Commerce Payments Audit 1

Coinbase Protocol Security

June 11, 2025

BASE

Contents

Audit Scope

Initial Review Commit: b35b022fe1833708a268d7d1ac45a2cac83a64fb

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

Files:

src/PaymentEscrow.sol
src/IMulticall3.sol
src/IERC3009.sol

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 18th to March 20th.

System Overview

Note: *The following system overview and invariants are described for the protocol presented as of* Audit 2. *Some of the descriptions may not align with the original design of the contracts reviewed in this audit.*

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.

Findings

Medium Severity

M-01: Merchant Can Circumvent Fee Being Taken from Payment

It is possible for a merchant to use the function `capture()` to collect the full `_paymentState.value` without contributing any percentage of the value toward fees, even if a positive feeBps is specified. This can be done by specifying a small enough input value on each call of `capture()` so that the fee calculation in `_distributeTokens()` rounds down to 0. This can be performed iteratively until the entire sum of the `_paymentState.value` is entirely consumed and sent to the capture address. This is possible since the `captureAddress` (aka merchant) can call `capture()` and because `capture()` can be called on a value lower than the `_paymentState.value`. The merchant would have incentive to do so since they are the entity receiving payment, but the operator could also perform this flow.

Recommendation: Add a check to the logic that If bps is nonzero, ensure that feeAmount is nonzero in both the captured amount and the leftover amount or else partial payment is invalid and reverts

Update: The logic has been updated so that now only the operator of a payment can call endpoints `charge()`, `authorize()`, and `capture()`. Since the merchant has no control over the payment details set, this removes incentive for using `capture()` in a way that reduces fees.

Status: **Fixed**

The engineering team resolved the finding in commit 547b55046f042cdb9e2bfff1e6a521d6bbae72fa.

Low Severity

L-01: Buyer Can Set Self as Operator in a Payment

The buyer can also be the operator for a payment. This causes some functions such as `reclaim()` and `void()` to potentially behave inconsistently, as the buyer should only be able to reclaim funds after the capture deadline, but as the operator can call `void()` whenever they choose.

A buyer can set themselves as the `captureAddress` for a payment, too but there is incentive for paying the correct capture address to validate a purchase.

Similarly, a buyer who is also set as an operator for the payment could issue themselves a refund, changing the recorded captured information on-chain.

Recommendation: If there is no security concern with allowing a buyer to reclaim their payment before the capture deadline, then consider removing this as a requirement in `reclaim()` in order to avoid the inconsistency.

Alternatively, consider a system for checking that an operator for a payment is valid, either onchain or offchain. For instance, one solution could be that a user could specify one operator address from a set of addresses whitelisted within the smart contract as allowable operator addresses.

Update: While a payer can set themselves as an operator of a payment through direct interaction with the smart contract, it has been confirmed that payments not entered by expected operator addresses will not be shown within the user interface or interacted with by the legitimate parties of the intended transactions. Thus, any payments added to the Escrow which are directly created by a payer will be ignored by the system as a whole and the inconsistency will not be relevant.

Status: **Fixed**

The engineering team resolved the finding by clarifying the design intent.

L-02: `preApprove()` Can Be Called More than Once on the Same Input

Function `preApprove()` can be called for the exact same `paymentDetails` input more than once. This will update the same `_paymentState[paymentDetailsHash]` state entry with the same information, and emit another event, but nothing will actually change. The issuance of more than one event for the same information may cause confusion.

Recommendation: Prevent `preApprove()` from being called if `isPreApproved` is already set to **`true`**.

Update: `preApprove()` has been removed from the design.

Status: **Fixed**

The engineering team resolved the finding in commit 7d07b1a14764b17360bc2f1ca88b22826478c9c6.

L-03: Checks-Effects-Interactions Pattern Violated

In functions `charge()` and `authorize()`, internal function `_pullTokens()` is called before updating state variables `_paymentState[paymentDetailsHash].captured` and `_paymentState[paymentDetailsHash].authorized` respectively. Function `_pullTokens()` contains an external transfer to the `buyer` address, which could cause a potential inconsistency in state if the token used allows callbacks.

Recommendation: Update the state of `captured` and `authorized` respectively, and emit respective events before calling `_pullTokens()`.

Status: **Fixed**

The engineering team resolved the finding in commit f1d807ea3b032770e19e7cd74c14e9a49853d67e.

L-04: Missing Input Validation

The following input validation may be beneficial to include: - A check to ensure `_multicall3` is nonzero in the `constructor` - A check to ensure `paymentDetails.operator`, `paymentDetails.captureAddress`, `paymentDetails.token` are nonzero in `preApprove()` - A check to ensure paymentDetails.authorizeDeadline is past the current block.timestamp and that paymentDetails.captureDeadline is past authorizeDeadline in function preApprove() - A check to ensure paymentDetails.feeBps is no larger than 10_000 in preApprove() - A check to ensure addresses paymentDetails.token, paymentDetails .buyer, and paymentDetails.captureAddress are nonzero in _pullToken() - A check to ensure paymentDetails.authorizeDeadline is no larger than type(uint48).max in preApprove() and _pullToken()

Recommendation: Add in the checks outlined above to ensure the logic behaves as expected.

Update: The corresponding logic has been removed.

Status: **Fixed**

The engineering team resolved the finding.

Informational

I-01: Typos and Suggested Changes to Comments

The following typos are present within the codebase:

- A comment within the PaymentState struct contains the misspelling “pre-aproved”
- The dev comment above function authorize() contains the misspelling “approva”

Within the comment above event PaymentCharged it might be better to state “Emitted when a payment is *approved* and immediately captured”

Recommendation: Correct the typos and make the suggested changes to the comments listed.

Status: **Fixed**

The engineering team resolved the finding.

I-02: Incompatibility with Fee-on-Transfer Tokens

The logic of authorize() and capture() is incompatible with tokens that take a fee upon transfer, since the value recorded as authorized and captured for the _paymentState mapping may be larger than what is actually transferred to the contract.

Recommendation: Avoid use of fee-on-transfer tokens within the Escrow contract.

Status: **Acknowledged**

The engineering team acknowledges the finding.

I-03: Inconsistency in Type

Within struct `PaymentDetails`, `authorizeDeadline` is type uint256 while `captureDeadline` is type `uint48`.

Recommendation: Make the types of these two deadlines consistent.

Status: **Fixed**

The engineering team resolved the finding.

I-04: Additional Information Can Be Added to Errors

- Error `InvalidSender` may be further clarified by including the address of the valid sender.
- Error `ValueLimitExceeded` may be further clarified by including the limit that was exceeded.

Recommendation: Add the information suggested above to the corresponding errors

Status: **Acknowledged**

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

Gas Optimization

G-01: Perform Checks on `msg.sender` Before Creating `paymentDetailsHash`

In functions `charge()`, `authorize()`, `capture()`, `void()`, and `reclaim()`, the `paymentDetailsHash` is calculated before making checks. If the checks revert, the calculation of `paymentDetailsHash` costs unnecessary extra gas.

Recommendation: Calculate `paymentDetailsHash` after checks are made in each function.

Status: **Fixed**

The engineering team resolved the finding.