# Code

Security Assessment

February 10th, 2025 — Prepared by OtterSec

| | |
|---|---|
| Tamta Topuria | tamta@osec.io |
| Nicola Vella | nick0ve@osec.io |
| Robert Chen | r@osec.io |

# Table of Contents

# 01 — Executive Summary

## Overview

Code payments engaged OtterSec to assess the `code-vm` program. This assessment was conducted between December 23rd, 2024 and January 31st, 2025. For more information on our auditing methodology, refer to Appendix B.

## Key Findings

We produced 9 findings during this audit engagement, all of which were successfully remediated according to our recommendations.

Furthermore, the argument parsing in several functions creates misaligned references, resulting in undefined behavior (OS-CDE-ADV-03), and the overflow checks are missing in release mode, potentially allowing an overflow if parameters such as account size and number of accounts are assigned large values during the initialization of a memory account (OS-CDE-ADV-04).

We also recommended optimizing the transfer and withdraw functionalities by consolidating them into a single function to reduce redundancy (OS-CDE-SUG-00) and advised avoiding the implicit handling of unexpected states, such as empty allocations or missing entries (OS-CDE-SUG-01). Additionally, we made suggestions for modifying the codebase to improve functionality, efficiency, and mitigate potential security issues (OS-CDE-SUG-02).

# 02 — Scope

The source code was delivered to us in a Git repository at https://github.com/code-payments/code-vm. This audit was performed against commit c49f95d.

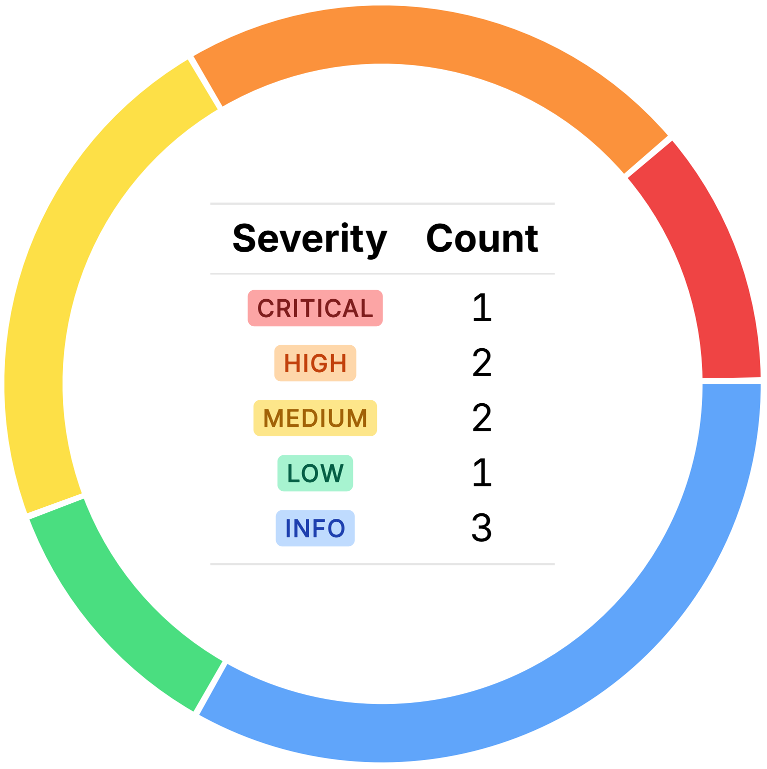
**A brief description of the program is as follows:**

| Name | Description |
|---|---|
| code-vm | A payment-optimized execution layer that reduces transaction fees and account rent by utilizing compact account representations and off-chain compression for dormant accounts. |

# 03 — Findings

Overall, we reported 9 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 | 1 |
| HIGH | 2 |
| MEDIUM | 2 |
| LOW | 1 |
| INFO | 3 |

# 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-CDE-ADV-00 | CRITICAL | RESOLVED ✓ | If source and destination accounts in `transfer` or `withdraw` operations are the same, inaccurate balance updates occur due to in-memory caching overwriting intermediate values. |
| OS-CDE-ADV-01 | HIGH | RESOLVED ✓ | `process_init_timelock` does not validate the input bump seeds are canonical, enabling malicious users to generate multiple valid PDAs for the same account. |
| OS-CDE-ADV-02 | HIGH | RESOLVED ✓ | `CircularBuffer::contains` may incorrectly return true for empty buffer slots, as it does not account for empty items. |
| OS-CDE-ADV-03 | MEDIUM | RESOLVED ✓ | The argument parsing in several functions creates misaligned references, resulting in undefined behavior. |
| OS-CDE-ADV-04 | MEDIUM | RESOLVED ✓ | Overflow checks are missing in release mode, potentially allowing an overflow if parameters such as `account_size` and `num_accounts` are assigned large values during the initialization of a `MemoryAccount`. |
| OS-CDE-ADV-05 | LOW | RESOLVED ✓ | Merkle tree functions lack verification to ensure that the proof vector matches the tree depth, enabling the possibility of proving the presence of intermediate nodes instead of leaves. |

## Inaccurate In-Memory Balance Updates CRITICAL OS-CDE-ADV-00

### Description

`transfer` and `withdraw` in `code-vm` are vulnerable to incorrect accounting when the source and destination accounts are the same. In scenarios where the source account and the destination account are identical, the program logic may process both operations in memory without distinguishing between the two. This creates a situation where only the final ”greater” destination value (after adding funds) is written back to storage, without accurately subtracting the funds first.

### Remediation

Introduce an explicit check to prevent operations where the source and destination are the same.

### Patch

Resolved in PR#11.

## Missing Canonical Bump Validation HIGH OS-CDE-ADV-01

### Description

`process_init_timelock` does not check that the input bumps are canonical. The bump values are taken directly from the caller without validating whether they match the canonical bumps derived programmatically. A canonical bump corresponds to the first valid seed value that may be used to generate a Program Derived Address (PDA). By not checking for the canonical bump, discrepancies may arise between the addresses expected by the program and those derived in real time, as the address would resolve to a different PDA than what the canonical calculation would yield.

### Remediation

Validate that the provided bumps match the canonical bumps.

### Patch

Resolved in PR#13.

## Improper Handling of Empty Items In Circular Buffer HIGH OS-CDE-ADV-02

### Description

`CircularBuffer::contains` checks whether a specific item exists within the buffer. However, it does not ignore empty items when checking for a given item. In the context of this CircularBuffer, empty items are represented as arrays filled with default values ( `[0; M]` ). These empty slots may skew the behavior of `contains` if they are not properly handled. Without ignoring empty items, the search may incorrectly match these slots and return a false positive. This is especially problematic when an item (such as `[0; M]` ) is used to represent an empty state in the buffer but is mistakenly treated as a valid entry.

>_ *api/src/types/circular_buffer.rs* RUST

```rust
pub fn find_index(&self, item: &[u8]) -> Option<usize> {
    self.items.iter().position(|x| x.eq(item))
}

pub fn contains(&self, item: &[u8]) -> bool {
    self.find_index(item).is_some()
}
```

### Remediation

Modify the the function to exclude empty items when performing the search, ensuring that only non-empty, valid items are considered in the check.

### Patch

Resolved in PR#14.

## Misaligned Memory Access MEDIUM OS-CDE-ADV-03

### Description

In the current implementation, function arguments are referenced with their type values without ensuring proper memory alignment, and the structures utilize `#[repr(C, packed)]`, preventing the compiler from inserting padding for alignment. This creates misaligned references during argument parsing in most functions, resulting in undefined behavior.

>_ *api/src/instruction.rs* RUST

```rust
#[repr(C, packed)]
#[derive(Clone, Copy, Debug, Pod, Zeroable)]
pub struct DepositIx {
    pub account_index: u16,
    pub amount: u64,
    pub bump: u8,
}
```

### Remediation

Store the types as a byte array ( `u64` may be stored as `[u8; 8]` as demonstrated in ore) and manually convert (as done here) when needed, ensuring proper alignment.

### Patch

Resolved in PR#6.

## Risk of Integer Overflow MEDIUM OS-CDE-ADV-04

### Description

Overflow checks are currently not enabled in release mode. Consequently, there is a possibility for integer overflow in the context of the `SliceAllocator`, specifically if a `MemoryAccount` is initialized with `account_size=u16::MAX` and `num_accounts=u32::MAX`. When calculating the total memory required for these accounts by multiplying `account_size` by `num_accounts`, the subsequent product may be extremely high resulting in an overflow.

### Remediation

Enable overflow checks and properly sanitize parameters such as `account_size` and `num_accounts`.

### Patch

Fixed in PR#18.

## Failure to Verify Proof Vector Length LOW OS-CDE-ADV-05

### Description

When performing a Merkle proof to verify the existence of specific data within the tree, a proof vector (collection of hashes) is provided to the verification function. This proof must correspond to the correct level of the tree. To ensure validity, the verification function should confirm that the proof vector length matches the tree depth. Without this check, invalid proofs may verify intermediate nodes instead of leaves. However, in the current implementation, this issue is mitigated because the `compress` / `decompress` methods construct the leaves in a manner that prevents intermediate hashes in the proof from matching the compressed data.

### Remediation

Verify the proof length against the tree depth to ensure correctness.

### Patch

Resolved in PR#7.

# 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-CDE-SUG-00 | `transfer` and `withdraw` instructions may be unified into one function as their implementations overlap with redundant code handling. |
| OS-CDE-SUG-01 | `try_write` and `try_compress` implicitly handle unexpected states, such as empty allocations or missing entries, which may result in unintentional behavior. |
| OS-CDE-SUG-02 | Recommendation for modifying the codebase to improve functionality, efficiency, and mitigate potential security issues. |

## Code Optimization OS-CDE-SUG-00

### Description

Both `transfer` and `withdraw` perform nearly identical operations, involving transferring balances between virtual accounts. The main difference lies in the deletion of the source account in `withdraw`. Thus, both these operations may be consolidated by passing the entire balance (`vta.amount`) and adding an optional `delete_if_empty` flag which will enable standard transfers without deletion (when the flag is set to false), and withdraw-like operations where the source account is emptied and subsequently deleted in cases where the flag is set.

### Remediation

Consolidate the `transfer` and `withdraw` instructions to reduce redundancy and improve maintainability by avoiding the need to maintain two parallel implementations.

## Implicit Handling of Unexpected States OS-CDE-SUG-01

### Description

`try_write` automatically allocates space if the memory slot at `account_index` is empty, without verifying whether this behavior is intentional, potentially overwriting an unintended slot. Additionally, `try_compress` lacks proper error handling. If the item (hash) does not exist in the storage state, `try_compress` fails to signal an error and simply ignores the missing entry.

>_ *api/src/helpers.rs* RUST

```rust
pub fn try_write<'a>(
    vm_memory: &AccountInfo<'_>,
    account_index: u16,
    account: &VirtualAccount,
) -> ProgramResult {
    let (n, m) = MemoryAccount::get_capacity_and_size(vm_memory);
    let mut data = MemoryAccount::get_data_mut(vm_memory)?;
    let mut mem = SliceAllocatorMut::try_from_slice_mut(&mut *data, n, m)?;
    let data = &account.pack();

    if mem.is_empty(account_index) {
        mem.try_alloc_item(account_index, data.len())?;
    }
    mem.try_write_item(account_index, data)?;
    Ok(())
}
```

### Remediation

Utilize a parameter in `try_write` that specifies whether allocation should occur if the slot is empty, as this should be known at compile time for almost all cases. Also, in `try_compress`, before attempting an insertion, ensure proper error handling in case the item does not exist.

## Code Refactoring OS-CDE-SUG-02

### Description

1. In `slice_allocator::try_free`, zero the account with `[0; account_size]` to avoid retaining stale data inside the freed slot.

   >_ *api/src/types/slice_allocator.rs* RUST

   ```rust
   pub fn try_free_item(&mut self, item_index: u16) -> ProgramResult {
       if item_index as usize >= self.capacity() || self.is_empty(item_index) {
           return Err(ProgramError::InvalidArgument);
       }

       self.state[item_index as usize] = ItemState::Free as u8;
       self.write(item_index as usize, &[0; 0])
   }
   ```

2. `check_omnibus` is currently sparsely utilized, despite multiple instructions relying on the `omnibus_info` account. Although this does not result in a security issue, inconsistent utilization of `check_omnibus` when interacting with the `omnibus_info` account risks disrupting expected accounting by potentially withdrawing the wrong token type.
3. Explicitly check `deposit_ata_info` (the `AccountInfo` representing a token account) utilizing `as_associated_token_account` within `process_deposit` and `process_withdraw_from_deposit`. This ensures that the provided `AccountInfo` corresponds to the expected Associated Token Account (ATA) for a given owner and mint.

   >_ *steel/lib/src/loaders.rs* RUST

   ```rust
   fn as_associated_token_account(
       &self,
       owner: &Pubkey,
       mint: &Pubkey,
   ) -> Result<spl_token::state::Account, ProgramError> {
       self.has_address(&spl_associated_token_account::get_associated_token_address(
           owner, mint,
           ))?
       .as_token_account()
   }
   ```

4. Modify `helpers::check_uninitialized_pda` to ensure the account is also owned by the `system_program`.

### Remediation

Incorporate the above-stated refactors.

# 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.