SOLANA FOUNDATION

Token22

Audit

Presented by:

OtterSec contact@osec.io

Akash Thota 0x4ka5h@osec.io

Tuyết Dương tuyet@osec.io

Robert Chen r@osec.io

Contents

01 | Executive Summary

Overview

Solana Labs engaged OtterSec to perform an assessment of the `token22` program. This assessment was conducted between October 8th and November 3rd, 2023. For more information on our auditing methodology, see Appendix B.

Key Findings

Over the course of this audit engagement, we produced 5 findings in total.

In particular, we discovered a vulnerability improper structuring of the order of the accounts results in the assignment of incorrect accounts (OS-SPL-ADV-00). Furthermore, we highlighted an issue related to the absence of validation to confirm that the multi-signature account is a signer account, which may result in transaction failures (OS-SPL-ADV-02).

We also made recommendations around the insufficient Validation Of Decryptable Balance (OS-SPL-SUG-01) and advised against the use of multi-signature accounts with writable access (OS-SPL-SUG-00).

02 | Scope

The source code was delivered to us in a git repository at github.com/solana-labs/solana-program-library/tree/master/token/program-2022. This audit was performed against commit c141fad and e924132.

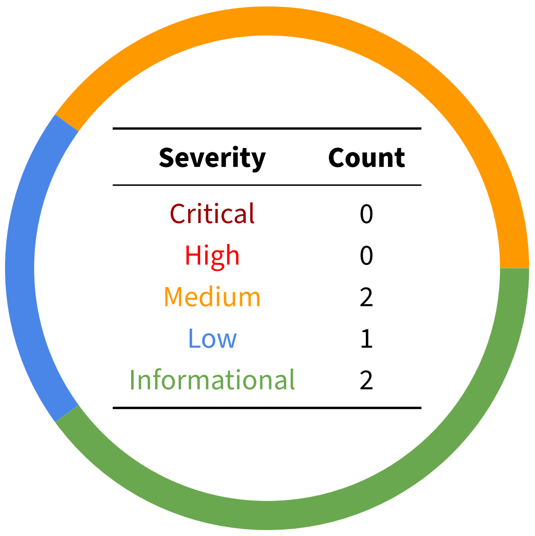
A brief description of the programs is as follows:

Name	Description
token22	A token program operating on the Solana blockchain, suitable for managing fungible and non-fungible tokens. It provides an interface and implementation that third parties may employ to establish and utilize their tokens.

03 | Findings

Overall, we reported 5 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	0
Medium	2
Low	1
Informational	2

04 | 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	Medium	Resolved	Improper structuring of the order of the accounts results in the assignment of incorrect accounts.
OS-SPL-ADV-01	Medium	Resolved	`process_approve_account` does not verify whether the mint associated with `token_account` matches the provided mint.
OS-SPL-ADV-02	Low	Resolved	Insufficient validation checks to verify that the multi-signature account is a signer account, resulting in transaction failures.

OS-SPL-ADV-00 [med] | Incorrect Account Ordering

Description

`transfer_with_split_proofs` assembles a list of seven `AccountMeta` objects, which provide metadata about accounts involved in the transaction. `process_transfer` iterates through these accounts; however, these accounts are structured incorrectly. In `verify_transfer_proof`, `account_info_iter` iterates through the first three accounts out of the seven accounts listed in `transfer_with_split_proofs`.

confidential_transfer/verify_proof.rs RUST

```rust
pub fn verify_transfer_proof(
    [...]
) -> Result<Option<TransferProofContextInfo>, ProgramError> {
    // The first three accounts are loaded here
        [...]
        if close_split_context_state_on_execution {
            let lamport_destination_account_info =
                ↪  next_account_info(account_info_iter)?;
            let context_state_account_authority_info =
                ↪  next_account_info(account_info_iter)?;

            msg!("Closing equality proof context state account");
            invoke(
                &zk_token_proof_instruction::close_context_state(
                    ContextStateInfo {
                        context_state_account:
                            ↪  equality_proof_context_state_account_info.key,
                        context_state_authority:
                            ↪  context_state_account_authority_info.key,
                    },
                    lamport_destination_account_info.key,
                ),
                &[
                    equality_proof_context_state_account_info.clone(),
                    lamport_destination_account_info.clone(),
                    context_state_account_authority_info.clone(),
                ],
            )?;
            //Similarly, cipher text validity proof and range proof context state
                ↪  accounts are also closed
        [...]
}
```

In the instance, `no_op_on_split_proof_context_state` is false, and `close_split_context_state_on_execution` is true, execution enters into the `if` condition and `account_info_iter` loads the next account into `lamport_destination_account_info` to load `lamport_destination`, however, due to the incorrect ordering of accounts, `source_account_authority` is loaded instead, which is a signer account.

Therefore, during the closure of each split context state account, it employs `lamport_destination_account_info` as the destination for lamport transfers, which currently references `source_account_authority`. This, in essence, results in the transfer of lamports to the signing authority of the source account, a behavior that was not the intended outcome. Note that a similar issue is present when transferring with fees.

Furthermore, if `no_op_on_split_proof_context_state` is set to true and a necessary context state account has not been initialized, the invocation of `verify_transfer_proof` results in a return of None as demonstrated below. Consequently, in the event that `close_split_context_state_on_execution` is true, `process_transfer` clears `lamport_destination`, `context_accounts.authority`, and `zk_token_proof_program` to load `source_account_authority` into `authority_info`.

confidential_transfer/verify_proof.rs RUST

```rust
pub fn verify_transfer_proof(
    [...]
) -> Result<Option<TransferProofContextInfo>, ProgramError> {
        [...]
        if no_op_on_split_proof_context_state
            && check_system_program_account
            (equality_proof_context_state_account_info.owner).is_ok()
        {   return Ok(None);    }
        [...]
}
```

However, due to the incorrect ordering of accounts, `source_account_authority`, `lamport_destination` and `context_accounts.authority` accounts are flushed out and `authority_info` is assigned `zk_token_proof_program` account.

confidential_transfer/processor.rs RUST

```rust
fn process_transfer(
    [...]
) -> ProgramResult {
    [...]
        if close_split_context_state_on_execution && maybe_proof_context.is_none()
            ↪  {
            let _lamport_destination_account_info =
                ↪   next_account_info(account_info_iter)?;
            let _context_state_authority_info =
                ↪   next_account_info(account_info_iter)?;
            let _zk_token_proof_program_info =
                ↪   next_account_info(account_info_iter)?;
        }
        let authority_info = next_account_info(account_info_iter)?;
    [...]
}
```

Thus, the incorrect authority may fail to verify the zero-knowledge proof, which is a critical part of ensuring the correctness of the transfer. If the proof fails or the verification process does not execute as intended, the transaction may result in an error and failure of the transfer.

Remediation

Modify the conditions in `process_transfer`, `verify_transfer_with_fee_proof`, `verify_transfer_proof` and the structuring of accounts in `transfer_with_split_proofs`, such that the new conditions accurately reflect the order of accounts as listed in `transfer_with_split_proofs`.

Patch

Fixed in 5931

OS-SPL-ADV-01 [med]| Lack Of Mint Account Verification

Description

`process_approve_account` ensures that a given token account is configured correctly for confidential transfers by authorizing the account's authority. However, it lacks a crucial check to validate that the associated mint of the token account (`token_account.base.mint`) matches the mint for which the confidential transfer approval is being granted (`mint_info.key`).

confidential_transfer/processor.rs RUST

```rust
/// Processes an [ApproveAccount] instruction.
fn process_approve_account(accounts: &[AccountInfo]) -> ProgramResult {
    let account_info_iter = &mut accounts.iter();
    let token_account_info = next_account_info(account_info_iter)?;
    let mint_info = next_account_info(account_info_iter)?;
    let authority_info = next_account_info(account_info_iter)?;
    [...]
}
```

Thus, this enables the approval of a token account for confidential transfers, even if it is associated with a different mint. Ideally, token accounts should only be allowed to hold tokens from the specific mint they are associated with. By not checking the mint consistency, the function effectively approves arbitrary token accounts for confidential transfers. Such unauthorized token mixing may have security and financial implications, as it could result in loss of value or assets for users who rely on the token system's integrity.

Remediation

Incorporate the following verification within `process_approve_account` to confirm that the token account's associated mint aligns with the mint for which the confidential transfer approval is sought.

confidential_transfer/processor.rs RUST

```rust
if token_account.base.mint != *mint_info.key {
    return Err(TokenError::MintMismatch.into());
}
```

Patch

Fixed in 5901

OS-SPL-ADV-02 [low] | Missing Signer Check On Accounts

Description

The vulnerability pertains to the management of multi-signature accounts when constructing withdrawal instructions for withheld tokens within the confidential transfer fee extension in both `withdraw_withheld_tokens_from_mint` and `withdraw_withheld_tokens_from_accounts`.

confidential_transfer_fee /instruction.rs RUST

```rust
pub fn inner_withdraw_withheld_tokens_from_mint(
    [...]
    new_decryptable_available_balance: &DecryptableBalance,
    authority: &Pubkey,
    multisig_signers: &[&Pubkey],
    proof_data_location: ProofLocation<CiphertextCiphertextEqualityProofData>,
) -> Result<Instruction, ProgramError> {
    [...]
    for multisig_signer in multisig_signers.iter() {
        accounts.push(AccountMeta::new(**multisig_signer, false));
    }
    [...]
}
```

The problem concerns the handling of multi-signature accounts as non-signer accounts within `inner_withdraw_withheld_tokens_from_mint` and `inner_withdraw_withheld_tokens_from_accounts`. In the context of the withdrawal operation, the program should recognize multi-signature accounts as signers, but currently, it treats them incorrectly as non-signers. This may result in transaction failures when multiple signers are involved, primarily during the owner check for multi-signature accounts, as shown below:

confidential_transfer_fee /processor.rs RUST

```rust
fn process_withdraw_withheld_tokens_from_mint(
    [...]
) -> ProgramResult {
    [...]
    Processor::validate_owner(
        program_id,
        &withdraw_withheld_authority,
        authority_info,
        authority_info_data_len,
        account_info_iter.as_slice(),
    )?;
    [...]
}
```

Proof of Concept

1. There exists a multi signature account named `multisig_account` configured to require three signers: `signer1`, `signer2`, and `signer3`. The purpose of this multi-signature account is to authorize the withdrawal of withheld tokens.
2. When creating the withdrawal instruction through `inner_withdraw_withheld_tokens_from_mint`, `multisig_account` is specified as a read-only, non-signer account.
3. In a transaction, if `signer1`, `signer2`, and `signer3` provide their signatures to authorize the withdrawal, the Solana runtime may fail to recognize their authorization due to the way the instruction was constructed.
4. Consequently, this may result in transaction failure as the code incorrectly categorizes the multi-signature account as a non-signer account.

Remediation

Designate multi-signature accounts as signers for authorizing transactions.

Patch

Fixed in 5900

05 | General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-SPL-SUG-00	Multi-signature accounts with write access in situations where it is unnecessary.
OS-SPL-SUG-01	The capability to alter `decryptable_available_balance` allows for manipulation of this value and initiations of unauthorized withdrawals.

OS-SPL-SUG-00 | Writable Access To Accounts

Description

This issue concerns the unnecessary use of writable access for multi-signature accounts in `Processor::validate_owner` within `process_withdraw_withheld_tokens_from_accounts` and `process_withdraw_withheld_tokens_from_mint` when validating authority. Multi-signature accounts typically are not required to be writable during the validation process.

confidential_transfer_fee /processor.rs RUST

```rust
fn process_withdraw_withheld_tokens_from_accounts(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    num_token_accounts: u8,
    new_decryptable_available_balance: &DecryptableBalance,
    proof_instruction_offset: i64,
) -> ProgramResult {
    [...]
    Processor::validate_owner(
        program_id,
        &withdraw_withheld_authority,
        authority_info,
        authority_info_data_len,
        &account_infos[..num_signers],
    )?;
    [...]
```

Remediation

Modify the accounts to grant them read-only access.

Patch

Fixed in 5900

OS-SPL-SUG-01 | Insufficient Validation Of Decryptable Balance

Description

The vulnerability concerns the `decryptable_available_balance` field in the `process_withdraw_withheld_tokens_from_accounts`. The `decryptable_available_balance` field represents the balance the authority may decrypt.

confidential_transfer_fee /processor.rs RUST

```rust
fn process_withdraw_withheld_tokens_from_accounts(
    [...]
) -> ProgramResult {
    [...]
    destination_confidential_transfer_account.decryptable_available_balance =
        *new_decryptable_available_balance;
    [...]
}
```

The issue occurs as the authority may update `decryptable_available_balance` without any prior validation or check. Specifically, suppose the instruction is not constructed utilizing the client program. In that instance, the authority may manipulate this field by initiating a withdrawal and passing an arbitrary `decryptable_available_balance` value, allowing them to withdraw more tokens than authorized, creating discrepancies between the `decryptable_available_balance` and the true available balance.

Remediation

Implement a check to validate that the passed decryptable available balance is the same as the available balance in the client utilizing advanced encryption standard keys and ElGamal keypairs.

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 section.

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.
Informational	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 the other 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.