Token22 Wrapper

Security Assessment

August 19th, 2024 — Prepared by OtterSec

Robert Chen r@osec.io

Table of Contents

01 — Executive Summary

Overview

Solend Protocol engaged OtterSec to assess the `token2022-wrapper` program. This assessment was conducted between July 30th and August 6th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 2 findings throughout this audit engagement.

In particular, we identified a high-risk vulnerability, where the program improperly checks for initialization by verifying lamports instead of the account owner and lacks strict account checks (OS-SPL-ADV-00). Additionally, the account creation functionality fails to correctly handle accounts with non-zero lamports (OS-SPL-ADV-01).

Scope

The source code was delivered to us in a Git repository at https://github.com/solendprotocol/token2022-wrapper. This audit was performed against commit 53e8dcd.

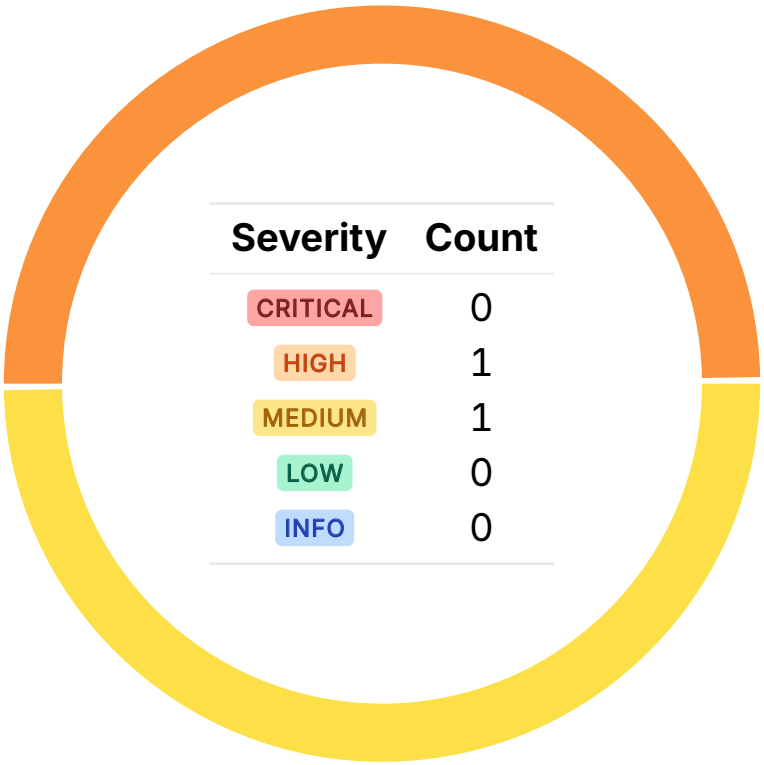
A brief description of the programs is as follows:

Name	Description
token2022-wrapper	A program to wrap Token2022 Solana tokens into SPL tokens.

02 — Findings

Overall, we reported 2 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	1
MEDIUM	1
LOW	0
INFO	0

03 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-SPL-ADV-00	HIGH	RESOLVED	The program improperly checks for initialization by verifying lamports instead of the account owner and lacks adequate account checks.
OS-SPL-ADV-01	MEDIUM	RESOLVED	`create_account` does not correctly handle accounts with non-zero lamports.

Incorrect Account Initialization Check HIGH OS-SPL-ADV-00

Description

The vulnerability is related to the practice of utilizing lamports to check for the initialization status of certain accounts (`assert_is_account_uninitialized` , `process_deposit_and_mint_wrapper_tokens`). This approach may be vulnerable to a Denial of Service (DoS) attack, manipulating the lamport balance to disrupt functionality by restricting initialization. Thus, if the lamport balance is insufficient then it may hinder account initialization. The ownership of an account is more reliable for checking initialization status. This field is less susceptible to manipulation compared to lamport balance.

Additionally, to improve overall security, implement stricter account checks in the codebase. Ensure that necessary accounts such as the `token_2022_program` , all mints, and the user authority are signers.

Remediation

Ensure that the account's owner field is set to the appropriate program ID. This is a more robust method for determining whether an account is initialized and correctly owned by the intended program.

Patch

Resolved in 41f7878 and 44db717.

Account Creation With Non-zero Lamports MEDIUM OS-SPL-ADV-01

Description

Currently, the program utilizes `create_account` for account creation. This may not be suitable in all scenarios, particularly when dealing with accounts that already have non-zero lamports. If an account already has lamports, it is considered initialized. In this scenario, the account already has some state, and `create_account` should not be utilized to reinitialize it. In the Anchor framework, `generate_create_account` is designed to handle both zero and non-zero lamport scenarios more gracefully.

>_ *src/utils/system_utils.rs* RUST

```rust
pub fn create_account<'a, 'info>(
    [...]
) -> ProgramResult {
    let current_lamports = **new_account.try_borrow_lamports()?;
    if current_lamports == 0 {
        // If there are no lamports in the new account, we create it with the create_account
            ↪ instruction
        invoke_signed(
            &system_instruction::create_account(
                payer.key,
                new_account.key,
                rent.minimum_balance(space as usize),
                space,
                program_owner,
            ),
            &[payer.clone(), new_account.clone(), system_program.clone()],
            &[seeds
                .iter()
                .map(|seed| seed.as_slice())
                .collect::<Vec<&[u8]>>()
                .as_slice()],
        )
    }[...]
}
```

For accounts with non-zero lamports, `generate_create_account` avoids reinitializing the account and instead allocates additional space (`system_instruction::allocate`) and assigns the account to the correct program. Furthermore, to deserialize account data from a byte buffer utilize `StateWithExtensions::<spl_token_2022::state::Account>::unpack` . This approach aligns with Anchor's `try_deserialize_unchecked`, providing a consistent way to deserialize account data.

Remediation

Utilize the Anchor framework (`generate_create_account`) instead of relying on `system_utils::create_account` for account creation with non-zero lamports. Also utilize `StateWithExtensions::<spl_token_2022::state::Account>::unpack` to deserialize account data.

Patch

Resolved in 8e13457 and 4551185.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.