Elusiv

Audit

Presented by:

OtterSec contact@osec.io

Harrison Green hgarrereyn@osec.io

Ajay Kunapareddy d1r3wolf@osec.io

Contents

Appendices

01 | Executive Summary

Overview

Elusiv engaged OtterSec to perform an assessment of the `elusiv` program. This assessment was conducted between June 29th and September 7th, 2022.

We performed an initial assessment of the work-in-progress codebase between June 29th, 2022 and July 8th, 2022. Later, once final changes were merged in, we performed a follow up assessment between August 29th, 2022 and September 7th, 2022.

Critical vulnerabilities were communicated to the team prior to the delivery of the report to speed up remediation. After delivering our audit report, we worked closely with the team over to streamline patches and confirm remediation.

Key Findings

The following is a summary of the major findings in this audit.

- 15 findings total
- 3 vulnerabilities which could lead to loss of funds
  - OS-ELV-ADV-00: Nullifier Trees can be Cleared
  - OS-ELV-ADV-01: BaseCommitmentHash Account Closing is not Atomic
  - OS-ELV-ADV-02: Incorrect Queue Length Calculation

02 | Scope

The source code was delivered to us in a git repository at github.com/elusiv-privacy/elusiv.

There was a total of 1 program included in this audit. A brief description of the program is as follows.

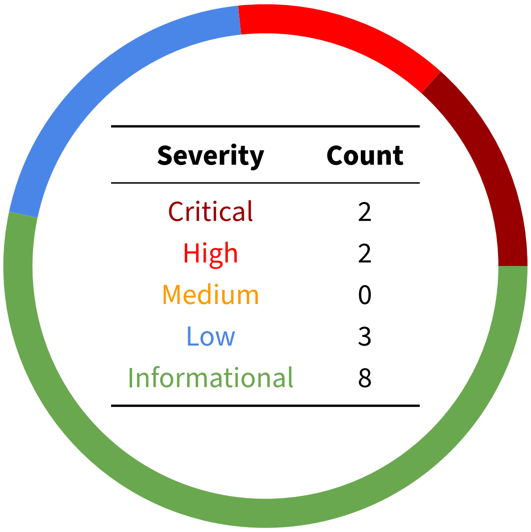
Name	Description
elusiv	Zero-knowledge payment system for SOL and spl-token

03 | Findings

Overall, we report 15 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 don't have an immediate impact but will help mitigate future vulnerabilities.

The below chart displays the findings by severity.



Severity	Count
Critical	2
High	2
Medium	0
Low	3
Informational	8

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 C.

ID	Severity	Status	Description
OS-ELV-ADV-00	Critical	Resolved	EnableNulliferSubAccount can be used to clear a nullifier tree, enabling infinite spending.
OS-ELV-ADV-01	High	Resolved	An attacker can steal pool money by farming duplicate work on commitment hashing.
OS-ELV-ADV-02	Critical	Resolved	Incorrect length calculation for ring queue allows attacker to drain the pool.
OS-ELV-ADV-03	High	Resolved	`root` of the nullifier account is getting updated with the storage root value after reset.
OS-ELV-ADV-04	Low	Resolved	In cases where instructions are not executed atomically during proof verification, an attacker can hijack and freeze the flow.
OS-ELV-ADV-05	Low	Resolved	A duplicate nullifier account PDA is used to prevent simultaneous verifications for the same proof but can be used to prevent computation.
OS-ELV-ADV-06	Low	Resolved	If a token account does not exist during finalization of a proof verification, instruction will not complete.

OS-ELV-ADV-00 [crit] [resolved] | Nullifier Trees can be Cleared

Description

In order to store large amounts of related data in Solana accounts, Elusiv uses the concept of a *subaccount*. In the Elusiv framework, a subaccount is a data-only account pointed to by a parent tracker account.

Nullifier trees are one example of a structure that must be stored via subaccounts. The purpose of a nullifier tree in the context of a zero knowledge token program is to prevent duplicate spending. Every spent transaction gets marked in the nullifier tree which means if a user tries to spend a transaction which has already been marked in the nullifier tree, it will be rejected as an attempted double-spend.

In order to construct the initial data accounts, a user invokes `EnableNullifierSubAccount` providing both the `nullifier_account` (parent tracker account) and the data-only `sub_account`:

src/processor/accounts.rs RUST

```rust
pub fn enable_nullifier_sub_account(
    nullifier_account: &AccountInfo,
    sub_account: &AccountInfo,

    _merkle_tree_index: u64,
    sub_account_index: u32,
) -> ProgramResult {
    // Note: we don't zero-check these accounts, BUT we need to
    ↪   manipulate the maps we store in each account and set the size to
    ↪   zero
    setup_sub_account::<NullifierAccount, {NullifierAccount::COUNT}>(
        nullifier_account,
        sub_account,
        sub_account_index as usize,
        false
    )?;

    // Set map size to zero (leading u32)
    let data = &mut sub_account.data.borrow_mut()[..];
    for b in data.iter_mut().take(4) {
        *b = 0;
    }

    Ok(())
}
```

However, this code does not verify that the provided `sub_account` is not already in use. Since it resets the map size to zero (by clearing the first four bytes of the data), it implicitly deletes all of the stored data in the account.

An attacker could use this vulnerability to clear the nullifier tree and spend the same money repeatedly until the program pool is drained.

Patch

This vulnerability has been fixed in commit `e436e69` by storing a one-byte marker at the start of all subaccounts that indicates whether the account is in use. Any code that initializes a subaccount ensures that this flag is not set before initializing the account.

OS-ELV-ADV-01 [high] [resolved] | BaseCommitmentHash Account Closing is not Atomic

Description

When a user deposits new funds into the system, the protocol must first compute the BaseCommitmentHash which represents a base commitment, the amount deposited, and the token type. This stage can be performed in parallel – each user using a separate BaseCommitmentHashingAccount to track the state of the computation.

Next, this base commitment is added to the global merkle tree which requires computing all of the pairwise hashes from the updated leaf to the root. This step must happen synchronously since the protocol uses a single, global merkle tree.

The instruction used to enqueue the base commitment hash is FinalizeBaseCommitmentHash:

```rust
pub fn finalize_base_commitment_hash<'a>(
    original_fee_payer: &AccountInfo<'a>,
    commitment_hash_queue: &mut CommitmentQueueAccount,
    hashing_account_info: &AccountInfo<'a>,

    _hash_account_index: u64,
) -> ProgramResult {
    let data = &mut hashing_account_info.data.borrow_mut()[..];
    let hashing_account = BaseCommitmentHashingAccount::new(data)?;
    guard!(hashing_account.get_is_active(), ComputationIsNotYetFinished);
    guard!(hashing_account.get_fee_payer() ==
    ↪   original_fee_payer.key.to_bytes(), InvalidAccount);
    guard!(
        (hashing_account.get_instruction() as usize) ==
    ↪   BaseCommitmentHashComputation::INSTRUCTIONS.len(),
        ComputationIsNotYetFinished
    );

    let commitment = hashing_account.get_state().result();
    let mut commitment_queue =
    ↪   CommitmentQueue::new(commitment_hash_queue);
    commitment_queue.enqueue(
        CommitmentHashRequest {
            commitment: fr_to_u256_le(&commitment),
            fee_version:
    ↪   u64_as_u32_safe(hashing_account.get_fee_version()),
            min_batching_rate: hashing_account.get_min_batching_rate(),
        }
    )?;

    // Close hashing account
    close_account(original_fee_payer, hashing_account_info)
}
```

This instruction takes the temporary `hashing_account` (containing the base commitment hash) and enqueues it in the singleton `commitment_hash_queue`. Once the request has been enqueued, it closes the temporary hashing account by transfering all the rent lamports back to the original fee payer.

However, since rent is not enforced immediately after each instruction in Solana, this hashing account may stay alive across multiple instructions if the instructions are included in a single transaction. With this mechanism, an attacker can invoke `FinalizeBaseCommitmentHash` multiple times in a row on the same hashing account in order to enqueue many duplicate requests. Following this, the attacker can invoke the subsequent crank instructions to farm rewards from the pool.

Patch

This vulnerability has been fixed in `33a542` by explicitly deactivating the hashing account before closing it. While the account may stay alive for several more instructions, subsequent calls to `FinalizeBaseCommitmentHash` will fail.

OS-ELV-ADV-02 [crit] [resolved] | Incorrect Queue Length Calculation

Description

In `src/state/queue.rs`, `RingQueue` trait was defined with incorrect logic in length function for `tail < head` case.

src/state/queue.rs RUST

```rust
fn len(&self) -> u64 {
    let head = self.get_head();
    let tail = self.get_tail();

    if tail < head {
        head + tail
    } else {
        tail - head
    }
}
```

This will return length more, less than actual based on the cases. For example, Lets take a Queue of len 10

1. Head = 1, Tail = 0. Actual length = 10, Length by len function = 1 + 0 = 1
2. Head = 9, Tail = 8. Actual length = 10, Length by len function = 9 + 8 = 17

The `next_batch`, `remove` functions of queue are guarded by the `len` function. By abusing the incorrect calculation, an attacker as an retainer can create commitment batches for an empty queue and get the extra hashing rewards. Since no one are paying fee is for those extra hashing rewards, it leads to draining of pool.

Proof of Concept

The following scenario explains how retainer can abuse this

1. Considering the commit queue with size 240, Batch length = 16.
2. Head was at queue end - 239, Tail was at 0. Actual len = 1, len() gives 0 + 239 = 239
3. Length = 239 . Commitment hashing was initiated with 16 elements, Head ptr moves to 15.
4. Length = 15. Commitment hashing was initiated with 15 elements, Head ptr moves to 30.
5. Length = 30. Commitment hashing was initiated with 16 elements, Head ptr moves to 46.
6. This will repeats in circular manner repeatedly until `head <= tail`

It is possible to do batching 225 times, for each batch number of hash computation rounds are 31. Based on the above condition (head = 238, Tail = 0), Attacker can drain `225 * 31 * hast_tx_fee`

Remediation

Correct the length calculation for Ring Queue in `tail <= head` case like `size - (head - tail)`

Patch

The calculation in `len` function of queue was corrected. Fixed in 7bd57b1.

OS-ELV-ADV-03 [high] [resolved] | Using Storage Root Value After Reset

Description

The reset_active_merkle_tree instruction is used to close the active Merkle tree by storing its root value in a nullifier account and activating a new one. But the storage root value is taken after resetting the storage, which returns a default value.

src/processor/accounts.rs RUST

```
storage_account.reset();
storage_account.set_trees_count(&(active_merkle_tree_index.checked_add(1).ok_or(MATH_ERR
active_nullifier_account.set_root(&storage_account.get_root());
```

The reset function updates the next_commitment_ptr to 0, which was used in the get_node function invoked by get_root function.

src/processor/accounts.rs RUST

```
let ptr = self.get_next_commitment_ptr() as usize;

// Accessing a node, that is non-existent (yet) -> we use the default
    ↪ value
if use_default_value(index, level, ptr) {
    EMPTY_TREE[MT_HEIGHT as usize - level]
} else {
```

Since ptr value is zero, use_default_value returns true. get_root function returns the default value instead of actual root value.

Proof of Concept

Here is a test case for proving get_root fails to give actual value after reset.

src/processor/accounts.rs RUST

```
#[test]
fn test_reset_root() {
    let mut data = vec![0; StorageAccount::SIZE];
    generate_storage_accounts_valid_size!(accounts);
    let mut storage_account = StorageAccount::new(&mut data,
    ↪ accounts).unwrap();
    let temp = Fr::new(BigInteger256::new([11638450640257199038,
     ↪  7595865483997155469, 13070293587080331272,
     ↪  1693899333659075201]));
    let test_root = fr_to_u256_le(&temp);

    assert_ne!(storage_account.get_root(), test_root);

    storage_account.set_node(&test_root, 0, 0);
    storage_account.set_next_commitment_ptr(&1);

    assert_eq!(storage_account.get_root(), test_root); // Temp root

    storage_account.reset();
    assert_eq!(storage_account.get_root(), test_root); // Fails
}
```

Remediation

Use the root value of storage before `reset`.

Patch

Fixed in d5c6f1d by fetching the storage root before resetting.

OS-ELV-ADV-04 [low] [resolved] | Proof Verification DoS via Hijacking

Description

In order to perform proof computation, the instructions `InitVerification`, `InitVerificationTransferFee`, and `InitVerificationProof` need to be executed in order. *Note: some initial steps of proof computation can run asynchronously with these instructions*.

These instructions are intended to be run by untrusted third parties (a.k.a. "wardens") who invoke Solana instructions on behalf of users in order to earn a small fee.

Most of the Elusiv program flow is designed as a series of "cranks" – instructions that can be invoked by *anyone* to progress the state of the system forward.

However, in the proof verification flow, there is the potential for the flow to get stuck. The fee payer that invokes `InitVerificationTransferFee` must also sign for the subsequent `InitVerificationProof` instruction.

If a well-behaved warden executes these instructions in separate transactions, a malicious user can invoke `InitVerificationTransferFee` before the warden in order to hijack the flow and prevent the proof from being computed.

Remediation

Refactor the `InitVerificationProof` instruction so it does not need to be signed by the original fee payer. Or require the first three instructions in proof verification to be signed by the warden such that an attacker cannot hijack the instruction flow in the middle.

Patch

Fixed in 6bd77e5 by requiring `InitVerification` and `InitVerificationTransferFee` to be executed by the same warden.

OS-ELV-ADV-05 [low] [resolved] | Proof Verification DoS via Duplicate Nullifier Account

Description

To protect multiple wardens from executing the same proof verification simultaneously (resulting in all but one of the wardens missing out on a fee payout), the verification flow uses a PDA derived from the nullifier hashes to act as a sentinel for the proof.

If the PDA exists during `InitVerification`, the instruction will fail, thereby preventing `InitVerification` from running more than once for the same proof.

However, a malicious user can listen for proof requests (via the same mechanism wardens use) and invoke `InitVerification` to initialize the PDA, thereby preventing any other wardens from verifying this proof.

Patch

Fixed in 0be0198 by providing wardens the option to skip the nullifier account check when initiating a proof verification.

OS-ELV-ADV-06 [low] [resolved] | Proof Verification DoS via Missing Token Account

Description

Recipients of a token transfer out of the zero-knowledge system can be either native Solana accounts (for SOL) or spl-token Token Accounts (for USDT or USDC).

In order to finalize a valid proof verification a warden calls `FinalizeVerificationTransfer` which both transfers money to the recipient account and pays the warden for computation fees.

If the recipient account does not exist at the time of this instruction, spl-token transfers will fail which means the warden is unable to complete the instruction and get paid. In general, even if a warden checks the existence of the account before starting verification, it may be deleted before finalization resulting in the same problem.

Remediation

A recommended fix is to send spl-token payments to an associated token account for a given recipient rather than directly specifying a token account. The benefit of this approach is that if the associated token account does not already exist, the `FinalizeVerificationTransfer` instruction can construct it, which means finalization will not be blocked by this constraint.

Patch

Fixed in 5a21fe3. Funds can now be sent to associated token accounts and funds that are sent to an address which becomes invalid at the time of verfication are consumed as fees.

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 do represent antipatterns and could introduce a vulnerability in the future.

ID	Status	Description
OS-ELV-SUG-00	Resolved	Ensure mt_array_index is invoked with legal values
OS-ELV-SUG-01	Resolved	The prg attribute is unused in the ElusivInstruction proc-macro.
OS-ELV-SUG-02	TODO	Enable stricter proc-macro usage requirements.
OS-ELV-SUG-03	TODO	Consider the Pyth oracle confidence interval when performing conversions.
OS-ELV-SUG-04	Resolved	Code for checking StorageAccount was repeated, can be eliminated by using `is_full` function.
OS-ELV-SUG-05	Resolved	Unused MT_COMMITMENT_START constant.
OS-ELV-SUG-06	Resolved	Irrelevant Error message for guards about computation.
OS-ELV-SUG-07	Resolved	Unnecessary invoke_signed without signer seeds.

OS-ELV-SUG-00 [resolved] | Assert Bounds of mt_array_index

Description

The `mt_array_index` utility function computes the offset into a MT array based on the `index` and `level`:

src/state/storage.rs RUST

```rust
pub fn mt_array_index(index: usize, level: usize) -> usize {
    two_pow!(usize_as_u32_safe(level)) - 1 + index
}
```

However, this function is only correct when $\texttt{index} < 2^{\texttt{level}}$.

Remediation

Consider adding an explicit assertion so that this function will abort rather than returning incorrect values when provided with bad arguments.

Patch

Fixed in 10c15d9.

OS-ELV-SUG-01 [resolved] | Remove prg Attribute

Description

The ElusivInstruction proc-macro defines a `prg` attribute:

elusiv-derive/src/elusiv$_{i}nstruction.rs$ RUST

```
// Program owned accounts that satisfy a pubkey constraint
"prg" => {
    user_accounts.extend(quote!{ #account: #user_account_type, });
    account_init.push(quote!{
        accounts.push(AccountMeta::#account_init_fn(#account.0,
     ↪  #is_signer));
    });
```

However, this attribute is not used by any of the existing elusiv instructions.

Remediation

Consider removing the attribute.

Patch

Fixed in 10c15d9.

OS-ELV-SUG-02 | Stricter Proc-Macro Usage Requirements

Description

Procedural macros are used throughout the codebase to remove boilerplate code and centralize logic. While this technique is generally useful, it also can make some bugs harder to spot if it occurs during the resolution of a macro.

Specifically, proc-macros that implement custom parsing on a token stream or list of sub-attributes (e.g. `signer`, `writable`, owned, etc... in the ElusivInstruction macro) should be very strict about malformed attributes.

For example, in the current implementation, mistyping `signer` as `singer` inside an ElusivInstruction macro will result in the proc-macro derivation silently ignoring this attribute and *not adding* an `is_signer` check. Instead, it would be much more secure to identify that `singer` is a malformed attribute and immediately panic.

While we did not identify any cases of mistyping in the current codebase, these types of bugs can sneak in during a refactor and be hard to spot since compilation will still succeed and, in general, tests will still pass.

Remediation

Explicitly verify that all sub attributes (and other custom properties of a proc-macro) are of the expected type or format and panic if not.

OS-ELV-SUG-03 | Use Pyth Oracle Confidence Interval

Description

The Elusiv program uses Pyth oracles to convert between USDC, USDT, and native SOL in order to compute fee amounts for spl-token transfers. Currently, the code does not utilize the confidence metric in the Pyth oracle.

In times of strange market conditions or potentially malicious activity, these rates may have a high variance and therefore large confidence interval.

Since token conversion only occurs to compute fixed computation fees (and not the actual value being transferred), the implication of using a bad exchange rate is minor.

Remediation

It is recommended to utilize the confidence interval when computing the exchange rate. For example, by using the side of the rate that is beneficial for Elusiv such that computation fees are always covered.

OS-ELV-SUG-04 [resolved] | Redundant StorageAccount's is_full condition

Description

In `is_mt_full` function, the code for checking `next_commitment_ptr` is redundant with the `is_full` function of `StorageAccount`.

src/processor/accounts.rs RUST

```rust
fn is_mt_full(
    storage_account: &StorageAccount,
    queue: &CommitmentQueue,
) -> Result<bool, ProgramError> {
    let commitments = storage_account.get_next_commitment_ptr() as usize;
    if commitments >= MT_COMMITMENT_COUNT {
        return Ok(true)
    }

    if commitments + queue.next_batch()?.0.len() >= MT_COMMITMENT_COUNT {
        return Ok(true)
    }

    Ok(false)
}
```

The above checks can be replaced by `is_full` function of `StorageAccount` to reduce redundancy in logic, which reduces the probability of making an error during code refactoring.

Remediation

Replace the `next_commitment_ptr` check with `is_full` function.

src/processor/accounts.rs RUST

```rust
fn is_mt_full(
    storage_account: &StorageAccount,
    queue: &CommitmentQueue,
) -> Result<bool, ProgramError> {
    if storage_account.is_full() {
        return Ok(true)
    }
```

Patch

Fixed in 10c15d9.

OS-ELV-SUG-05 [resolved] | Unused MT_COMMITMENT_START constant

Description

In `src/storage/state.rs`, there is a constant which was not used anywhere.

src/state/storage.rs RUST

```rust
/// Index of the first commitment in the MT
pub const MT_COMMITMENT_START: usize = two_pow!(MT_HEIGHT) - 1;
```

Remediation

Since, the constant is not used anywhere remove it.

Patch

Fixed in 10c15d9.

OS-ELV-SUG-06 [resolved] | Irrelevant Error message

Description

For a Hashing account, `is_active` is used to track the computation has been initiated/started or not. In compute and finalize hashing functions for `is_active` the guard assertion gives an incorrect error message.

src/processor/commitment.rs RUST

```rust
pub fn compute_base_commitment_hash(
    ...
) -> ProgramResult {
    guard!(hashing_account.get_is_active(), ComputationIsNotYetFinished);
```

src/processor/commitment.rs RUST

```rust
pub fn finalize_base_commitment_hash<'a>(
    ...
) -> ProgramResult {
    pda_account!(mut hashing_account, BaseCommitmentHashingAccount,
     ↪  hashing_account_info);
    guard!(hashing_account.get_fee_version() == fee_version,
     ↪  InvalidFeeVersion);
    guard!(hashing_account.get_is_active(), ComputationIsNotYetFinished);
```

src/processor/commitment.rs RUST

```rust
pub fn compute_commitment_hash<'a>(
    ...
) -> ProgramResult {
    guard!(hashing_account.get_is_active(), ComputationIsNotYetFinished);
```

src/processor/commitment.rs RUST

```rust
pub fn finalize_commitment_hash(
    ...
) -> ProgramResult {
    guard!(hashing_account.get_is_active(), ComputationIsNotYetFinished);
```

Incorrect error messages might lead in confusion while debugging the logs.

Remediation

It is recommended to create a new error `ComputationIsNotYetStarted` and use it in the above four places.

Patch

Fixed in 10c15d9.

OS-ELV-SUG-07 [resolved] | Unnecessary invoke_signed with no signer seeds

Description

In `src/processor/utils.rs`, `transfer_with_system_program` function is using `invoke_signed` without the signer seeds.

src/processor/utils.rs RUST

```rust
solana_program::program::invoke_signed(
    &instruction,
    &[
        source.clone(),
        destination.clone(),
        system_program.clone(),
    ],
    &[],
)
```

Since the program is invoked without signer seeds, there is no need to use `invoke_signed`.

Remediation

Replace the `invoke_signed` with `invoke` function.

Patch

Fixed in 10c15d9.

A | 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 onchain program. In other words, there is no way to steal tokens or deny service, ignoring any Solana specific quirks such as account ownership issues. An example of a design vulnerability would be an onchain oracle which could be manipulated by flash loans or large deposits.

On the other hand, auditing the implementation of the program requires a deep understanding of Solana's execution model. Some common implementation vulnerabilities include account ownership issues, arithmetic overflows, and rounding bugs. For a non-exhaustive list of security issues we check for, see Appendix B.

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 get a comprehensive understanding of the program first. In our audits, we always approach any target in a team of two. 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.

B | Implementation Security Checklist

Unsafe arithmetic

Integer underflows or overflows	Unconstrained input sizes could lead to integer over or underflows, causing potentially unexpected behavior. Ensure that for unchecked arithmetic, all integers are properly bounded.
Rounding	Rounding should always be done against the user to avoid potentially exploitable off-by-one vulnerabilities.
Conversions	Rust as conversions can cause truncation if the source value does not fit into the destination type. While this is not undefined behavior, such truncation could still lead to unexpected behavior by the program.

Account security

Account Ownership	Account ownership should be properly checked to avoid type confusion attacks. For Anchor, the safety of unchecked accounts should be clearly justified and immediately obvious.
Accounts	For non-Anchor programs, the type of the account should be explicitly validated to avoid type confusion attacks.
Signer Checks	Privileged operations should ensure that the operation is signed by the correct accounts.
PDA Seeds	PDA seeds are uniquely chosen to differentiate between different object classes, avoiding collision.

Input validation

Timestamps	Timestamp inputs should be properly validated against the current clock time. Timestamps which are meant to be in the future should be explicitly validated so.
Numbers	Sane limits should be put on numerical input data to mitigate the risk of unexpected over and underflows. Input data should be constrained to the smallest size type possible, and upcasted for unchecked arithmetic.
Strings	Strings should have sane size restrictions to prevent denial of service conditions
Internal State	If there is internal state, ensure that there is explicit validation on the input account's state before engaging in any state transitions. For example, only open accounts should be eligible for closing.

Miscellaneous

Libraries	Out of date libraries should not include any publicly disclosed vulnerabilities
Clippy	cargo clippy is an effective linter to detect potential anti-patterns.

C Vulnerability Rating Scale

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

Critical Vulnerabilities which immediately lead to loss of user funds with minimal preconditions

Examples:

- Misconfigured authority/token account validation
- Rounding errors on token transfers

High Vulnerabilities which could lead to 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 which could lead to denial of service scenarios or degraded usability.

Examples:

- Malicious input cause computation limit exhaustion
- Forced exceptions preventing normal use

Low Low probability vulnerabilities which could still be 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
- Uncaught Rust errors (vector out of bounds indexing)