Commerce Payments Audit 2

Coinbase Protocol Security

June 11, 2025

Contents

Audit Scope

Initial Review Commit: 41acc691607abac99562388d5c9a1017fe8d76bd

Repository: https://github.com/base/commerce-v1/tree/main

Files:

PaymentEscrow.sol
ERC3009Collector.sol
Permit2TokenCollector.sol
PreApprovalTokenCollector.sol
SpendPermissionTokenCollector.sol
TokenCollector.sol

Latest Review Commit: 8a2c6e97f32ad35961f05fc34699b1c63e286c31

Executive Summary

This report presents the outcomes of our collaborative engagement with the Base team, focusing on the comprehensive evaluation of the Commerce V1 Smart Contracts. The commerce-v1 repository was reviewed from March 26 to March 31.

System Overview

The Commerce Escrow protocol consists of the Escrow smart contract itself, and a few starter Token Collector contracts which hold their own custom logic for handling how tokens are to be moved from the payer to the Escrow smart contract.

The current possible Token Collector contracts are for interaction with:

- ERC3009-standard tokens
- Prepproval of the token through user providing a token allowance independently
- Interaction with the canonical Permit2 smart contract to provide a signed allowance
- Interaction with the Spend Permission Manager smart contract to provide access to retail funds

The Escrow Smart contract is designed so that any independent vendor can interact with the contract without special permissions. The optional flows are to authorize the collection of funds from the user and immediately move the funds to the intended receiver through function `charge()`, or to perform in two separate actions through first calling `authorize()` to move funds from the payer

to the Escrow contract and then calling `capture()` to move the funds from the Escrow contract to the intended receiver.

If the latter flow is chosen, the operator has the option to call `capture()` several times on the same payment details, using smaller amounts that sum up to less or equal to the total initial amount authorized.

The operator can void any remaining funds left in the Escrow contract at any given time, sending those funds to the payer. A user also has the option to reclaim funds left in the Escrow contract after the authorization deadline has passed. Lastly, funds that were already captured can be refunded to the user within the refund deadline. These funds must have been logged as refundable for the payment details within the Escrow contract; a separate TokenCollector contract is expected to handle the refund logic.

It is important to note that the Escrow contract holds large amounts of user funds. The operator of a payment has freedom to choose the target for TokenCollector and data to use with the call, from the Escrow Smart contract. The smart contract is protected from the threat this presents by ensuring that the external call chosen by the operator address results in the expected amount of tokens being held by the contract by the end of the call.

Properties of the Protocol

- It is not possible to update a `paymentDetailsHash` key in the mapping `_paymentState` that has the `operator` set as `address(0)`. This is because all initial updates to the mapping occur through functions `charge()` and `authorize()` which can only be called by the specified `operator` in `paymentDetails`.
- It is not possible to update a `paymentDetailsHash` key in the mapping `paymentState` that has the `token` set as `address(0)` due to the call to `_collectTokens()` in `charge()` and `authorize()`
- The `amount` cast to a `uint120` and used as either `state.capturable` or `state.refundable` is always less than `type.max(uint120)`, ensuring unsafe integer casting does not occur.
  - This is guaranteed by both the use of the `validAmount` modifier on input `amount` **and** by ensuring `amount` is always less or equal to `state.capturable` in function `capture()`.
- Let x be a `paymentDetailsHash`, and assume `capture()` is called n times on the `paymentDetails` corresponding to x. The sum of all n amounts used in `capture()` is less

or equal to the `initial state.capturable` amount corresponding to x, since each ith time `capture()` is called, amount_i must be less than or equal to state.capturable_i.

- If `capture()` can be called, then `authorize()` must have been called on the same `paymentDetails` first.
- If **`void`**`()` and `reclaim()` can be called on `paymentDetails` input then that input hash must have been previously updated by calling external function `authorize()` (ie, only if a nonzero payment has been collected inside the `PaymentEscrow` contract) **and** the total sum of all `amount` input used in calling `capture()` using that same `paymentDetails` is strictly less than the initial `capturable` amount set in the call to `authorize()`.
- If `refund()` can be called on `paymentDetails` input then the input must correspond to a hash that has been updated in a call to `charge()` or `capture()`.

Remarks

I. A user can give the `PreApprovalTokenCollector` contract infinite allowance but an operator is still required to possess pre-approval for a specific amount to use the allowance to move tokens to the Escrow contract.

II. While a payer can specify themselves as operator, feeReceiver, TokenCollector, or receiver of a payment, there is no incentive for doing so.

- Funds initially come from the payer
- The only payments seen as "legitimate" within the system as a whole are those that are created by a specific operator entity. All other payments entered into the smart contract are ignored.

III. More generally, any third party can choose to interact with this smart contract for their own purposes, separate from the smart contract's intended use. This information should be taken carefully into consideration.

IV. Any custom TokenCollector smart contracts being used with the Escrow smart contract should only implement one action of refund or payment, not both.

Findings

Medium Severity

M-01: `collectTokens()` in `SpendPermissionTokenCollector.sol` Uses input `amount` Instead of `paymentDetails.maxAmount`

Signatures created for use with `SpendPermissionTokenCollector.sol` are meant to sign for the `paymentDetails.maxAmount` instead of the input `amount` that an operator uses at the time of the external call to `collectTokens()`.

As a result, the operator cannot call `authorize()` or `charge()` on an amount that is smaller than the `maxAmount` relayed in the `paymentDetails`.

Recommendation: Change `allowance` in the created `SpendPermission` struct to be `paymentDetails.maxAmount` instead of `amount`.

Status: **Fixed**

The engineering team resolved the finding in commit f4a39f2522d932bdf8ef7aba9bfa02818b4f5e39.

Low Severity

L-01: Possible to Call `authorize()` with `paymentDetails.receiver` Set to `address(0)`

Since the `receiver` is not immediately transferred tokens in a call to `authorize()` it is possible to set it to `address(0)`. Then when `capture()` is called later, most tokens will revert when attempting to call `transfer()` with a destination address of `address(0)`, requiring the payment either be voided or refunded.

Recommendation: Prevent `address(0)` from being used as a `receiver` in `paymentDetails` input.

Status: **Acknowledged**

The engineering team acknowledges the finding and opts not to make the suggested change.