Metaplex Core Program Audit Report *(V-1.0.0)*

– Mad Shield

contact@madshield.xyz

Table of Contents

1. Introduction

Metaplex engaged Mad Shield to audit the MPL Core program, a new initiative aimed at introducing a simpler, more intuitive NFT standard interface on the Solana blockchain. The audit included an initial full analysis of the core functionality of the library, spanning approximately two weeks from April 1st to April 15th. Following this, a subsequent audit of the plugins took place over the course of a development cycle between July and August, concluding on August 8th.

This audit report presents the findings from our comprehensive security analysis of the program. In summary, our initial audit identified 5 vulnerabilities within the Metaplex Core program: 2 categorized as Critical, 1 Medium-severity, and 2 Low-severity. Additionally, we reported an Informational finding focused on the central authority validation unit in the program, recommending additional separation of duties and continuous incremental testing and auditing of this particular segment of the business logic.

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

1.1 Overview

The MPL-Core program is designed to serve as a foundational component within the Metaplex ecosystem, facilitating various functionalities centered around NFTs and token management on the Solana blockchain and to improve upon the original mpl-token-metadata program.

A critical transformation in this upgrade is the removal of the SPL-Token program and associated accounts, which simplifies the underlying architecture significantly against original design which was heavily reliant on multiple Program Derived Addresses (PDAs) from the mint account such as metadata and master editions.

Additionally, Metaplex Core introduces a plugin system, an enhancement that replaces the diverse functionalities previously scattered across various accounts in mpl-token-metadata. This allows for extensible development and integration, where specific functionalities can be incorporated as needed through these plugins.

This modular approach not only simplifies the account management by consolidating functionalities into a single account but also enhances the program's flexibility, adaptability to meet diverse use cases and reduces the complexity that makes the system more efficient and less error-prone.

For more information on the Core program and plugins, please visit the official documentation [1]

The time and release schedule of the audits and the report is highlighted in Tab1.

Version Number	Release Date	Scope and Key Objectives
v -1.0.0	May 3, 2024	Initial Review of The Core Program
v -1.1.0	Aug 10, 2024	External Plugin Support

Tab1. Audit Milestones History

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 Raydium 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.

Metaplex has delivered these programs to Mad Shield at the following Github repositories.

Repository URL	https://github.com/metaplex-foundation/mpl-core/
Commit (start of audit)	cd53660fd2652836e07c4299611dc40fca96d8fd
Commit (end of v-1.0.0)	fa3c7f7f4675e2bceea64ca51dae2d32b8c6efe8
Commit (end of v-1.1.0)	3ce110483c01f2698e3f26301bfd9b1b87beaaec

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.

The standard supports the same features as the original mpl-token-metadata program, however, reshaped through the plugin structure. Regardless, it was straight-forward for us to assume invariants that the program must follow and potential pitfalls that it must avoid.

We applied a targeted testing approach using our in-house developed testing suite to scrutinize scenario use cases drawn from previous audits of the mpl-token-metadata. This suite specifically focuses on vulnerabilities such as account substitutions and authority counterfeiting, the prevalent Solana security bugs. We also targeted the test cases toward specific coreNFT functionalities such as plugin access violations and account resizing.

During our tests, we discovered several vulnerabilities, notably SHIELD_MC_01, where a collection authority, surprisingly non-existent on the asset, was still able to alter plugin data such as Attributes. This highlighted a major oversight in authority validation mechanisms. Another significant finding was SHIELD_MC_04, which involved the problematic scenario of nested collection features potentially misleading on-chain data and indexers about the true size of a collection, compromising data integrity.

Our review also covered the asset account creation process, where we found that the current implementation inadequately addressed scenarios where the target keypair already held a SOL balance described in SHIELD_MC_05.

Additionally, our consistency checks on programmable NFT specifications revealed that the implementation of the `allow_list` was insufficient as captured under SHIELD_MC_02 opening up the possibility for marketplaces/programs to avoid creator `rule_sets`.

Finally, we corroborate in SHIELD_MC_GR_01 that the program is in a nascent stage of development and relies heavily on specific assumptions in critical validation functions. In addition, the absence of some functionalities yet to be enabled by Metaplex is to be carefully evaluated before release.

4. Findings & Recommendations

Our severity classification system adheres to the criteria outlined here.

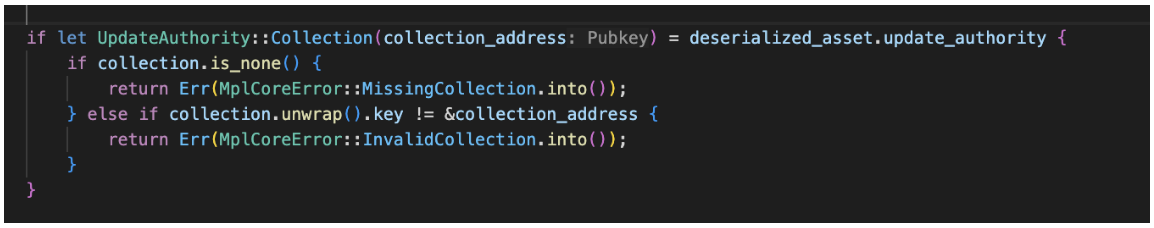
Severity Level	Exploitability	Potential Impact	Examples
Critical	Low to moderate difficulty, 3rd-party attacker	Irreparable financial harm	Direct theft of funds, permanent freezing of tokens/NFTs
High	High difficulty, external attacker or specific user interactions	Recoverable financial harm	Temporary 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_MC_01 [RESOLVED]	Collection Authority Overreach	*Critical*
SHIELD_MC_02 [RESOLVED]	Faulty Allow-List Logic	*Critical*
SHIELD_MC_03 [RESOLVED]	Incomplete Collection Authority Check Allowing Arbitrary Collection Size Modification	*Medium*
SHIELD_MC_04 [HAWKED]	Possibility Of Nested Collections	*Low*
SHIELD_MC_05 [HAWKED]	Asset Creation Fails If Account Is Pre Topped Up With Any Amount of SOL	*Low*
SHIELD_MC_GR_01 [ACKNOWLEDGED]	On the extra dependence on the validate_asset_permissions function	*Informational*

1. SHIELD_MC_01 - Collection Authority Overreach *[Critical]*

The collection authority feature may override attributes on ccNFTs without an assigned collection by merely specifying a collection. This bypasses the intended security checks and is a clear breach of the intended authority over the ccNFT where a non-existent collection authority can override the plugin details for a token that does not have a collection authority set. This is due to the incomplete collection check in the `validate_asset_permission` function.



```
if let UpdateAuthority::Collection(collection_address: Pubkey) = deserialized_asset.update_authority {
    if collection.is_none() {
        return Err(MplCoreError::MissingCollection.into());
    } else if collection.unwrap().key != &collection_address {
        return Err(MplCoreError::InvalidCollection.into());
    }
}
```

Recommended Fix

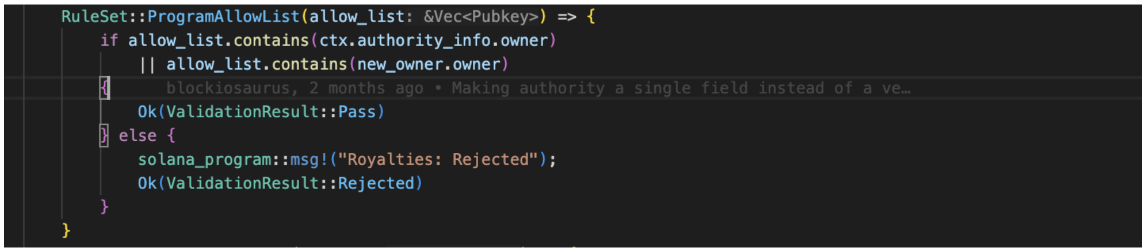
Add a validation check to ensure that the collection account is non-existent if the asset does not belong to a collection returning an error upon asset deserialization.

The patch commit: [b5c6d97f]()

2. SHIELD_MC_02 – Faulty Allow-List Logic [Critical]

The logic implemented to manage allow lists is incorrectly formulated. Specifically, the allow list should verify that both the destination and owner of the core NFT are controlled by the current program. However, the existing checks treat these conditions as mutually exclusive rather than concurrent requirements.

The condition also does not account for PDAs that are not owned i.e. assigned to the target program being allowed/disallowed yielding a discrepancy ƒrom the original mpl-token-metadata program.



```
RuleSet::ProgramAllowList(allow_list: &Vec<Pubkey>) => {
    if allow_list.contains(ctx.authority_info.owner)
        || allow_list.contains(new_owner.owner)
    {       blockiosaurus, 2 months ago • Making authority a single field instead of a ve…
        Ok(ValidationResult::Pass)
    } else {
        solana_program::msg!("Royalties: Rejected");
        Ok(ValidationResult::Rejected)
    }
}
```

The Metaplex team acknowledged that they are aware of this issue, however, they will take a stealth approach, monitoring on-chain transactions especially those of marketplaces to observe any potential ruleset violations as a fix involves account creation for the owner accounts proven to complicate interactions in the case of the mpl-token-metadata program.

Recommended Fix

Update the allow list verification logic to require both the destination and owner of the core NFT to be owned by the current program simultaneously.

The second issue is on the Hawk Deck and no actions are taken until further notice.

The patch commit: fa3c7f7f

3. **SHIELD_MC_03 – Incomplete Collection Authority Check allows Arbitrary Collection Size Modification [Medium]**

   Due to the incomplete collection authority check in the `validate_asset_permission` function (as described in SHIELD_MC_01), an attacker can arbitrarily change the size of a collection.

   This is possible because the current implementation does not verify if the asset being burned is truly part of the collection, allowing an attacker to manipulate the collection size without proper authorization.

   Recommended Fix

   The fix for SHIELD_MC_01 will also remove the possibility for exploit in the `burn` instruction, however, to avoid the excessive dependency on the `validate_asset_permission` that exhibits a single point of failure behavior (elaborated more extensively in SHIELD_MC_GR_01), we recommend adding checks on the asset's update authority similar to the `create` instruction.

   The patch commit – b5c6d97

4. **SHIELD_MC_04 – Possibility Of Nested Collections [Low]**

   It is possible to assign a collection as the update authority of another collection, creating nested collections which could complicate the authority hierarchy and meddle with the collection sizing and indexers.

   Recommended Fix

   Introduce constraints to prevent a collection from being its own update authority or that of another collection.

   The patch status – *Hawked*

5. SHIELD_MC_05 – Asset Creation Fails If Account Is Pre Topped Up With Any Amount of SOL [Low]

The create_account function used in the code snippet is a system call provided by the Solana runtime, which creates a new account and allocates space and transfers some enough SOL to the account to be rent-exempt. However, this function fails if the account has a balance greater than zero.

This could cause a DoS attack especially if the function is invoked to initialize a PDA from another contract if an attacker tops up the account with a non-zero SOL balance.

Recommended Fix

This implementation seems to be an active design decision to prevent users from using their wallets and other addresses with spendable SOL balance as MPL Core accounts.

However, To mitigate the DoS risk, it is recommended to use the `create_or_allocate_account_raw` routine provided in the mpl-utils crate. This function does check if there is a non-zero balance present and will use the alternative account creation mechanism using the allocate and assign instructions from the system program.

The patch status – *Hawked*

5. General Recommendations

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

1. **SHIELD_MC_GR_01 – On The Extra Dependence On The validate_asset_permissions Function**

   The consolidation of authority checking into the `validate_asset_permissions` function presents both strengths and risks. While this centralization simplifies the management of permissions and aligns with Rust's design philosophy by providing a clear, singular point of validation, it introduces a significant risk as a single point of failure.

   The absence of certain functionalities, such as burning, transferring, and plugin management for compressed assets, introduces yet another additional layer of complexity and potential risk. It is crucial that these incomplete features do not compromise the overall security and functionality of the Core program. Therefore, we suggest:

   - **Incremental Deployment –** As new functionalities are developed and ready for release, they should be deployed incrementally with thorough beta testing and limited user exposure until fully vetted. This allows for controlled observation of the features' behavior under real conditions without risking the entire platform.
   - **Perpetual Testing and Monitoring –** Ongoing testing and real-time monitoring of the system should be implemented to quickly detect and respond to any issues arising from newly introduced features. This includes automated systems to monitor the behavior of these features in live environments to ensure they perform as intended and do not introduce vulnerabilities.

6. External Plugins

Following the simple idea of allowing creators to have maximum freedom in defining the features and capabilities of their assets, the program introduces external plugins enabling such customization of the plugins outside the current base default plugins inherited from the legacy TM program. In the following we briefly describe the nature of the pathways defined for creators to exert more control over their digital artifacts.

1. **LifeCycleHook & LinkedLifeCycleHook**

   This plugin allows for a highly customizable validation process, where additional accounts can be included in the Cross-Program Invocation (CPI) call. These additional accounts can be explicitly listed or derived as Program Derived Addresses (PDAs) using the hooked program's public key. The hooked program, upon receiving the CPI call, provides a validation result along any new data that should be stored at a specific offset within the plugin's data structure.

   The LifecycleHook plugin is designed to be flexible and updatable. It allows for the configuration of extra accounts, updating of the schema used by the plugin, and setting an authority for data updates. The authority, once set, cannot be changed, ensuring that data integrity and access controls are maintained. The plugin abstains from validating certain actions directly, delegating this responsibility to the hooked program, thus providing a modular and extensible validation mechanism for lifecycle events within the MPL-Core framework.

2. **Oracle**

   The Oracle plugin in the MPL-Core program is designed to facilitate external validations for lifecycle events on specified accounts. It operates by referencing an oracle account, which can either be directly specified or derived using a program-derived address (PDA) based on a provided base address. This plugin is capable of validating various lifecycle events such as creation, transfer, burning, and updates of assets. The oracle account holds validation results, which are accessed based on a configurable offset within the account's data structure.

   To implement validations, the Oracle plugin leverages the `PluginValidation` trait, which defines methods for validating the addition of external plugin adapters and the specific lifecycle events. During validation, the plugin fetches the oracle account data, checks the relevant offset for validation results, and ensures the data is initialized and correctly formatted. Depending on the lifecycle event being validated, the plugin returns a corresponding validation result, ensuring that all actions conform to the expected validations specified in the oracle account.

3. **AppData & LinkedAppData**

   The `AppData` plugin provides a mechanism for storing and managing arbitrary application-specific data within a Solana account. This plugin designates a `data_authority` who is exclusively authorized to update the app data. Unlike the overall plugin authority stored in the `ExternalRegistryRecord`, the `data_authority` cannot change the plugin's metadata or revoke authority. The data managed by the plugin is stored at the plugin's data offset, which is located immediately after the plugin header in the account.

   The `AppData` plugin is configured to be flexible and adaptable. It allows the `data_authority` to update the schema for the stored data, ensuring that the structure of the data can evolve as needed. The plugin implements the `PluginValidation` trait but abstains from active validation during the addition of external plugin adapters or transfers, indicating that it does not enforce specific validation rules in these contexts. This setup makes the `AppData` plugin a versatile tool for developers needing to store and manage custom data within their digital assets.

Finally, the audit covered the new additions to the `UpdatePlugin` including two new features,

- Addition and Removal of assets from collections.
- Supporting additionalDelegates in the UpdatePlugin to allow for more granular and customizable access control.

7. Extended Findings & Recommendations

In this section, we enumerate the findings for the extension of the audit that includes the new features mentioned in section 6.

Finding	Description	Severity Level

SHIELD_MCEP_01 [RESOLVED]	update_external_plugin_adapter instruction does not use the correct validation function pointer	*Medium*
SHIELD_MCEP_02 [RESOLVED]	update_authority overreach to update external plugin adapters blocking the external_plugin_adapter authority	*Medium*
SHIELD_MCEP_GR_01 [ACKNOWLEDGED]	Separate plugin authority validation from plugin validity in the create instruction	*Informational*

1. **SHIELD_MCEP_01 – update_external_plugin_adapter instruction does not use the correct validation function pointer *[Medium]***

   The `update_external_plugin_adapter` instruction in the first pass of the program uses the default `check_update_plugin` and not the `check_update_external_plugin_adapter`. This in turn results in skipping the validation of the `external_plugin_adapters` where testing shows that creating a collection with external plugin adapters results in an error.

   Recommended Fix

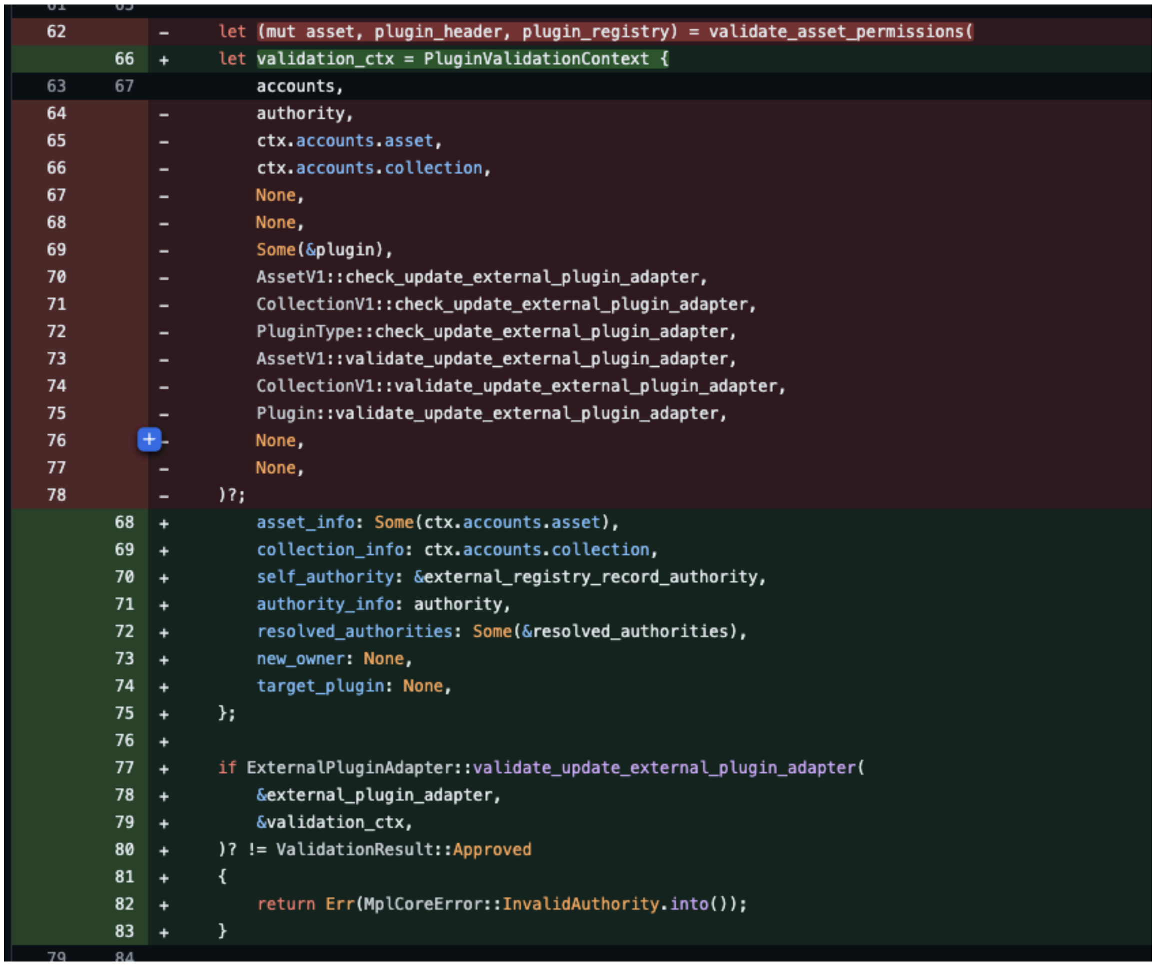
   To fix the issue, the development team first modified the `validate_asset_permission` function inputs to pass in the right function pointer aka the `check_update_external_plugin_adapter` and then to move forward with simplifying the validation by replacing it with an explicit invocation of the `validate_update_external_plugin_adapter` in the main body of the `update_external_plugin_adapter`. This is inline with SHIELD_MCEP_GR_01 as described in the next section.

   The patch commit: [1f0d87bc]()

2. **SHIELD_MCEP_02 – update_authority overreach to update external plugin adapters blocking the external_plugin_adapter authority *[Medium]***

   During our investigation of SHIELD_MCEP_02, it became evident that in the `update_external_plugin_adapter` instruction for both assets and collections, the validation checks were performed against the asset/collection's `update_authority` not the *external* plugin adapter's own authority. Although the asset/collection ability to update these plugins is not misplaced this prevents the `external_registry_record` authority from updating the external plugins which is its designated role.

To fix this issue, the authority check of both instructions were simplified (cf. SHIELD_MCEP_GR_0) to separate the context construction from the validation call, fetching the correct `external_registry_record.authority` and then calling the associated function to validate the authorities.



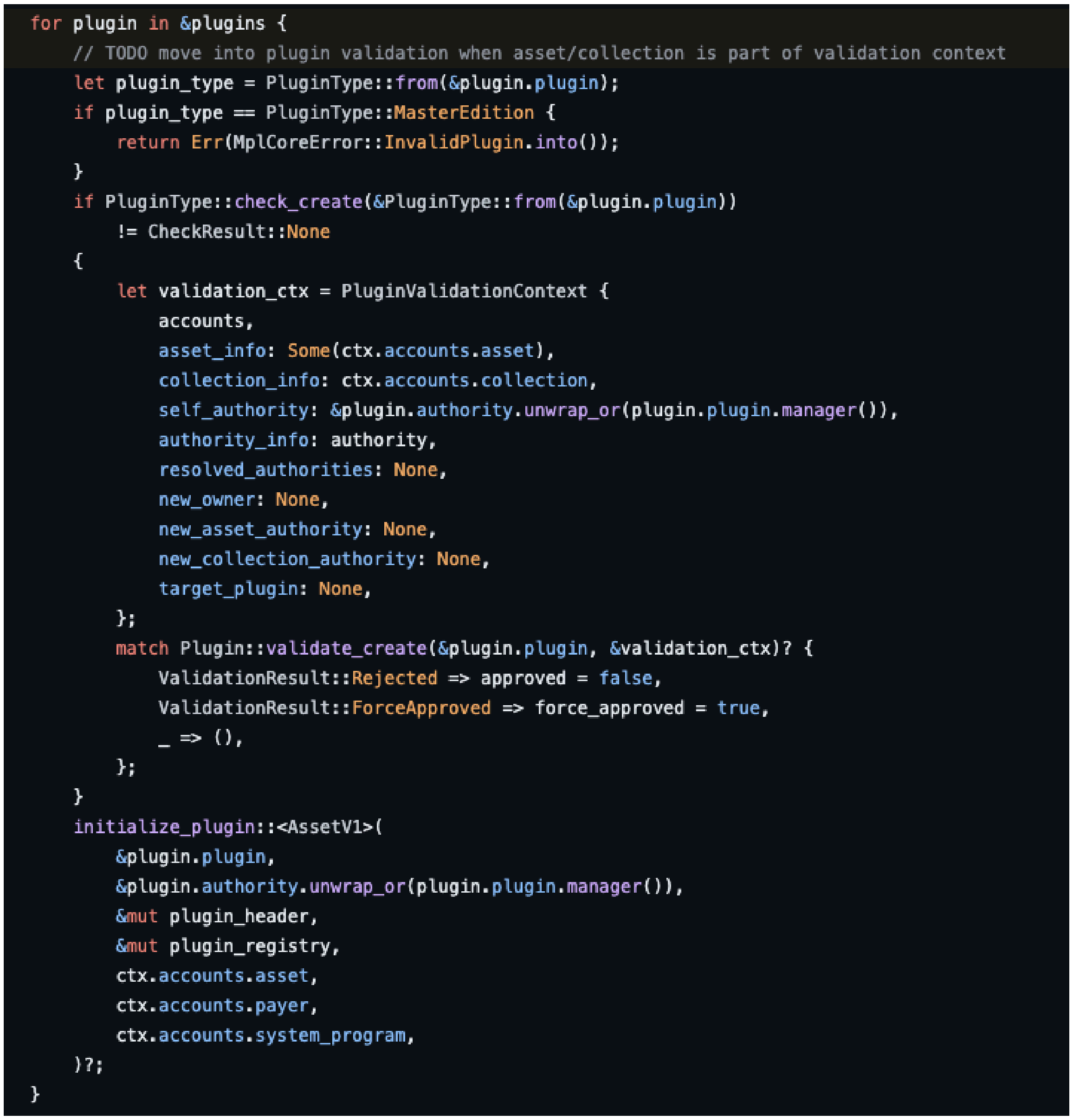
```
-   let (mut asset, plugin_header, plugin_registry) = validate_asset_permissions(
+   let validation_ctx = PluginValidationContext {
        accounts,
-       authority,
-       ctx.accounts.asset,
-       ctx.accounts.collection,
-       None,
-       None,
-       Some(&plugin),
-       AssetV1::check_update_external_plugin_adapter,
-       CollectionV1::check_update_external_plugin_adapter,
-       PluginType::check_update_external_plugin_adapter,
-       AssetV1::validate_update_external_plugin_adapter,
-       CollectionV1::validate_update_external_plugin_adapter,
-       Plugin::validate_update_external_plugin_adapter,
-       None,
-       None,
-   )?;
+       asset_info: Some(ctx.accounts.asset),
+       collection_info: ctx.accounts.collection,
+       self_authority: &external_registry_record_authority,
+       authority_info: authority,
+       resolved_authorities: Some(&resolved_authorities),
+       new_owner: None,
+       target_plugin: None,
+   };
+
+   if ExternalPluginAdapter::validate_update_external_plugin_adapter(
+       &external_plugin_adapter,
+       &validation_ctx,
+   )? != ValidationResult::Approved
+   {
+       return Err(MplCoreError::InvalidAuthority.into());
+   }
```

Fig 1: Reauthorize the update_external_plugins authority fix

The patch commit: PR#170

3. **SHIELD_MCEP_GR_01 – Separating plugin authority validation from correctness in the create instruction**

In the create instruction, the `validate_asset_permission` function is used to validate the plugins of the newly created asset inherited from the collection. Then the plugins for the newly created assets are validated one by one. This separation is recommended as the authority validation is independent from the plugins' correctness checks. Presently, the resolved authorities in the function are passed as `None` as can be seen in the code snippet below.



```rust
for plugin in &plugins {
    // TODO move into plugin validation when asset/collection is part of validation context
    let plugin_type = PluginType::from(&plugin.plugin);
    if plugin_type == PluginType::MasterEdition {
        return Err(MplCoreError::InvalidPlugin.into());
    }
    if PluginType::check_create(&PluginType::from(&plugin.plugin))
        != CheckResult::None
    {
        let validation_ctx = PluginValidationContext {
            accounts,
            asset_info: Some(ctx.accounts.asset),
            collection_info: ctx.accounts.collection,
            self_authority: &plugin.authority.unwrap_or(plugin.plugin.manager()),
            authority_info: authority,
            resolved_authorities: None,
            new_owner: None,
            new_asset_authority: None,
            new_collection_authority: None,
            target_plugin: None,
        };
        match Plugin::validate_create(&plugin.plugin, &validation_ctx)? {
            ValidationResult::Rejected => approved = false,
            ValidationResult::ForceApproved => force_approved = true,
            _ => (),
        };
    }
    initialize_plugin::<AssetV1>(
        &plugin.plugin,
        &plugin.authority.unwrap_or(plugin.plugin.manager()),
        &mut plugin_header,
        &mut plugin_registry,
        ctx.accounts.asset,
        ctx.accounts.payer,
        ctx.accounts.system_program,
    )?;
}
```

Fig 2: Plugin Validation In The Create Instruction

This will prevent collection plugin authorities to directly authorize an asset creation which is the desired behavior at this stage of the development. However, it is recommended to monitor the authorities capable of plugin creation carried through the collection authorities. This is inline with the approach to simplify the plugin validation as a rule of thumb to improve the auditability of the code.

5. Conclusion

The MPL-Core program exhibits a robust framework with numerous innovative features. However, the vulnerabilities identified need to be addressed to ensure the program can operate securely and efficiently within the digital assets ecosystem. Our recommendations aim to assist the development team in enhancing the program's security posture and operational stability.

The protocol evolves to enable more freedom in customizing collectibles, digital artifacts and in-game elements through the external plugin support. The audit is an ongoing effort between Metaplex and Mad Shield to ensure the security and desired behavior of the program as more applications and use cases exhibit these multifaceted new execution paths.

References

[1] Metaplex Developer Hub – https://developers.metaplex.com/core/
[2] MPL-Core Github – https://github.com/metaplex-foundation/mpl-core
[3] External Plugins Docs – https://developers.metaplex.com/core/external-plugins/overview
[4] Oracle Plugin Example – https://developers.metaplex.com/core/guides/oracle-plugin-example