenDelegate	Thaws a SPL Token account
Metadata		
`Create`	MintAuthority	Create a `Metadata` account for a mint and optionally a `MasterEdition`
`CreateMetadataAccountsV3`	MintAuthority	Create a `Metadata` account for a mint
`Mint`	UpdateAuthority or MintAuthority	Mints tokens to a token account, NFTs can only have a supply of 1
`Print`	PrintDelegate or TokenOwner	Creates an edition mint form a `MasterEdition`
`PuffMetadata`	Permissionless	Pads metadata name, symbol and URI with null bytes
`RemoveCreatorVerification`	Creator	Removes the verified flag for a metadata creator
`SetTokenStandard`	UpdateAuthority	Sets token standard according to the mints decimals, supply and if an edition is provided
`SignMetadata`	Creator	Sets the verified flag for a metadata creator
`Transfer`	TokenOwner or TokenDelegate	Transfers tokens of a `Metadata` mint while enforcing pNFT rules
`Update`	UpdateAuthority or MetadataDelegate	Updates metadata fields
`UpdateMetadataAccountV2`	UpdateAuthority	Updates metadata fields

Instruction	Permission	Summary
`UpdatePrimarySale--HappenedViaToken`	TokenOwner	Set the PrimarySaleHappend field to true
Resize		
`Resize`	TokenOwner or ResizeAuthority	Reduces the `Metadata` and `Edition` account size
State		
`Lock`	TokenOwner or TokenDelegate	Freezes account or sets its state to `Locked` for pNFTs
`Unlock`	TokenOwner or TokenDelegate	Unfreezes account or sets its state to `Unlocked` for pNFTs
Uses		
`ApproveUseAuthority`	TokenOwner	Creates UseAuthorityRecord for a user
`RevokeUseAuthority`	TokenOwner	Closes an UseAuthorityRecord
`Utilize`	UseAuthority	Decrements `metadata.uses` by a specified number. Can burn a token, if the uses reach zero
Verification		
`VerifyCollectionV1`	UpgradeAuthority or MetadataDelegate	Sets a metadata collection to verified
`VerifyCreatorV1`	Creator	Sets the verified flag for a metadata creator
`UnverifyCollectionV1`	UpgradeAuthority or MetadataDelegate	Removes the verified flag for a metadata collection
`UnverifyCreatorV1`	Creator	Removes the verified flag for a metadata creator

Authority Structure

The planned immutability of the Metaplex Token Metadata program raises the question of which non-upgrade authorities remain in the current program, what powers they hold, and what the worst-case impact on the NFT ecosystem would be if one of these authorities were to act maliciously.

SEED_AUTHORITY

The `SEED_AUTHORITY` (`AqH29mZfQFgRpfwaPoTMWSKJ5kqauoc1FwVBRksZyQrt`) can be found here in the source code. This public key was the former upgrade authority of the Metaplex Token Metadata program before it was transferred to the new upgrade council.

It is used in `process_create_metadata_accounts_logic` to bypass the restriction that only the mint authority of a specific mint can create the corresponding metadata account:

```
assert_mint_authority_matches_mint(&existing_mint_authority,
mint_authority_info).or_else(
    |e| {
  // Allow seeding by the authority seed populator
  if mint_authority_info.key == &SEED_AUTHORITY &&
     mint_authority_info.is_signer {
      // When metadata is seeded, the mint authority should be able to change it
      if let COption::Some(auth) = existing_mint_authority {
          update_authority_key = auth;
          is_mutable = true;
      }
      Ok(())
  } else {
      Err(e)
  }
    },
)?;
```

We assess the impact of this feature on the NFT ecosystem as non-critical, even if the authority were compromised, as the original mint authority can still change the associated metadata.

Nevertheless, we recommend removing this authority from the immutable release, as there is no valid reason to retain it.

As a result of this report, Metaplex decided to remove the `SEED_AUTHORITY` in commit `4284d5d1608da887a79d64e3765efb2809ab87da`. Neodyme verified this.

FEE_AUTHORITY

The `FEE_AUTHORITY` (`Levytx9LLPzAtDJJD7q813Zsm8zg9e1pb53mGxTKpD7`) can be found here in the source code.

It is used in `process_collect_fees` to collect metadata account creation fees. These fees are paid as additional rent during account creation. The `FEE_AUTHORITY` collects this additional rent and reduces the account's lamports to the rent-exempt minimum. The fees are transferred to `FEE_DESTINATION` (`2fb1TjRrJQLy9BkYfBjcYgibV7LUsr9cf6QxvyRZyuXn`).

Beyond this, the `FEE_AUTHORITY` has no additional powers. It will remain in the immutable program, as it is part of the legal contract between Metaplex and the Solana Foundation.

`OWNERLESS_CLOSE_AUTHORITY`

The `OWNERLESS_CLOSE_AUTHORITY` (`C1oseLQExhuEzeBhsVbLtseSpVgvpHDbBj3PTevBCEBh`) can be found here in the source code.

It is used in `process_close_accounts` to close metadata accounts:

- where the mint has been closed, or
- where the mint supply is zero, and no mint authority is set, or
- where the mint supply is zero, and the NFT is an edition (Metaplex will also close the correlated edition account).

These cases typically apply to burned NFTs. The rent from these accounts is transferred to `OWNERLESS_CLOSE_DESTINATION` (`GxCXYtrnaU6JXeAza8Ugn4EE6QiFinpfn8t3Lo4UkBDX`).

Beyond this, the `OWNERLESS_CLOSE_AUTHORITY` has no additional powers.

`RESIZE_AUTHORITY`

The `RESIZE_AUTHORITY` (`ResizebfwTEZTLbHbctTByvXYECKTJQXnMWG8g9XLix`) can be found here in the source code .

It is used in the `process_resize` instruction. The resize feature, activated on October 10, reduces the account size of metadata accounts by removing unused account space.

The size reduction can be triggered:

- by the owner of the NFT, transferring the excess rent lamports to themselves, or
- by the `RESIZE_AUTHORITY`, transferring the excess rent lamports to `RESIZE_DESTINATION`.

According to the feature announcement, the `RESIZE_AUTHORITY` will not be used until April 25, 2025. Until then, users can claim the rent themselves. However, this date is not enforced in the code, meaning the `RESIZE_AUTHORITY` could claim the rent prematurely.

`BUBBLEGUM` PDAs

The Bubblegum program introduced compressed NFTs to Solana. To ensure compatibility, the Token Metadata Program includes specific capabilities for Bubblegum.

In `process_create_metadata_accounts_logic`, Bubblegum can set the `allow_direct_creator_writes` and `allow_direct_collection_verified_writes` flags:

```
1 // This allows the Bubblegum program to create metadata with verified creators
  since they were
2 // verified already by the Bubblegum program.
3 let is_decompression = is_decompression(mint_info, mint_authority_info);
```

```
4 let allow_direct_creator_writes = allow_direct_creator_writes || is_decompression;
5 ...
6 let allow_direct_collection_verified_writes = is_edition || is_decompression;
```

The `is_decompression` function checks that the `mint_authority` is the corresponding PDA of the Bubblegum Program for the specific mint pubkey and that this PDA has signed the instruction.

Verified Creators Feature

The `allow_direct_creator_writes` flag enables Bubblegum to skip the checks for the Verified Creators feature.

Typically, when creating an NFT, users specify a list of public keys involved in its creation. These public keys are all marked as unverified. Using `verify_creator_v1`, a user can sign and verify their involvement in the creation of the NFT.

Bubblegum bypasses this process because it has already verified the creators in the Bubblegum program.

Verified Collections Feature

The `allow_direct_collection_verified_writes` flag allows the Bubblegum program to skip the checks for the Verified Collections feature.

Normally, an NFT creator specifies the collection field in the metadata. This assignment is considered unverified until the collection authority confirms it using `verify_collection`.

Bubblegum bypasses this step because it has already verified the collection association.

Compromise of the Bubblegum program

The Bubblegum program is currently deployed as an upgradable contract at address `BGUMAp9Gq7iTEuizy4pqaxsTyUCBK68MDfK752saRPUY` with upgrade authority `GUMAqc14ZC6z4fEmPrHWHc8HJg3KJdtB6JBjfXcLSyB`.

If the Bubblegum program is compromised, it could critically impact the Solana NFT ecosystem. An attacker could create NFTs that falsely appear to belong to collections like MadLads, seemingly verified by their creators, with no distinguishable on-chain evidence.

As a result of this report, Metaplex decided to move the Bubblegum upgrade authority to a multisig deployed at `bfQVv6niKVgEURYqQ1beJmiEQQN7MrvLRvk3mZGFubb`.

BUBBLEGUM_SIGNER

The `BUBBLEGUM_SIGNER` (`4ewWZC5gT6TGpm5LZNDs9wVonfUT2q5PP5sc9kVbwMAK`) can be found here in the source code.

It is used in the `bubblegum_set_collection_size` instruction to update the size (i.e., the number of associated members) of an NFT collection. This feature has been deprecated because there is no reliable way to track collection size due to NFT burns. There is also no reference to this instruction in the

As such, the veracity of a collection size must be questioned, and a compromise of this authority would not cause significant disruptions to the Solana NFT ecosystem.

`Token Auth Rule Program`

Programmable NFTs (pNFTs) are a special token type introduced by the Metaplex Token Metadata Program. They enable NFT creators to implement more fine-grained access controls and restrictions on NFTs. This is achieved by setting the freeze authority of an SPL token account to Metaplex, thereby restricting access to the underlying SPL Token Program by freezing the accounts. As a result, Metaplex is enforced as a proxy for operations concerning the NFT.

pNFT-specific rules are enforced during two types of operations:

- `Delegate`: Approves a new delegate for a specific operation (e.g., allowing the delegate to burn the NFT).
- `Transfer`: Permits the transfer of the underlying token only if the rules are met (e.g., requiring a second signer specified in the rule set).

The creation, updating, and **enforcement** of these rules do not occur within the Metaplex Token Metadata Program itself. Instead, they are managed by a separate program called Token Auth Rules. This program is deployed as an upgradable contract at `auth9SigNpDKz4sJJ1DfCTuZrZNSAgh9sFD3rboVmgg`, and the upgrade authority was held by `5jAhh15D6HDZbyysHFvJ1PAXgYcgsahkNW3yN5Q6EZCw`.

As a result of this report, Metaplex decided to move the Token Auth upgrade authority to a multisig deployed at `bfQVv6niKVgEURYqQ1beJmiEQQN7MrvLRvk3mZGFubb`.

Potential Consequences of a Compromised Token Auth Rules Program

A compromise of the Token Auth Rules Program would have several critical consequences:

- **Rule Enforcement Failure**: All rules created for a pNFT would become unenforceable, rendering features such as creator royalties ineffective.
- **DoS**: The Token Auth Rules Program could fail all CPI invocations, making it impossible to transfer any pNFT or set any delegate.
- **Security Exploitation**: During the validation of a Transfer operation, the signing token authority is passed to the Token Auth Rules Program without revoking the signer's privilege. An attacker could exploit this to drain all SOL funds from the token holder.

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 Metaplex team a list of nit-picks and additional notes, which are not part of this report.

All findings are listed in Table 2 and further described in the following sections.

Identifier	Name	Severity	Status
`ND-MPL-HI-01`	Anyone can close a master edition	**HIGH**	Resolved
`ND-MPL-LO-01`	No type check on creation of Metadata for a mint	**LOW**	Resolved
`ND-MPL-IN-01`	User can create `FungibleAssets` with decimals $\neq 0$	**INFORMATIONAL**	Resolved
`ND-MPL-IN-02`	Creator verification conflicts with `is_mutable`	**INFORMATIONAL**	Acknowledged
`ND-MPL-IN-03`	User can create Master Edition for fungible asset	**INFORMATIONAL**	Acknowledged
`ND-MPL-IN-04`	User can create a `Print` delegate for fungibles	**INFORMATIONAL**	Resolved
`ND-MPL-IN-05`	Fee for NFT creation is not configurable	**INFORMATIONAL**	Acknowledged

Table 2: Findings

[ND-MPL-HI-01] Anyone can close a master edition

Severity	Impact	Affected Component	Status
HIGH	Loss of rent, ecosystem disruption	Burn functionality	Resolved

Metaplex uses an account at the PDA `[program_id, mint, EDITION]` of type `MasterEditionV2` to indicate that the metadata linked to the mint represents an NFT and serves as its master edition.

With the `BurnV1` instruction, an NFT owner can burn their token and close the associated `Metadata` and `MasterEditionV2` accounts.

While the `Metadata` account was properly validated for the mint, there was no verification that the provided master edition account matched the correct PDA, since `metadata.edition_nonce` is set by default:

burn/nonfungible.rs, lines 44-55

```
// Has a valid Master Edition or Print Edition.
let edition_info_path = Vec::from([
    PREFIX.as_bytes(),
    crate::ID.as_ref(),
    ctx.accounts.mint_info.key.as_ref(),
    EDITION.as_bytes(),
]);

let bump = match args.metadata.edition_nonce {
    Some(bump) => Ok(bump),                                    //<- NO CHECK
    None => assert_derivation(&crate::ID, edition_info, &edition_info_path),
}?;
```

This allowed an attacker to close the master edition accounts of NFTs they did not control. This results in the loss of rent funds stored in those edition accounts and poses a significant disruption risk to the NFT ecosystem. However, the attacker could not profit, as creating a malicious master edition also required paying rent.

The issue could be mitigated by the original NFT owner recreating the master edition using `CreateMasterEditionV3`, although he would have to pay the rent again.

Resolution

The Metaplex team responded quickly, confirmed the bug, and published a fix in commit `5bf34e68ad6491727dd598d0ad41b93a65ad79f1`. Neodyme and other auditors verified the resolution.

[ND-MPL-LO-01] No type check on creation of Metadata for a mint

Severity	Impact	Affected Component	Status
LOW	Inconsistent state	Metadata creation	Resolved

When creating a `Metadata` account for a mint using `CreateMetadataAccountsV3`, there is no check to ensure that the provided `mint_info` is actually of type `Mint`. The function only verifies that `mint_info` is owned by the SPL program and that `mint_info.mint_authority` is a signer. Potential type confusion targets include the `Account` and `Metadata` structures, which are also SPL-owned.

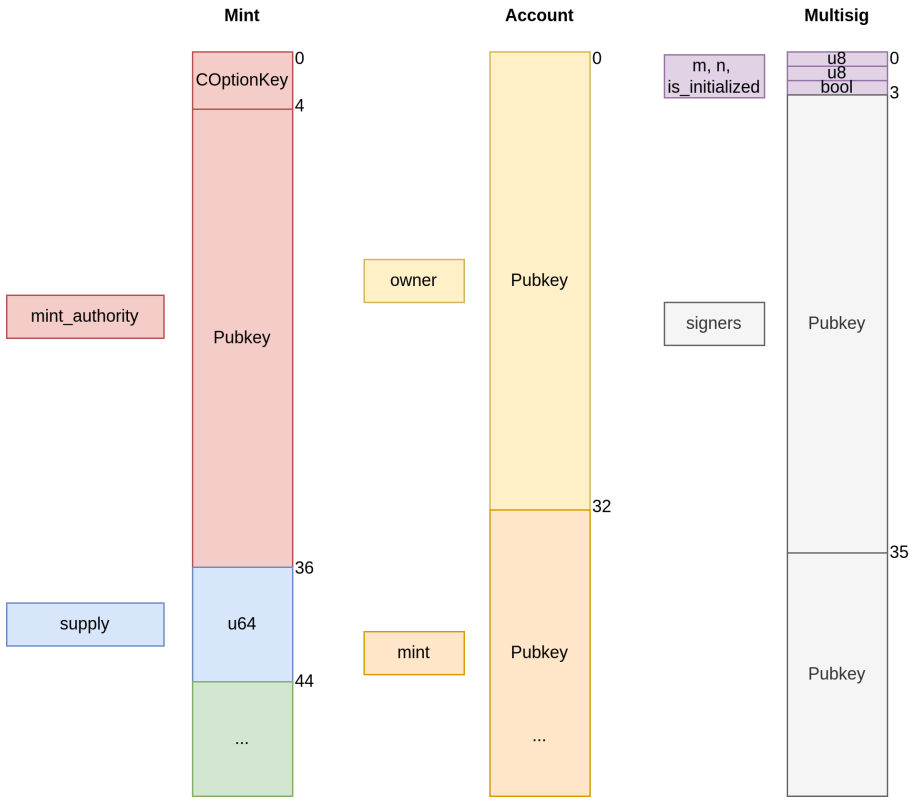

Figure 2: Struct layouts

As shown in Figure 2, the `mint_authority` field layout does not match those of the other types. An attacker would need to brute-force at least 28 bytes of a keypair to impersonate the `mint_authority` for an `Account` or `Multisig`, which is currently infeasible.

Only the `SEED_AUTHORITY` can bypass the signer check for `mint_authority`, making this issue exploitable solely by the Metaplex team.

Resolution

Metaplex resolved the issue by removing the `SEED_AUTHORITY` in commit `4284d5d1608da887a79d64e3765efb2809ab87da`. Neodyme verified the fix.

[ND-MPL-IN-01] User can create `FungibleAssets` with decimals $\neq 0$

Severity	Impact	Affected Component	Status
INFORMATIONAL	Inconsistent Token Standard	Token Standard	Resolved

In `UpdateV2`, the update authority of a metadata account can specify a new desired token standard.

metadata/update.rs, lines 333-345

```
fn check_desired_token_standard(
    existing_or_inferred_token_std: TokenStandard,
    desired_token_standard: TokenStandard,
) -> ProgramResult {
    match (existing_or_inferred_token_std, desired_token_standard) {
        (
            TokenStandard::Fungible | TokenStandard::FungibleAsset,
            TokenStandard::Fungible | TokenStandard::FungibleAsset,
        ) => Ok(()),
        (existing, desired) if existing == desired => Ok(()),
        _ => Err(MetadataError::InvalidTokenStandard.into()),
    }
}
```

A change is only allowed between `Fungible` and `FungibleAsset`, and vice versa. However, there is no check ensuring that the mint's decimals are zero when switching to `FungibleAsset`, which violates its specification.

Resolution

Metaplex fixed the issue in commit `4284d5d1608da887a79d64e3765efb2809ab87da`. Neodyme verified the fix.

[ND-MPL-IN-02] Creator verification conflicts with `is_mutable`

Severity	Impact	Affected Component	Status
INFORMATIONAL	-	Creator verification	Acknowledged

Metaplex allows a `Metadata` creator to mark the metadata as immutable, ensuring that fields like name, URI, and other data remain unchanged. This data resides in the `metadata.data` field. Once `is_mutable` is set to false, it should no longer be modified:

```
pub struct Metadata {
  ...
  /// Asset data.
  pub data: Data,
  // Whether or not the data struct is mutable, default is not
  pub is_mutable: bool,
  ...
}
```

The `Data` struct also stores the creators of the metadata and whether they are verified (i.e., they signed a statement confirming they created the NFT).

Verification and unverification are handled via `SignMetadata`, `RemoveCreatorVerification`, `VerifyCreatorV1`, and `UnverifyCreatorV1`. These functions do not check the `is_mutable` flag and can modify the `Data` struct even when the metadata is marked as immutable.

Since this appears to be intentional, we recommend updating the documentation to clarify this behavior.

Resolution

Metaplex acknowledged the finding and expanded its documentation on this.

[ND-MPL-IN-03] User can create Master Edition for fungible asset

Severity	Impact	Affected Component	Status
INFORMATIONAL	-	Master Editions	Acknowledged

In `CreateMasterEditionV3`, there is no check to ensure that `metadata.token_standard` is set to `NonFungible`. The function only verifies that `mint.decimals` is zero and the supply is one.

As a result, a `Metadata` with token standard `FungibleAsset` can still have an associated `MasterEditionV2` account.

While we found no exploitable impact from this inconsistency, we recommend adding the check for correctness and clarity.

Resolution

Metaplex acknowledged the issue as acceptable and intended behaviour.

[ND-MPL-IN-04] User can create a `Print` delegate for fungibles

Severity	Impact	Affected Component	Status
INFORMATIONAL	-	Delegation	Resolved

Metaplex supports creating delegates for various metadata-related functions.

One such role is `Print`, which allows the current NFT holder to delegate the ability to print editions of a master edition to another account.

However, Metaplex does not verify that the metadata in `Delegate` corresponds to a master edition NFT. As a result, print delegates can be created for tokens with the `Fungible` or `FungibleAsset` token standards.

Attempts to use the `Print` instruction on such tokens will fail due to a proper check ensuring only master editions can be printed. Thus, we see no exploitable vector but recommend addressing the inconsistency.

Resolution

Metaplex mitigated the issue in commit `4284d5d1608da887a79d64e3765efb2809ab87da`. Neodyme verified the fix.

[ND-MPL-IN-05] Fee for NFT creation is not configurable

Severity	Impact	Affected Component	Status
INFORMATIONAL	Cost of minting NFTs might make the creation of new non-compressed NFTs infeasible in the future	NFT creation	Acknowledged

Metaplex levies a 0.01 SOL for the creation of NFTs. This amount is hard-coded in the contract, which means no one will be able to change it once the contract is read-only. As this fee is ultimately arbitrary and doesn't serve a protective purpose similar to rent, an increase in the price of SOL might make it economically infeasible to mint new NFTs in the future using Token Metadata with no clear upside.

Resolution

Metaplex acknowledged the issue as acceptable and intended behaviour.

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