Metaplex Token Metadata Program Audit Report

18.12.2023

– Mad Shield

Table of Contents

1. Introduction **3**

1.1 Overview 4

1.2 System Specification 5

1. Account Structure 5

2. Unified Instruction API 5

3. Programmable NFTs (pNFTs) 6

4. SPL-Token-22 support 6

2. Scope and Objectives **7**

3. Methodology **8**

4. Findings & Recommendations **10**

1. SHIELD_MTM_01 - Burn instruction could be exploited to permanently disable all pNFT operations [Critical] 11

2. SHIELD_MTM_02 - All the pNFT rules can be bypassed in the transfer instruction [Critical] 13

3. SHIELD_MTM_03 - The pNFT AllowList rule can be bypassed in the transfer instruction [Critical] 14

4. SHIELD_MTM_04 - pNFTs can become non-transferable [High] 15

5. SHIELD_MTM_05 - Program panics during Lock/Unlock of fungible tokens [Low] 17

General Recommendations **18**

1. SHIELD_MTM_GR_01 - Token accounts for NonTransferable Mints must have the ImmutableOwner extension [Informational] 18

Conclusion **19**

References **19**

1. Introduction

Metaplex engaged Mad Shield to audit the MPL Token Metadata program. The audit included an initial full analysis of the core functionality of the library, with subsequent reviews following major releases. Over the time period between February to November 2023 we performed multiple intermittent cycles of intense security analysis of the codebase.

In summary, our audit yielded 5 vulnerabilities within the Metaplex Token Metadata program: 3 categorized as Critical, 1 High-severity, and 1 Low-severity. Additionally, we reported 1 Informational finding in the Token-22 program that may impact the Token Metadata program functionality. A detailed walk-through of these findings is provided in Section [4].

We communicated all our findings with proper mitigations through private channels to the Metaplex developer team. **We are glad to confirm that all the vulnerabilities disclosed here have since been addressed and resolved accordingly.** This document outlines the audit procedure, the assessment methodology, key findings and suggestions to further improve the program.

1.1 Overview

The Metaplex Token Metadata program, a core component of the Metaplex Program Library (MPL) for managing NFTs on Solana, has undergone significant updates to address evolving industry challenges and leverage new feature capabilities. These upgrades include:

- **Programmable NFTs (pNFTs)** – A novel digital asset standard empowering creators to define limitations and restrictions, such as royalty enforcement, enhancing creator control and value capture.
- **Token-22 Integration** – Expansion of the program's functionality to support the advanced features introduced by the Token-22 program on Solana, opening doors for broader enterprise adoption.

However, the addition of substantial new code presents potential risks alongside the anticipated benefits. Breaking changes, potential footguns, and security vulnerabilities can emerge within the revamped system.

Recognizing these issues, Metaplex proactively engaged Mad Shield for periodic security audits after each major code release. This report presents a comprehensive analysis of the implemented changes and their security implications, focusing on identifying and mitigating program vulnerabilities.

1.2 System Specification

Here, we go through some of the technical intricacies of Token Metadata. This is not intended to serve as a documentation; instead, it highlights the specific elements most relevant to the context of this audit. For the official documentation, see [1].

1. Account Structure

The primary accounts of the Token Metadata program for a NFT are `metadata` and `master_edition`. two new accounts are added to facilitate the new asset standard.

- **Token Record** – Stores delegate and token state information specific to pNFTs.
- **Metadata Delegate Record** – Stores information for delegates that can modify metadata details according to their role.

The account layout, relationships, seed derivation, and authority associations of the program referenced in this article are displayed in Fig. 1.

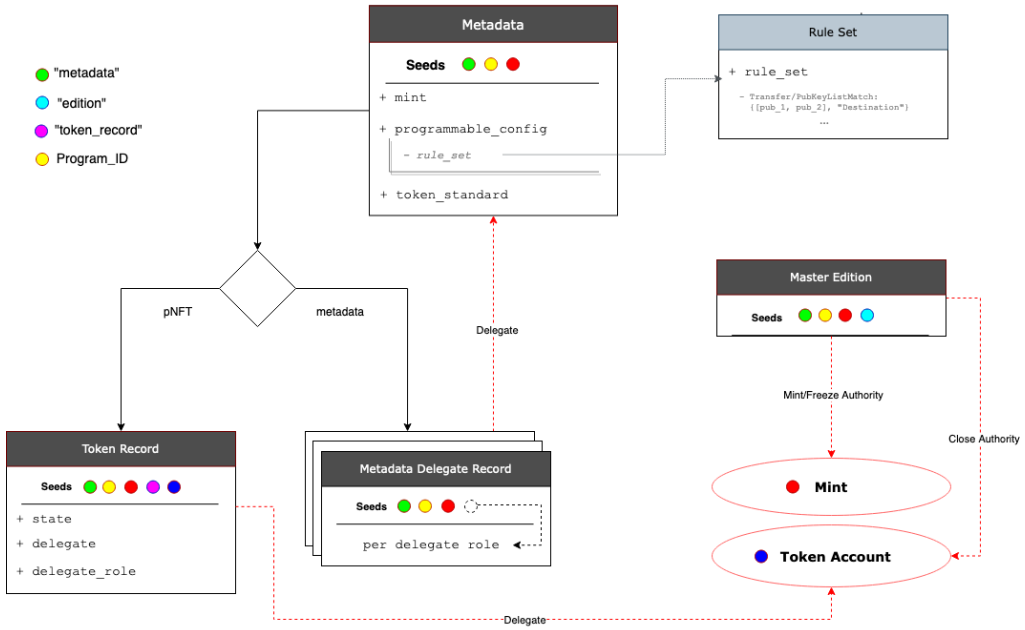

Fig 1. Account layout of Token Metadata program

2. Unified Instruction API

The program introduces a new set of instructions to support the new digital asset standard. This has significantly improved the navigability, readability and the quality of the codebase. Furthermore, it establishes an independent context for the implementation of the digital asset standard that is clearly separated from the legacy code.

3. Programmable NFTs (pNFTs)

The programmable NFTs are a new asset class which can be utilized by creators to establish permission policies for any instruction modifying the NFT state. The creation and validation of these policies is implemented in the separate Token Auth Rules program.

Each policy is represented by a mapping of <Operation, Rule> called a *RuleSet* that is stored in the `rule_set` account of the Token Auth Rules program. To apply a Rule Set to a pNFT, the `rule_set` account is stored in the `programmable_config` attribute of the metadata.

To enforce the rules on a pNFT operation, the Token Metadata program invokes the validate instruction of the Token Auth Rules program which requires that

- The details of the operation (e.g. destination of a transfer) are provided as a well-defined `Payload` structure in the input arguments of the instruction.
- The pNFT `rule_set` is in the instruction account list.

The Auth Rules Program then evaluates the `Payload` against `rule_set` returning successfully if the Rule Set is satisfied.

4. SPL-Token-22 support

The SPL-Token-22 program adds extra functionality and customizable behavior on top of the existing SPL-Token program. This is implemented by adding new fields in the form of extensions to both `token_account` and `mint_account` layouts. There are multiple extensions supported by the SPL-Token-22 program [2], however, Token Metadata only integrates the following few for NFTs and pNFTs.

Account	Extensions
Mint	*MintCloseAuthority*, *NonTransferable*, *Token MetadataPointer*
Token	*ImmutableOwner*, *NonTransferableAccount*

The main purpose of these extensions is to enable soul-bound assets where the NFT or pNFT cannot be moved and is attached to the holding account.

Additionally, Token Metadata leverages the instruction interface from the SPL-Token-22 program enabling compatibility with both Token program versions. The consistent usage of this interface across both programs is crucial to the integrity of the Token Metadata program.

2. Scope and Objectives

The primary objectives of the audit are defined as:

- Minimizing the possible presence of any critical vulnerabilities in the program. This would include detailed examination of the code and edge case scrutinization to find as many vulnerabilities.

- 2-way communication during the audit process. This included for Mad Shield to reach a perfect understanding of the design of Token Metadata and the goals of the Metaplex team.

- Provide clear and thorough explanations of all vulnerabilities discovered during the process with potential suggestions and recommendations for fixes and code improvements.

- Clear attention to the documentation of the vulnerabilities with an eventual publication of a comprehensive audit report to the public audience for all stakeholders to understand the security status of the programs.

The Scope encompasses

Programmable NFT (pNFT) –

Analysis of Pull Requests [#947, #993, #1005, #1014, #1015, #1017, #1028, #1030, #1058, #1061, #1096, and #1117] from the legacy `metaplex-program-library` repository

Token-22 Integration –

Analysis of Pull Request [#17] from the `mpl-token-metadata` repository [1]

1 The Token Metadata was moved to its own stand-alone repository with preserved commit history in July 2023.

3. Methodology

After the initial request from Metaplex to review we did a quick read-through of the code to evaluate the scope of the work and recognize early potential footguns. Due to the high complexity of the Token Metadata program, we were in constant contact with the Metaplex team to realize the design in detail and that the program implements it accordingly.

Since our team was familiar with the legacy instructions, we started by exploring the unified instruction set in the context of the legacy interface. Our primary goal besides getting a solid understanding of the implementation, was to check the mapping of the legacy instructions to the new ones ensuring that there were no deviations from the expected behavior leading to a safety violation.

Next, we shifted our focus to the implementation of the new asset standard. A symbolic execution generated a large execution tree for the program. This was the main challenge in our analysis as exploring all the execution paths was intractable. Therefore, together with the Metaplex team, we developed a heuristic to identify critical paths with higher security risk based on the usage frequency of the instructions, possibility of fund embezzlement, Rule Set violations and the mainnet deployment schedule.

As the Token Metadata program does not use Anchor, the standard framework for program development on Solana, we took extra measures in scrutinizing the account validations to test the program against classic Solana account re-initialization and substitution attacks. To this end, we specified the list of input accounts and the corresponding validations at the start of a symbolic program execution. Upon processing a control flow statement, this list was updated to remove the assertions verified up to that point. Any remaining validations at the end of the execution was a missing account check, and a potential security footgun for that path.

This technique turned out to be fruitful as we found multiple paths with missing validations that further inspection revealed two of them to be critical vulnerabilities (**SHIELD_MTM_01**, **SHIELD_MTM_02**). As a general guideline, it is imperative to avoid all security critical assertions such as account validations inside conditional branches where the assertion can be skipped due to non-deterministic execution.

Additionally, we analyzed the correctness and safety of the program with regard to composability with other programs. Token Metadata invokes instructions from SPL-Token, SPL-Token-22 and Token Auth Rules programs. In our assessment, we made sure to review the code of all the programs above to understand their design and specification at a high-level. We emphasize that this does not, by any means, constitute an audit for these programs by Mad Shield, thereby we command a black-box view, assuming these programs operate correctly within the boundaries of their specification.

To uncover potential vulnerabilities that arise from the interaction of the programs, we did extensive fuzzy testing targeted at Token Metadata's Cross Program Invocations (CPI) to SPL-Token(-22) and Token Auth Rules. We used the provided suite of testing tools such as Rooster to define various program and token account states, authority role assignments and input data mutations.

As a result, we identified several discrepancies (**SHIELD MTM 03**, **SHIELD MTM 04**, **SHIELD MTM 05**) in the assumptions regarding account/input sanitization between the caller and callee, specifically pertaining to the assertions that the Token Metadata program was responsible to enforce prior to program invocation.

4. Findings & Recommendations

Our severity classification system adheres to the criteria outlined here.

Severity Level	Exploitability	Potential Impact	Examples
Critical	Low to moderate difficulty, 3rd-party attacker	Irreparable financial harm	Direct theft of funds, permanent freezing of tokens/NFTs
High	High difficulty, external attacker or specific user interactions	Recoverable financial harm	Temporary freezing of assets
Medium	Unexpected behavior, potential for misuse	Limited to no financial harm, non-critical disruption	Escalation of non-sensitive privilege, program malfunctions, inefficient execution
Low	Implementation variance, uncommon scenarios	Zero financial implications, minor inconvenience	Program crashes in rare situations, parameter adjustments by authorized entities
Informational	N/A	Recommendations for improvement	Design enhancements, best practices, usability suggestions

In the following, we enumerate some of the findings and issues we discovered and explain their implications and corresponding resolutions.

Finding	Description	Severity Level
SHIELD_MTM_01 [RESOLVED]	Burn instruction could be exploited to permanently disable all pNFT operations	*Critical*
SHIELD_MTM_02 [RESOLVED]	All the pNFT rules can be bypassed in the transfer instruction	*Critical*
SHIELD_MTM_03 [RESOLVED]	The pNFT AllowList rule can be bypassed in the transfer instruction	*Critical*
SHIELD_MTM_04 [RESOLVED]	pNFTs can become non-transferable	*High*
SHIELD_MTM_05 [RESOLVED]	Program panics during Lock/Unlock of fungible tokens	*Low*
SHIELD_MTM_GR_01 [ACKNOWLEDGED]	Token accounts for NonTransferable Mints must have the ImmutableOwner extension	*Informational*

1. **SHIELD_MTM_01 - Burn instruction could be exploited to permanently disable all pNFT operations *[Critical]***

   To burn a pNFT, the program closes the following 4 accounts: `token_account`, `metadata`, `master_edition` and the `token_record`. In addition, only the *Holder* or a delegate with the *UtilityDelegate* role have the permission to burn the token.

   We noticed that account validation for the `token_record` PDA was non-existent if the transfer authority was the token owner. This would allow an attacker to close the `token_record` account of all pNFTs in existence with an account substitution exploit i.e. passing any `token_record` account instead of the correct one in an instruction that burns a pNFT they own.

   Upon further inspection, we realized that the assumption during initial development was to rely on the `get_authority_type` utility function to validate the `token_record` account. This function determines the authority type with the highest precedence in the permissible list of authority types for which the user has a valid authority.

```
pub fn get_authority_type(request: AuthorityRequest,) -> Result<AuthorityResponse, ProgramError>
{
        // the evaluation follows the `request.precedence` order; as soon as a match is
        // found, the type is returned
        for authority_type in request.precedence {
            match authority_type {
               AuthorityType::Holder => {
                    // checks if the authority is the token owner
                    if let Some(token_account) = request.token_account {
                        if cmp_pubkeys(&token_account.owner, request.authority) {
                            return Ok(AuthorityResponse {
                                authority_type: AuthorityType::Holder,
                                ..Default::default()
                            });
                        }
                    }
                }
                AuthorityType::TokenDelegate => {
                    // checks if the authority is a token delegate
                    let pda_key = find_token_record_account(request.mint, request.token);
                    // validation of the token_record account
                    if cmp_pubkeys(&pda_key, request.token_record) {
                                return Ok(AuthorityResponse {
                                    authority_type: AuthorityType::TokenDelegate,
                                    token_delegate_role: token_record.delegate_role,
                                    ..Default::default()
                                });
                     }
                }
                AuthorityType::MetadataDelegate => { ... }
                AuthorityType::Metadata => { ... }
            }
        }
        Ok(AuthorityResponse::default())
    }
}
```

The function has a short-circuit evaluation where it returns as soon as a matching authority type is found. As highlighted in the snippet, the account validation for `token_record` takes place in the *TokenDelegate* code block but not the *Holder.*

Since the *Holder* has a higher precedence over *TokenDelegate* in the burn instruction, if the authority is the pNFT owner, `get_authority_type` returns Holder as the authority type and the *TokenDelegate* code block – where the accounts are validated – is not executed

Recommended Fix

To address the problem, we recommended validating the `token_record` account derivation for programmable NFTs regardless of the invoking authority, eliminating the dependence on the `get_authority_type` function and its optimistic execution.

The patch commit: [4d1cc5ea]

2. SHIELD_MTM_02 - All the pNFT rules can be bypassed in the transfer instruction *[Critical]*

In the transfer instruction, the `rule_set` account must match the `rule_set` in the `programmable_config` attribute of the `metadata` account. As long as the [metadata,mint] dyadic relationship is checked, this ensures the pNFT transfer is validated against the correct *RuleSet*.

However, if the transfer authority was the token delegate, there was no validation of the `metadata`. An attacker could exploit this to bypass all the pNFT transfer rules by passing a `metadata` account that has a `rule_set` with no defined Rules.

The core of this issue is similar to SHIELD_MTM_01. The validity of the `metadata` is checked by the `assert_holding_amount` utility function which is called in one branch of a conditional, but not the other. This condition was to differentiate the `wallet_to_wallet` transfer scenario which is exempt from the pNFT Rule Set.

Recommended Fix

To include the `assert_holding_amount` function in all execution paths, we recommended moving the function outside the match expression over the authority type.

The patch commit: 7f163a17

3. SHIELD_MTM_03 - The pNFT AllowList rule can be bypassed in the transfer instruction *[Critical]*

One of the Rules in a pNFT `rule_set` is the definition of an *AllowList*, a set of public keys that the token can be transferred to. The primary use case of this rule is to only allow marketplaces that enforce creator royalties to escrow a pNFT.

In the transfer instruction, the public key of the destination `token_account` owner is provided in the `Payload` argument to the cross program invocation of the Token Auth Rules program where the *AllowList* is validated.

However, there was no check to ensure that the input public key was the owner of the `token_account`, binding them together. As a consequence, an attacker could bypass the Rule Set by providing a public key in the *AllowList* as the owner but a destination `token_account` with a different owner e.g. the authority of a marketplace violating the royalty enforcement.

Recommended Fix

To fix the issue, we suggested modifying the `validate_token` function to accept the owner as a parameter and add the condition that the public key matches the `token_account` owner.

The patch commit: [c4d3d596]

Extra Improvement

Specifying the owner address did not guarantee that the pNFT would be minted to the correct destination token account as the Mint instruction used the same `validate_token` function. Through the fix, the code was additionally improved to mitigate an inconsistency in the client and avoid user confusion.

4. SHIELD_MTM_04 - pNFTs can become non-transferable [High]

In the transfer instruction, the program attempts to clear the close authority of the source `token_account` of a pNFT. This was originally added to avoid another problem related to the *UtilityDelegate* role in which the close authority of the `token_account` would be set to the delegate. The problem was that upon transfer and without the clean-up of the close authority, closing and future re-delegation of the token_account was impossible as the current close authority is required to manipulate this field.

To avoid this, the `master_edition` account is set as the close authority of the token account where the program can then change this field through a cross program invocation of the SPL-Token program.

However, we noticed that the program doesn't enforce any conditions on the close authority of the destination `token_account` during transfer. A scenario could occur where the close authority of the destination is set to an account other than the `master_edition`. In this case, the transfer to this account renders the token untransferable as future calls to transfer fails due to the program expecting the `master_edition` as the close authority. This is showcased in the following code snippet.

```
if let COption::Some(close_authority) = token.close_authority {
    if let Some(edition) = ctx.accounts.edition_info {
        if close_authority ≠ *edition.key {
            return Err(MetadataError::InvalidCloseAuthority.into());
        }
    } else {
        return Err(MetadataError::MissingEditionAccount.into());
    };
}
... // proceed to the rest of the transfer
```

The same issue would occur in the mint instruction. The program did not perform proper checks on the `token_account` that the new pNFT would be minted to.

Recommended Fix

To alleviate this, we suggested adding an assertion to both the transfer and mint instructions where the close authority of the receiving `token_account` is validated as shown below.

```rust
if matches!(
    metadata.token_standard,
    Some(TokenStandard::ProgrammableNonFungible)
        | Some(TokenStandard::ProgrammableNonFungibleEdition)
) {
    if let COption::Some(close_authority) = token.close_authority {
        // the close authority must match the master edition if there is one set
        // on the token account
        if let Some(master_edition) = ctx.accounts.master_edition_info {
            if close_authority != *master_edition.key {
                return Err(MetadataError::InvalidCloseAuthority.into());
            }
        } else {
            return Err(MetadataError::MissingMasterEditionAccount.into());
        };
    }
}
```

This would guarantee the invariant that the `token_account` holding the pNFT (balance = 1), always has either the `master_edition` or `None` as its close authority, resulting in successful execution of the transfer instruction in perpetuity.

The patch commit: [7139b878]

5. SHIELD_MTM_05 - Program panics during Lock/Unlock of fungible tokens *[Low]*

The freeze authority of non-fungible tokens is set to the `master_edition` account. As such, the `Lock/Unlock` instruction of Token Metadata can toggle the freeze status of a NFT/pNFTs providing the proper seeds through a signed Cross Program Invocation of the SPL-Token `freeze/thaw` instructions.

However, this is not the case for fungible tokens as the freeze authority can belong to arbitrary keypairs. In the absence of the `master_edition` account, The program was incorrectly assigning the token owner or the delegate as the freeze authority during `Lock/Unlock` of fungible assets. This would cause the program to panic abruptly since the SPL-Token program does not recognize these entities as valid freeze authorities.

Recommended Fix

We suggested separating the scope of the `Lock/Unlock` instruction for pNFT/ NFTs from fungible tokens. The ability to modify the freeze status of the former was limited to the proper delegates via the `master_edition` account while the true freeze authority of the `mint_account` was required for the latter. The code snippet below illustrates the change.

```
let mint = unpack_initialized::<Mint>(&accounts.mint_info.data.borrow())?;
assert_freeze_authority_matches_mint(&mint.freeze_authority, accounts.authority_info)
    .map_err(|_| MetadataError::InvalidAuthorityType)?;
match to {
    TokenState::Locked => {
        // for fungible assets, we invoke spl-token directly
        // since we have the freeze authority
        invoke(
            &freeze_account(
                ...
                accounts.authority_info.key,
                &[],
            )?,
            ...
        )
    }
    TokenState::Unlocked => {
        // for fungible assets, we invoke spl-token directly
        // since we have the freeze authority
        invoke(
            &thaw_account(
                ...
                accounts.authority_info.key,
                &[],
            )?,
            ...
        )
    }
```

The patch commit: [a6e70845]

General Recommendations

In this section, we provide general recommendations and informational issues that we found during the process of the audit.

1. **SHIELD_MTM_GR_01 - Token accounts for NonTransferable Mints must have the ImmutableOwner extension [Informational]**

   The SPL-Token-22 program offers the *NonTransferable* extension for `mint_accounts` to allow definition of tokens soul-bound to an owner.

   There is also the *ImmutableOwner* extension, which independently prevents owner reassignment for any `token_account`. This is particularly useful in the context of Associated Token Accounts (ATAs) where the mint is not necessarily non-transferable but ownership immutability is enforced by default to avoid security vulnerabilities that have arised in the past.

   Although a *NonTransferable* token cannot be transferred by design, a potential inconsistency arises as SPL-Token-22 currently allows creating `token_accounts` without the *ImmutableOwner* extension for *NonTransferable* mints.

   While no critical vulnerabilities involving funds are introduced due to inherent transfer restrictions, this does

   - Create potential confusion regarding token functionality.
   - Risk unnecessary program panics if the extension is overlooked during mint.

   This was acknowledged by the SPL-Token-22 contributors and an issue was opened to address the possibility of creating such redundant `token_accounts`.

Conclusion

In conclusion, our thorough audit of the Metaplex Token Metadata program has yielded significant results. By identifying and remediating critical vulnerabilities, we've addressed potential threats jeopardizing user funds. Moreover, proposed solutions for the remaining vulnerabilities enhance the overall reliability and performance of the program.

By meticulously examining the codebase, this audit aims to ensure the continued robustness and security of the Metaplex Token Metadata, fostering trust and promoting safe adoption of new innovative features for both creators and users within the Solana NFT ecosystem.

References

[1] Metaplex Developer Hub – https://developers.metaplex.com/token-metadata
[2] Token-22 Extension Guide – https://spl.solana.com/token-2022#extensions