Commerce Payments Audit 3

Coinbase Protocol Security

June 11, 2025

Contents

Audit Scope

Initial Review Commit: dc809ed910050100a644b2b2c86bddab08735681

Repository: https://github.com/base/payment-escrow

Files:

ERC3009PaymentCollector.sol
ERC6492SignatureHandler.sol
OperatorRefundCollector.sol
Permit2PaymentCollector.sol
PreApprovalPaymentCollector.sol
SpendPermissionPaymentCollector.sol
TokenCollector.sol
IERC3009.sol
IMulticall3.sol
PaymentEscrow.sol
TokenStore.sol

SLOC: 582

Latest Review Commit: 1801352ba9862e6f3345fd0181a890e7927d6abd

Executive Summary

This report presents the outcomes of our collaborative engagement with the Base team, focusing on the comprehensive evaluation of the Payment Escrow v1 Smart Contracts. The Payment Escrow v1 repository was reviewed from April 15th to April 18th, 2025.

Summary of Changes

This is the third internal review of the Payment Escrow protocol. Between this review and the previous, the following main changes were made.

- Introduction of a TokenStore contract cloning system to hold funds on a per-operator basis, moving the holding of funds out of the Escrow contract;
- Inclusion of a nonReentrant modifier on all external-facing state-changing functions within the Escrow contract;

- Transition to a payer-agnostic hash of paymentInfo in external use as a signature nonce;
- Unchecked blocks for gas optimization (removed at time of remediation).

Properties of the Protocol

The following are properties that are expected to be held by the protocol.

- A payment is uniquely defined by its paymentInfo struct; changing any of this information results in two distinct payments and two distinct payments are well-defined in the protocol.
- Each operator address corresponds to its own unique TokenStore contract address; an address acting as an operator cannot access another operator's TokenStore funds except in the case where it has already paid that amount into the TokenStore contract as a payer.
- The sum of all USDC held by all TokenStore contracts of the system is greater or equal to the sum of all paymentState capturableAmount values pertaining to USDC (may be greater in the case of direct transfers which cannot be prevented).

Remarks

- In the event of a migration from one escrow contract to another, the TokenStore contract instances already in existence cannot be used with the new Escrow contract; similarly, in order to use a new TokenStore clone implementation, a new Escrow contract must be deployed. Each Escrow version with its corresponding TokenStore version should be demarcated within the Escrow documentation.
- For a period of time, funds may be sent through authorize() to addresses which do not yet have a TokenStore contract instance deployed; each TokenStore contract is only deployed after funds are captured or reclaimed/voided/refunded.
- Older versions of the Permit2 contract do not validate the proper msg.sender is using the signature, which allowed anyone to call permitTransferFrom() and consume a validation. Use of older versions of Permit2 contracts will allow users to trigger movement of funds to a TokenStore contract without being properly recorded within the Escrow contract, resulting in users' funds getting stuck in the protocol. It is important that signatures constructed for use with this protocol and Permit2 are only done for deployed Permit2 contracts that use the file at commit cc56ad0f3439c502c246fc5cfcc3db92bb8b7219 or later to avoid this issue. (Example of incompatible Permit2 version: 8b49f88f357663d4b9481eab443be48b94895021)

Findings

Low Severity

L-01: Potential Silent Failure at Level of `_sendTokens()`

In function `_sendTokens()` within the Escrow contract, the function preemptively returns if three conditions are simultaneously met for a low-level call. The call must have that the returned bool `success` is true, that the length of `returnData` is 32, and that `abi.decode(returnData, (bool))` is true.

The remaining logic in the `_sendTokens()` function only performs a subsequent action if the `tokenStore.code.length == 0` on its own, this does not capture other possibilities such as a `tokenStore` contract of nonzero code length, which failed to make the `sendTokens()` call, or which made the call but had an unexpected `returnData` value. One potential instance in which this could occur is if the sender (`tokenStore` contract) or recipient has been blacklisted by the USDC contract.

As a result, it is possible to update the state of the Escrow contract to inaccurately reflect a transfer that has not actually occurred from the `tokenStore` contract.

Recommendation: Handle the potential failure with a revert.

Status: **Fixed**

The engineering team resolved the finding in 76728e450a0c7cbdcaed8be6be0c58645866d3a7.

Informational

I-01: Suggested `_validateFee()` Check

Function `_validateFee()` in the Escrow contract reverts if `feeBps` is positive while `feeReceiver` is 0.

Additionally, if `feeReceiver` is nonzero while `paymentInfo.feeReceiver` is zero, then `feeBps` should be nonzero too, because the operator likely means to take a fee.

Recommendation: The following check should be added to account for this:

```
if (feeReceiver != address(0)&& paymentInfo.feeReceiver == address(0)
&& feeBps == 0)revert;
```

Status: **Acknowledged**

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

I-02: Solady's SafeTransferLib does not check for token contract's existence

There is a subtle difference between the implementation of Solady's SafeTransferLib and OZ's SafeERC20: OZ's SafeERC20 checks if the token is a contract, Solady's SafeTransferLib does not.

Recommendation: Use OpenZeppelin's SafeERC20 contract to include the check for contract existence

Status: **Fixed**

The engineering team switched out Solady's SafeTransferLib with OpenZeppelin's SafeERC20 in to resolve the finding commit c9fd65658ce056be0a8a95db42c2f5ddff663db3.

I-03: Use the latest audited Solady version (v0.1.1)

The Solady v0.1.1 release was created after implementing all fixes during the Coinbase-funded audit of Solady. As such, we should always be using v0.1.1 of Solady for our future development.

Recommendation: Use the Solady v0.1.1 release.

Status: **Fixed**

The engineering team resolved the finding in commit daa2e16f9c23f71bf46f7ee65e562f245f16f934.

I-04: Loss of precision for USDC amounts of under $0.001 on payment fees in `_distributeTokens()` function within PaymentEscrow

Division by large numbers may result in a loss of precision in the result or the result being zero, due to solidity not supporting fractions. Considering USDC could be a payment token and it only supports 6 decimals, amounts under $0.001 could be truncated by the default fee precision of 10,000 and the protocol could lose out on fees.

This is also dependent upon the protocol fee set. For fee rates under 1%, amounts under $0.0001 would be truncated. For fee rates under 0.1%, the truncation increases and would apply to amounts under $0.001.

The amount of fees lost are very low, especially considering the intended use case for this product where transactions will rarely be under $1.

Recommendation: Determine if the truncation outlined above is acceptable by the protocol.

Status: **Acknowledged**

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

I-05: TokenStore allows a version of Solidity that is susceptible to an assembly optimizer bug; though the bug is not present.

In solidity versions 0.8.13 and 0.8.14, there is an optimizer bug where, if the use of a variable is in a separate `assembly` block from the block in which it was stored, the `mstore` operation is optimized out, leading to uninitialized memory.

The code currently does not have a pattern of execution that makes it vulnerable, but the affected solidity versions should be avoided whenever possible. It is noted that the float included within the pragma bumps the version to its most recent.

Recommendation: Since it is confirmed that making older versions of solidity available is unintended, rewrite the Solidity pragma statement to a more modern version.

Status: **Fixed**

The engineering team resolved the finding in commit 3628ea894e6710ac94446765f80fb68bd272e969.

I-06: The `sendTokens()` function in TokenStore is missing the `@return` parameter from its NatSpec.

The sendTokens function in PaymentEscrow is missing a `@return` NatSpec comment.

Recommendation: Include the`@return` parameter.

Status: **Fixed**

The engineering team resolved the finding in commit 56702ec6241fa75501453853a533322801f184c8.

I-07: Natspec for `@param feeReceiver` Can Be Clarified

Consider including a comment instructing to set `feeReceiver` to the `paymentInfo.feeReceiver` value if the original `feeReceiver` is nonzero. As is, it could read as if it should be left as `address(0)` which would cause a revert.

Additionally, use of the word “original” in reference to `paymentInfo.feeReceiver` may be confusing.

Recommendation: Include a clarifying comment to the `feeReceiver` NatSpec.

Status: **Fixed**

The engineering team resolved the finding in commit 40fab91b967b1a02a2a192354685c906fde3ac55.

I-08: Incorrect Casting in `_distributeTokens()`

In function `_distributeTokens()` within the Escrow contract, the input `feeBps` is a `uint16` and input `amount` is a `uint256`. Even so, when calculating `uint256 feeAmount`, the amount is cast as a `uint256` redundantly, while `feeBps` is not cast.

Recommendation: Remove the redundant casting and set `feeBps` appropriately.

Status: **Fixed**

The engineering team resolved the finding in commit 466c7ec3a1a129a4d346fb88bec013ec25d717ee.

I-09: Events Should Be Carefully Checked for Accuracy

It is possible to make an event look as if a payer has made a payment when they have not, if an unreliable `TokenCollector` contract is used. Along with carefully noting the `operator` in an event, `TokenCollector` should be checked to ensure it is not an unexpected contract when processing events.

Recommendation: Both the operating addresses and the TokenCollector contracts they use should be monitored to ensure the expected protocols are in use. Acknowledgement of this finding is sufficient for resolution.

Status: **Acknowledged**

The engineering team acknowledges the finding.

I-10: Note that Function Selector Protection Prevents TokenStore Misuse

It is imperative that the call made to `tokenCollector.collectTokens()` not be replaced with a low-level call to `tokenCollector` in `_collectTokens()` of the Escrow contract. Without the protection imposed by the function selector, an operator's `tokenStore` contract could be used as input for `tokenCollector` allowing an attack vector where all funds can be stolen from that `tokenStore` contract.

Recommendation: Ensure that the call to `tokenCollector.collectTokens()` within the Escrow contract is not replaced with a low-level call in future versions of the contract. Acknowledgement of this finding is sufficient for resolution.

Status: **Acknowledged**

The engineering team acknowledges the finding.

Gas Recommendations

G-01: Cache array lengths outside of loops

Save ~5.5k gas on each use of the ERC6492SignatureHandler collector by caching the length of the array outside of the loop.

```
        for (uint256 i; i < signature.length - 32; i++) {
```

If not cached, the solidity compiler will always read the length of the array during each iteration. That is, if it is a storage array, this is an extra `sload` operation (100 additional extra gas for each iteration except for the first) and if it is a memory array, this is an extra `mload` operation (3 additional gas for each iteration except for the first).

A signature is 65 bytes, so looping will always occur here. As such, we advise implementing the following changes:

```
  function _handleERC6492Signature(bytes memory signature) internal
     returns (bytes memory) {

        // Early return if signature less than 32 bytes

        uint256 sigLength = signature.length;

        if (sigLength < 32) return signature;


        // Early return if signature suffix not ERC-6492 magic value

        bytes32 suffix;

        assembly {

            suffix := mload(add(add(signature, 32), sub(mload(signature),
               32)))

        }

        if (suffix != _ERC6492_MAGIC_VALUE) return signature;


        // Parse inner signature from ERC-6492 format

        bytes memory erc6492Data = new bytes(sigLength - 32);

        for (uint256 i; i < sigLength - 32; ++i) {
            erc6492Data[i] = signature[i];
        }
```

This also enables us to reap the gas optimizations introduced with Solidity 0.8.22, which are otherwise lost due to the comparison value being dynamic rather than a local variable. For more information, read the Solidity 0.8.22 release notes.

Recommendation: Store the state of `signature.length` in a temporary variable as demonstrated above.

Status: **Fixed**

The engineering team resolved the finding in commit 640d1dce9aacd124399cb32b0eabd9f0a47ef872.

G-02: Inconsistent Temporary Variable Use in `_validatePayment()`

In `_validatePayment()` of the Escrow contract, the parameter `paymentInfo.maxAmount` of the `paymentInfo` struct is not set to a temporary local variable when all other used parameters in the `paymentInfo` struct are set to temporary local variables.

Recommendation: Since `paymentInfo.maxAmount` is used more than once, it could be set as a local variable.

Status: **Fixed**

The engineering team resolved the finding in commit b52138170430e27406b6f02c9965c0af9181b844.