HALBORN

Solana - token-2022

Solana Program Security Assessment

Prepared by: **Halborn**

Date of Engagement: **January 10th, 2024 - March 8th, 2024**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	03/11/2024
0.2	Draft Version	03/11/2024
1.0	Remediation Plan	03/11/2024
1.1	Remediation Plan Review	03/11/2024

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

spl-token-2022 aka Token extensions (TE) is a new token program on the Solana blockchain that enables a set of modular extensions for token issuers. These extensions are built into the core protocol level of Solana and apply to both fungible and non-fungible tokens.
With token extensions, developers can now use a set of more than a dozen proven, audited extensions, that quickly add the needed advanced functionality to their tokens, such as the privacy-preserving technology of confidential transfers, new compliance frameworks such as transfer hooks, and the ability to charge fees on transfers.

Halborn conducted a security assessment on a set of changes to the program made between two different commits, beginning on January 10th, 2024 and ending on March 8th, 2024 . The security assessment was scoped to the updates to the master branch of the spl-token-2022 GitHub repository. Commit hashes and further details can be found in the **Scope** section of this report.

1.2 ASSESSMENT SUMMARY

The team at Halborn was provided 7 weeks for the engagement and assigned 1 full-time security engineer to review the security of the programs in scope. The security engineer is a blockchain and Solana Program security expert with advanced penetration testing and Solana Program hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to identify potential security issues within the programs.

In summary, Halborn did not identify any **new** significant issues in the code in scope. All reported issues have already been fixed by the Solana team and were included in the report for the sake of continuity of the audit commit history.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of a manual review of the source code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the program assessment. While manual testing is recommended to uncover flaws in business logic, processes, and implementation; automated testing techniques help enhance coverage of programs and can quickly identify items that do not follow security best practices.

The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture, purpose, and use of the platform.
- Manual program source code review to identify business logic issues.
- Mapping out possible attack vectors
- Thorough assessment of safety and usage of critical Rust variables and functions in scope that could lead to arithmetic vulnerabilities.

- Scanning dependencies for known vulnerabilities (cargo audit).
- Local runtime testing (solana-test-framework)

2. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

2.1 EXPLOITABILITY

Attack Origin (AO):

Captures whether the attack requires compromising a specific account.

Attack Cost (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

Attack Complexity (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

Metrics:

Exploitability Metric (m_E)	Metric Value	Numerical Value
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

2.2 IMPACT

Confidentiality (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

Integrity (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

Availability (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

Deposit (D):

Measures the impact to the deposits made to the contract by either users or owners.

Yield (Y):

Measures the impact to the yield generated by the contract for either users or owners.

Metrics:

Impact Metric (m_I)	Metric Value	Numerical Value
Confidentiality (C)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

2.3 SEVERITY COEFFICIENT

Reversibility (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

Scope (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

Severity	Score Value Range
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

2.4 SCOPE

Code repositories:

1. Token Extensions

- Repository: spl-token-2022
- Commit Range:
  - initial: 4587da1
  - final: 56aaa67
- Programs in scope:
  1. spl-token-2022 (token/program-2022)

Out-of-scope:
- third-party libraries and dependencies
- financial-related attacks

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
2	0	4	1	1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) INFLATING SUPPLY ON CONFIDENTIAL TRANSFERS	Critical (10)	SOLVED - 12/3/2022
(HAL-02) INFLATING SUPPLY ON ENCRYPTED BALANCE WITHDRAW	Critical (10)	SOLVED - 10/27/2022
(HAL-03) SILENT TOKEN BURN ON EMPTY ACCOUNT	Medium (6.7)	SOLVED - 10/27/2022
(HAL-04) MINT ADDRESS VERIFICATION MISSING ON APPROVE	Medium (6.2)	SOLVED - 11/29/2023
(HAL-05) INCORRECT ACCOUNT ORDER	Medium (5.0)	SOLVED - 12/2/2023
(HAL-06) CONFIDENTIAL TRANSFER AMOUNTS INFO LEAK VIA TRANSFER FEES	Medium (5.0)	SOLVED - 10/28/2022
(HAL-07) UNCONSTRAINED CONFIDENTIAL TRANSFER FEE WITHDRAW	Low (2.5)	SOLVED - 10/28/2022
(HAL-08) MULTISG SIGNERS NOT REFERENCED CORRECTLY BY THE INSTRUCTION BUILDER	Informational (0.0)	SOLVED - 10/27/2022

FINDINGS & TECH DETAILS

4.1 (HAL-01) INFLATING SUPPLY ON CONFIDENTIAL TRANSFERS - CRITICAL(10)

Description:

Transfers between confidential accounts necessitate a zero-knowledge (ZK) proof that verifies two conditions: the source account has a balance greater than the amount being transferred, and the amount being transferred is non-negative. These transactions are executed through two distinct operations. The initial operation involves a ZK proof that validates the legitimacy of the transfer without revealing any details about the account balances or the amount transferred. The subsequent operation handles the computational aspects and updates the account states to finalize the transfer.

The verification of the ZK proof is carried out by a dedicated built-in program, which checks the proof's correctness. If the proof fails to validate, the operation is aborted. Specifically, the ZK proof comprises an equation with variables representing the states of the accounts participating in the transaction.

The execution of the second operation falls to the token program, which ensures the presence and integrity of the ZK proof and its consistency with the current account states, thus linking the proof to the blockchain's state.

However, the token program overlooks the full verification of the ZK proof inputs. It neglects the `new_source_ciphertext` field within the ZK proof, which is supposed to hold the encrypted balance of the source account after the transaction. The omission of this check means the encrypted balance of the source account is not verified, disconnecting the ZK proof from the actual balance in the source account.

Code Location:

Listing 1: solana-zk-token-sdk/src/instruction/transfer.rs (Line 55)

```
42 #[derive(Clone, Copy, Pod, Zeroable)]
43 #[repr(C)]
44 pub struct TransferData {
45     /// Group encryption of the low 32 bits of the transfer amount
46     pub ciphertext_lo: pod::TransferAmountEncryption,
47
48     /// Group encryption of the high 32 bits of the transfer
   ↳ amount
49     pub ciphertext_hi: pod::TransferAmountEncryption,
50
51     /// The public encryption keys associated with the transfer:
   ↳ source, dest, and auditor
52     pub transfer_pubkeys: pod::TransferPubkeys,
53
54     /// The final spendable ciphertext after the transfer
55     pub new_source_ciphertext: pod::ElGamalCiphertext,
56
57     /// Zero-knowledge proofs for Transfer
58     pub proof: TransferProof,
59 }
```

Listing 2: src/extension/confidential_transfer/processor.rs

```
622 #[allow(clippy::too_many_arguments)]
623 #[cfg(feature = "zk-ops")]
624 fn process_source_for_transfer(
625     program_id: &Pubkey,
626     token_account_info: &AccountInfo,
627     mint_info: &AccountInfo,
628     authority_info: &AccountInfo,
629     signers: &[AccountInfo],
630     source_encryption_pubkey: &EncryptionPubkey,
631     source_ciphertext_lo: &EncryptedBalance,
632     source_ciphertext_hi: &EncryptedBalance,
633     new_source_decryptable_available_balance: DecryptableBalance,
634 ) -> ProgramResult {
```

Proof of Concept:

To exploit the described vulnerability, multiple transfers that cumulatively exceed the encrypted balance of the source account can be performed. The vulnerability can be demonstrated through a proof-of-concept exploit that involves a transaction with several instructions for transfers, all linked to a single instruction that includes the ZK proof.

As a result of this exploit, the encrypted balance of the source account underflows and fails silently, rendering it invalid, while the encrypted pending balance of the destination account receives credits multiple times. This anomaly effectively generates tokens from nothing, leading to an increase in the token supply which is not reflected in the `total_supply` of the mint account. Consequently, the destination account can integrate the unjustly acquired balance and utilize it as if it was legitimately obtained.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:C/Y:N/R:N/S:U (10)

Recommendation:

When transferring, ensure the encrypted balance of the source account is aligned with the expected amount included in the ZK argument as `new_source_ciphertext`.

Remediation Plan:

SOLVED: The Solana team solved this issue in commit `c7fbd4b`.

4.2 (HAL-02) INFLATING SUPPLY ON ENCRYPTED BALANCE WITHDRAW - CRITICAL(10)

Description:

The transaction for a confidential withdrawal is divided into two parts: one for verifying the ZK proof through a specialized built-in program (which aborts the transaction if the proof is invalid), and another for the SPL Token 2022 program to process the balance operations and execute the withdrawal. The program checks for the presence of the ZK proof and its inputs' alignment with the account states, thereby linking the proof to the blockchain's current state.

However, the program fails to verify that the public key associated with the ZK proof matches the public key of the source account's encrypted balance. This oversight allows an attacker to forge a ZK proof to authorize any withdrawal amount, irrespective of the source account's actual encrypted balance.

By exploiting this flaw, an attacker could withdraw an unlimited amount of tokens to their plaintext balance, effectively creating tokens from thin air and causing inflation of the token supply. This inflation would not be recorded in the mint account associated with the token. The inflated plaintext balance becomes spendable like any legitimate account balance. While full exploitability of this vulnerability has not been confirmed, the Solana team acknowledges that forging a deceptive ZK proof is likely feasible.

Code Location:

Listing 3: (Line 90)

```
84 pub struct ConfidentialTransferAccount {
85     /// `true` if this account has been approved for use. All
 ↳ confidential transfer operations for
86     /// the account will fail until approval is granted.
87     pub approved: PodBool,
88
89     /// The public key associated with ElGamal encryption
90     pub encryption_pubkey: EncryptionPubkey,
91
92     /// The low 16 bits of the pending balance (encrypted by `
 ↳ encryption_pubkey`)
93     pub pending_balance_lo: EncryptedBalance,
94
95     /// The high 48 bits of the pending balance (encrypted by `
 ↳ encryption_pubkey`)
96     pub pending_balance_hi: EncryptedBalance,
97
98     /// The available balance (encrypted by `encrypiton_pubkey`)
99     pub available_balance: EncryptedBalance,
100
101     /// The decryptable available balance
102     pub decryptable_available_balance: DecryptableBalance,
103
104     /// `pending_balance` may only be credited by `Deposit` or `
 ↳ Transfer` instructions if `true`
105     pub allow_balance_credits: PodBool,
106
107     /// The total number of `Deposit` and `Transfer` instructions
 ↳ that have credited
108     /// `pending_balance`
109     pub pending_balance_credit_counter: PodU64,
110
111     /// The maximum number of `Deposit` and `Transfer`
 ↳ instructions that can credit
112     /// `pending_balance` before the `ApplyPendingBalance`
 ↳ instruction is executed
113     pub maximum_pending_balance_credit_counter: PodU64,
114
115     /// The `expected_pending_balance_credit_counter` value that
 ↳ was included in the last
116     /// `ApplyPendingBalance` instruction
```

```
117     pub expected_pending_balance_credit_counter: PodU64,
118
119     /// The actual `pending_balance_credit_counter` when the last
 ↳ `ApplyPendingBalance` instruction
120     /// was executed
121     pub actual_pending_balance_credit_counter: PodU64,
122
123     /// The withheld amount of fees. This will always be zero if
 ↳ fees are never enabled.
124     pub withheld_amount: EncryptedWithheldAmount,
125 }
```

Listing 4: asd (Line 36)

```
34 pub struct WithdrawData {
35     /// The source account ElGamal pubkey
36     pub pubkey: pod::ElGamalPubkey, // 32 bytes
37
38     /// The source account available balance *after* the withdraw
 ↳ (encrypted by
39     /// `source_pk`
40     pub final_ciphertext: pod::ElGamalCiphertext, // 64 bytes
41
42     /// Range proof
43     pub proof: WithdrawProof, // 736 bytes
44 }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:C/Y:N/R:N/S:U (10)

Recommendation:

Ensure to verify that `ConfidentialTransferAccount.encryption_pubkey` matches `WithdrawData.pubkey` when processing withdraws.

Remediation Plan:

SOLVED: The Solana team solved this issue in commit `94b912a`.

4.3 (HAL-03) SILENT TOKEN BURN ON EMPTY ACCOUNT - MEDIUM (6.7)

Description:

In the token-2022 program, an account holding tokens can only be closed if its balance is zero, a rule that applies both to the plaintext balance and to encrypted balances managed by the confidential transfer feature. To close an account with encrypted balances, a specific instruction named EmptyAccount must be executed. This instruction requires the verification of a zero-knowledge (ZK) proof demonstrating that the account's balance is indeed zero.

As with other operations involving confidential tokens, the ZK proof must be included within an instruction in the transaction that triggers the EmptyAccount instruction handler. The handler verifies the existence of the ZK proof and its correlation with the blockchain's current state.

However, the verification process fails to ensure that the public key linked to the ZK proof matches the public key of the token account intended for closure. This oversight could enable an attacker to present a fabricated ZK proof, erroneously indicating that the account balance is zero, thereby permitting the closure of an account with a nonzero balance.

Closing an account in this manner could lead to a reduction in the circulating token supply without the corresponding supply tracker in the mint account being updated. It is worth to note an attacker would forfeit their balance by performing this action, making the practical incentive for such an attack unclear.

Code Location:

Listing 5: solana-zk-token-sdksrc/instruction/close_account.rs (Line 29)

```
27 pub struct CloseAccountData {
28     /// The source account ElGamal pubkey
29     pub pubkey: pod::ElGamalPubkey, // 32 bytes
30
31     /// The source account available balance in encrypted form
32     pub ciphertext: pod::ElGamalCiphertext, // 64 bytes
33
34     /// Proof that the source account available balance is zero
35     pub proof: CloseAccountProof, // 64 bytes
36 }
```

BVSS:

AO:A/AC:M/AX:L/C:N/I:N/A:N/D:C/Y:N/R:N/S:U (6.7)

Recommendation:

Ensure to verify that `ConfidentialTransferAccount.encryption_pubkey` matches `CloseAccountData.pubkey` when processing account close.

Remediation Plan:

SOLVED: The Solana team solved this issue in commit d6a72eb.

4.4 (HAL-04) MINT ADDRESS VERIFICATION MISSING ON APPROVE - MEDIUM (6.2)

Description:

The process_approve_account function is designed to prepare a token account for confidential transfers by authorizing its account authority. However, it omits an essential verification step to ensure that the mint associated with the token account corresponds to the mint intended for the confidential transfer approval.

As a consequence, the system permits the approval of token accounts for confidential transfers, even when those accounts belong to a different mint than the one being authorized. Token accounts are expected to hold tokens only from their associated mint. The absence of a check for mint alignment allows for the approval of any token account for confidential transfers, disregarding this principle.

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:M/D:N/Y:N/R:N/S:U (6.2)

Recommendation:

Modify the conditions in the process_transfer, verify_transfer_with_fee_proof and verify_transfer_proof functions, and amend the ordering of accounts in the transfer_with_split_proofs function such that the new conditions accurately reflect the order of accounts as required by the transfer_with_split_proofs instruction handler.

Remediation Plan:

SOLVED: The Solana team solved this issue in commit b410945.

4.5 (HAL-05) INCORRECT ACCOUNT ORDER - MEDIUM (5.0)

Description:

The transfer_with_split_proofs function in the token-2022 program compiles a sequence of seven AccountMetas to detail the accounts participating in a transaction. The process_transfer instruction handler iterates over these accounts, and the verify_transfer_proof function iterates over the first three of the seven accounts required by the transfer_with_split_proofs function.

When the no_op_on_split_proof_context_state flag is set to false and the close_split_context_state_on_execution flag is set to true, the process enters a conditional block where account_info_iter is supposed to fetch the next account as lamport_destination_account_info for lamport transfers. However, due to the misalignment of account order, source_account_authority is loaded instead of the intended lamport destination. This account is a signer account.

As a result, when closing each split context state account, the system mistakenly uses lamport_destination_account_info, now referencing source_account_authority, as the recipient for lamport transfers. This unintentionally directs lamports to the signing authority of the source account, diverging from the expected behavior. A similar problem occurs during transfers that involve fees.

Additionally, if the no_op_on_split_proof_context_state flag is set to true and a required context state account is uninitialized, calling the verify_transfer_proof function leads to a return value of None. Should the close_split_context_state_on_execution flag also be true under these conditions, the process_transfer function attempts to reset lamport_destination, context_accounts.authority, and zk_token_proof_program addresses, intending to load source_account_authority into authority_info. However, due to the account order error, source_account_authority, lamport_destination, and context_accounts.authority are erroneously cleared, and authority_info

gets assigned to the zk_token_proof_program account.

This misalignment can cause failures in verifying the zero-knowledge proof essential for confirming the transfer's validity. Failure of the proof or the verification process may lead to transaction errors and the inability to complete the transfer.

Code Location:

Listing 6: src/extension/confidential_transfer/instruction.rs (Lines 1357,1368)

```
let close_split_context_state_on_execution =
    if let Some(close_split_context_state_on_execution_accounts) =
        context_accounts.close_split_context_state_accounts
    {
        // If `close_split_context_state_accounts` is set, then
 ↳ all context state accounts must
        // be `writable`.
        accounts.push(AccountMeta::new(*context_accounts.
 ↳ equality_proof, false));
        accounts.push(AccountMeta::new(
            *context_accounts.ciphertext_validity_proof,
            false,
        ));
        accounts.push(AccountMeta::new(*context_accounts.
 ↳ range_proof, false));
        accounts.push(AccountMeta::new_readonly(*
 ↳ source_account_authority, true));
        accounts.push(AccountMeta::new(
            *close_split_context_state_on_execution_accounts.
 ↳ lamport_destination,
            false,
        ));
        accounts.push(AccountMeta::new_readonly(*context_accounts.
 ↳ authority, true));
        accounts.push(AccountMeta::new_readonly(
            *close_split_context_state_on_execution_accounts.
 ↳ zk_token_proof_program,
            false,
        ));
        true
    } else {
```

```
355         // If `close_split_context_state_accounts` is not set,
 ↳ then context state accounts can
356         // be read-only.
357         accounts.push(AccountMeta::new_readonly(
358             *context_accounts.equality_proof,
359             false,
360         ));
361         accounts.push(AccountMeta::new_readonly(
362             *context_accounts.ciphertext_validity_proof,
363             false,
364         ));
365         accounts.push(AccountMeta::new_readonly(
366             *context_accounts.range_proof,
367             false,
368         ));
369         accounts.push(AccountMeta::new_readonly(*
 ↳ source_account_authority, true));
370
371         false
372     };
```

Listing 7: src/extension/confidential_transfer/verify_proof.rs (Line 187)

```
186 if close_split_context_state_on_execution {
187     let lamport_destination_account_info = next_account_info(
 ↳ account_info_iter)?;
188     let context_state_account_authority_info = next_account_info(
 ↳ account_info_iter)?;
189
190     msg!("Closing equality proof context state account");
191     invoke(
192         &zk_token_proof_instruction::close_context_state(
193             ContextStateInfo {
194                 context_state_account:
 ↳ equality_proof_context_state_account_info.key,
195                 context_state_authority:
 ↳ context_state_account_authority_info.key,
196             },
197             lamport_destination_account_info.key,
198         ),
199         &[
200             equality_proof_context_state_account_info.clone(),
```

```
201                 lamport_destination_account_info.clone(),
202                 context_state_account_authority_info.clone(),
203             ],
204         )?;
```

Listing 8: src/extension/confidential_transfer/verify_proof.rs

```
145 if no_op_on_split_proof_context_state
146             && check_system_program_account(
 ↳ equality_proof_context_state_account_info.owner).is_ok()
147         {
148             msg!("Equality proof context state account not initialized
 ↳ ");
149             return Ok(None);
150         }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:N/Y:N/R:N/S:U (5.0)

Recommendation:

Modify the conditions in the process_transfer, verify_transfer_with_fee_proof and verify_transfer_proof functions, and amend the ordering of accounts in the transfer_with_split_proofs function such that the new conditions accurately reflect the order of accounts as required by the transfer_with_split_proofs instruction handler.

Remediation Plan:

SOLVED: The Solana team solved this issue in commit 5da184a.

4.6 (HAL-06) CONFIDENTIAL TRANSFER AMOUNTS INFO LEAK VIA TRANSFER FEES - MEDIUM (5.0)

Description:

In the token-2022 program tokens can be set to incur a transfer fee, which is a percentage of the transaction amount and may have a maximum fee limit. This setup extends to confidential transfers, which utilize zero-knowledge proofs to confirm the legitimacy of encrypted balance changes. However, this mechanism inadvertently reveals information about every transfer's value to the individuals holding the transfer fee management keys.

Owners of the keys responsible for fee management can discern the value of confidential transfers by decrypting and comparing the fee balance before and after each transfer. When the fee is below the maximum cap, the precise transfer amount can be deduced. If the fee hits the cap, it implies that the transfer amount was at least the minimum necessary to incur the maximum fee.

BVSS:

AO:A/AC:L/AX:L/C:M/I:N/A:N/D:N/Y:N/R:N/S:U (5.0)

Recommendation:

Educate the program users on this limitation of the confidential transfer feature.

Remediation Plan:

SOLVED: The Solana team solved this issue in commit 1c3af5e.

4.7 (HAL-07) UNCONSTRAINED CONFIDENTIAL TRANSFER FEE WITHDRAW - LOW (2.5)

Description:

The procedures for processing confidential transfer instructions, specifically `WithdrawWithheldTokensFromAccounts` and `WithdrawWithheldTokensFromMint`, overlook certain limitations that should apply to confidential token accounts. These oversights include not adhering to an account's setting that may block credits to its pending balance, and failing to check or update the `pending_balance_credit_counter`, which should not exceed `maximum_pending_balance_credit_counter`. These functions directly transfer the full amount of withheld balance to the `pending_balance_lo` of the recipient account, potentially exceeding the decryptable limit of the balance.

An attacker with access to the fee management keys could manipulate the encrypted pending balance of an account, overriding the account owner's settings and potentially rendering the balance not decryptable for the victim.

BVSS:

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (2.5)

Recommendation:

Revert the transaction if `allow_balance_credits` is set on the destination account or if `pending_balance_credit_counter` is greater or equal `maximum_pending_balance_credit_counter`. Increment `pending_balance_credit_counter` after the transfer is executed.

Remediation Plan:

SOLVED: The Solana team solved this issue in commit 16384e2.

4.8 (HAL-08) MULTISG SIGNERS NOT REFERENCED CORRECTLY BY THE INSTRUCTION BUILDER - INFORMATIONAL (0.0)

Description:

This vulnerability affects the handling of multi-signature accounts during the creation of withdrawal instructions for withheld tokens in the confidential transfer fee feature, in both the `withdraw_withheld_tokens_from_mint` and `withdraw_withheld_tokens_from_accounts` handlers.

The issue arises from treating multi-signature accounts as non-signer accounts by the instruction builder `inner_withdraw_withheld_tokens_from_mint` and `inner_withdraw_withheld_tokens_from_accounts` functions. For withdrawals, the builder should reference multi-signature participants as signers, but it mistakenly categorizes them as non-signers. This misclassification can cause transactions to fail, especially when validating ownership in multi-signature accounts that necessitate multiple signers.

Code Location:

Listing 9: src/extension/confidential_transfer_fee/instruction.rs (Line 394)

```
388 accounts.push(AccountMeta::new_readonly(
389     *authority,
390     multisig_signers.is_empty(),
391 ));
392
393 for multisig_signer in multisig_signers.iter() {
394     accounts.push(AccountMeta::new(**multisig_signer, false));
395 }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

Reference the multisig signers as signers when assembling the withdraw fee instructions

Remediation Plan:

SOLVED: The Solana team solved this issue in commit d3de202.

4.9 AUTOMATED ANALYSIS

Description:

Halborn used automated security scanners to assist with the detection of well-known security issues and vulnerabilities. Among the tools used was cargo-audit, a security scanner for vulnerabilities reported to the RustSec Advisory Database. All vulnerabilities published in https://crates.io are stored in a repository named The RustSec Advisory Database. cargo audit is a human-readable version of the advisory database which performs a scanning on Cargo.lock. Security Detections are only in scope. All vulnerabilities shown here were already disclosed in the above report. However, to better assist the developers maintaining this code, the reviewers are including the output with the dependencies tree, and this is included in the cargo audit output to better know the dependencies affected by unmaintained and vulnerable crates.

Results:

ID	package	Short Description
RUSTSEC-2022-0093	ed25519-dalek	Double Public Key Signing Function Oracle Attack on ‘ed25519-dalek‘

THANK YOU FOR CHOOSING

// HALBORN