Payment Escrow

Security Review

Cantina Managed review by:

Chris Smith, Lead Security Researcher

Cccz, Security Researcher

Jonatas Martins, Associate Security Researcher

May 9, 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

Base is a secure, low-cost, builder-friendly Ethereum L2 built to bring the next billion users onchain.

From Apr 23rd to Apr 26th the Cantina team conducted a review of payment-escrow on commit hash ef53dac6. The team identified a total of **9** issues:

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	1	0	1
Gas Optimizations	1	1	0
Informational	7	4	3
Total	**9**	**5**	**4**

3 Findings

3.1 Low Risk

3.1.1 USDC blacklisted feeReceiver will block the capture

Severity: Low Risk

Context: PaymentEscrow.sol#L284

Description: In the `authorize` → `capture` flow, the operator is responsible for collecting tokens from the payer during authorize and distributing tokens to the receiver and `feeReceiver` during capture:

```
function capture(PaymentInfo calldata paymentInfo, uint256 amount, uint16 feeBps, address feeReceiver)
    external
    nonReentrant
    onlySender(paymentInfo.operator)
    validAmount(amount)
{
    // ...

    // Transfer tokens to receiver and fee receiver
    _distributeTokens(paymentInfo.token, paymentInfo.receiver, amount, feeBps, feeReceiver);
}
```

Consider the token is USDC, if the feeReceiver is added to the USDC blacklist between authorize and capture, `capture()` call will always fail due to the feeReceiver cannot receive USDC tokens, resulting in the receiver also being unable to receive USDC tokens. And after the authorization expires, the payer can call `reclaim()` to retrieve these escrowed tokens.

Recommendation: It is recommended that when sending tokens to the feeReceiver fails, just skip it and send tokens to the receiver.

Coinbase: After exploring possible mechanistic options to avoid this scenario in the core protocol, we decided the added complexity wasn't justified by the reasonably small likelihood and impact of this edge case. Alternatively, we've documented this possibility in the README documentation for now, and identified an existing mechanism that allows operators who are more concerned with liveness than with fee capture to use the protocol in a way that mitigates this as a risk at the expense of slightly less secure handling of their fees. README update PR 70.

Cantina Managed: Acknowledged.

3.2 Gas Optimization

3.2.1 Unnecessary reset of `payer` information

Severity: Gas Optimization

Context: TokenCollector.sol#L67

Description: The function `_getHashPayerAgnostic` in `TokenCollector` sets `paymentInfo.payer` to the `payer` value at the end of its execution. Since this is a view function, resetting this value is unnecessary and consumes extra gas.

Recommendation: Remove the line that sets `paymentInfo.payer` to `payer`.

Coinbase: We chose to add an inline comment above `paymentInfo.payer = payer` explaining the importance of resetting this memory location. Included in PR 65.

Cantina Managed: Fix verified.

3.3 Informational

3.3.1 Add `preApprovalExpiry` check in `PreApprovalPaymentCollector.preApprove()`

Severity: Informational

Context: PreApprovalPaymentCollector.sol#L39-L54

Description: In `PreApprovalPaymentCollector.preApprove()`, if the current time exceeds `preApprovalExpiry`, the `preApproval` will be invalid due to expiration.

Recommendation: It is recommended to add `preApprovalExpiry` check in `PreApprovalPaymentCollector.preApprove()`:

```
  function preApprove(PaymentEscrow.PaymentInfo calldata paymentInfo) external {
      // Check sender is buyer
      if (msg.sender != paymentInfo.payer) revert PaymentEscrow.InvalidSender(msg.sender, paymentInfo.payer);
+     if (block.timestamp >= paymentInfo.preApprovalExpiry) revert
↪  PaymentEscrow.AfterPreApprovalExpiry(block.timestamp, paymentInfo.preApprovalExpiry);
      // Check has not already pre-approved
      bytes32 paymentInfoHash = paymentEscrow.getHash(paymentInfo);
      if (isPreApproved[paymentInfoHash]) revert PaymentAlreadyPreApproved(paymentInfoHash);

      // Check has not already collected
      (bool hasCollectedPayment,,) = paymentEscrow.paymentState(paymentInfoHash);
      if (hasCollectedPayment) revert PaymentAlreadyCollected(paymentInfoHash);

      // Set payment as pre-approved
      isPreApproved[paymentInfoHash] = true;
      emit PaymentPreApproved(paymentInfoHash);
  }
```

Coinbase: Accepted recommendation. Change included in PR 65.

Cantina Managed: Fix verified.

3.3.2 Potential ERC1271 replay issue

Severity: Informational

Context: TokenCollector.sol#L65

Description: Before collecting tokens, the payer is required to provide a signature containing `paymentInfo` according to the different `TokenCollector`, where `paymentInfo` will consider the payer as `address(0)` to avoid potential bugs.

```
function _getHashPayerAgnostic(PaymentEscrow.PaymentInfo memory paymentInfo) internal view returns (bytes32) {
    address payer = paymentInfo.payer;
    // paymentInfo.payer = address(0);
    bytes32 hashPayerAgnostic = paymentEscrow.getHash(paymentInfo);
    paymentInfo.payer = payer;
    return hashPayerAgnostic;
}
```

If payer is a smart wallet, and the user has multiple smart wallets, not including payer in the hash may be affected by the ERC1271 replay issue. ERC3009 (mock version) and `SpendPermission` are not affected because they will include payer in further rehashing.

```
    bytes32 structHash =
        keccak256(abi.encode(RECEIVE_WITH_AUTHORIZATION_TYPEHASH, from, to, value, validAfter, validBefore,
        ↪  nonce)); // @sr : from is payer address

    bytes32 digest = keccak256(abi.encodePacked("\x19\x01", DOMAIN_SEPARATOR(), structHash));

    require(SignatureCheckerLib.isValidSignatureNow(from, digest, signature), "MockERC3009: invalid signature");
// ...
function getHash(SpendPermission memory spendPermission) public view returns (bytes32) {
    return _hashTypedData(
        keccak256(
            abi.encode(
                SPEND_PERMISSION_TYPEHASH,
                spendPermission.account,              // account is payer address
                spendPermission.spender,
                spendPermission.token,
                spendPermission.allowance,
                spendPermission.period,
                spendPermission.start,
                spendPermission.end,
                spendPermission.salt,
                keccak256(spendPermission.extraData)
            )
        )
    );
}
```

But `Permit2` will not include payer (i.e. `owner`) in further rehashing:

```
bytes32 public constant _TOKEN_PERMISSIONS_TYPEHASH = keccak256("TokenPermissions(address token,uint256
↪  amount)");
// ...
bytes32 public constant _PERMIT_TRANSFER_FROM_TYPEHASH = keccak256(
    "PermitTransferFrom(TokenPermissions permitted,address spender,uint256 nonce,uint256
    ↪  deadline)TokenPermissions(address token,uint256 amount)"
);
// ...
function hash(ISignatureTransfer.PermitTransferFrom memory permit) internal view returns (bytes32) {
    bytes32 tokenPermissionsHash = _hashTokenPermissions(permit.permitted);
    return keccak256(
        abi.encode(_PERMIT_TRANSFER_FROM_TYPEHASH, tokenPermissionsHash, msg.sender, permit.nonce,
        ↪  permit.deadline)
    );
}
```

Consider Alice has two smart wallets, wallet A and wallet B, each with 1000 USDC approved for `Permit2`. When Alice signs the `permit2` message with wallet A, the operator is allowed to collect 1000 USDC from wallet A. Meanwhile, the signature is also valid for wallet B, and the operator can also collect 1000 USDC from wallet B.

```
function _permitTransferFrom(
    PermitTransferFrom memory permit,
    SignatureTransferDetails calldata transferDetails,
    address owner,
    bytes32 dataHash,
    bytes calldata signature
) private {
    uint256 requestedAmount = transferDetails.requestedAmount;

    if (block.timestamp > permit.deadline) revert SignatureExpired(permit.deadline);
    if (requestedAmount > permit.permitted.amount) revert InvalidAmount(permit.permitted.amount);

    _useUnorderedNonce(owner, permit.nonce);

    signature.verify(_hashTypedData(dataHash), owner);

    ERC20(permit.permitted.token).safeTransferFrom(owner, transferDetails.to, requestedAmount);
}
// ...
function verify(bytes calldata signature, bytes32 hash, address claimedSigner) internal view {
    bytes32 r;
    bytes32 s;
    uint8 v;
    if (claimedSigner.code.length == 0) {
        if (signature.length == 65) {
            (r, s) = abi.decode(signature, (bytes32, bytes32));
            v = uint8(signature[64]);
        } else if (signature.length == 64) {
            // EIP-2098
            bytes32 vs;
            (r, vs) = abi.decode(signature, (bytes32, bytes32));
            s = vs & UPPER_BIT_MASK;
            v = uint8(uint256(vs >> 255)) + 27;
        } else {
            revert InvalidSignatureLength();
        }
        address signer = ecrecover(hash, v, r, s);
        if (signer == address(0)) revert InvalidSignature();
        if (signer != claimedSigner) revert InvalidSigner();
    } else {
        bytes4 magicValue = IERC1271(claimedSigner).isValidSignature(hash, signature);
        if (magicValue != IERC1271.isValidSignature.selector) revert InvalidContractSignature();
    }
}
```

Currently, wallets mostly implement replay protection to avoid being affected by the issue. Note that for wallets that support ERC7739. If the sender is considered safe, it will skip the rehash which contains the wallet address(i.e. payer), so these wallets that consider Permit2 to be a safe caller are affected by this issue.

```
function _erc1271IsValidSignature(bytes32 hash, bytes calldata signature)
    internal
    view
    virtual
    returns (bool)
{
    return _erc1271IsValidSignatureViaSafeCaller(hash, signature)
        || _erc1271IsValidSignatureViaNestedEIP712(hash, signature)
        || _erc1271IsValidSignatureViaRPC(hash, signature);
}
```

Recommendation: It is not recommended to implement any fixes as this is more the responsibility of the wallet side, it is recommended to document the issue to inform users.

Coinbase: Acknowledged, no operation. CBSW implements replay-safety, and other wallets are responsible for this on their own behalf.

Cantina Managed: Acknowledged.

3.3.3 Transient Reentrancy Guard does not provide gas savings on Base

Severity: Informational

Context: PaymentEscrow.sol#L19

Description: The Solady Transient Reentrancy protection contract operates (almost) exactly the same on Base as the regular Reentrancy Guard contract as it only uses transient storage for `chain.id == 1`.

Recommendation: If this is establishing a pattern for use on Mainnet, then it is fine. However, if there is no intention to use Transient storage, then changing to the regular Reentrancy guard can save some contract code size and make the code slightly more comprehensible. If the intention is to use Transient storage for reentrancy on Base, then the `_useTransientReentrancyGuardOnlyOnMainnet()` function needs to be overwritten in a way to enable it on Base.

Coinbase: Accepted recommendation to override the solady internal function to return `false`: use transient reentrancy guard on all chains. We accept the possible restrictions this creates for chains that don't support `tstore` to use this protocol as-written. Change included in PR 65.

Cantina Managed: Fix turns on transient storage on all chains and addresses this issue.

3.3.4 Import and Remapping simplifications

Severity: Informational

Context: remappings.txt#L1, TokenStore.sol#L4-L5

Description: There are a couple of places OZ imports/remappings could be simplified.

Recommendation: In `TokenStore`, the imports could be combined for simplicity:

```
import {SafeERC20, IERC20} from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
```

In `remappings.txt`, `@openzeppelin/contracts/security/=lib/spend-permissions/lib/magicspend/lib/openzeppeli` `contracts/contracts/utils/` does not seem to be used and could be accessed using the standard OZ remapping: `@openzeppelin/contracts/=lib/spend-permissions/lib/openzeppelin-contracts/contracts/`.

Coinbase: Accepting recommendation, fix included in PR 65.

Cantina Managed: Fix verified.

3.3.5 Operators are highly trusted actors

Severity: Informational

Context: PaymentEscrow.sol#L475-L500

Description: The protocol places a high amount of trust on the operators including expecting them to process all payments and allowing them to define the exact fee and potentially even the `feeReceiver`. Great care should be taken by protocols in selecting, monitoring and securing the operator addresses.

The `_MAX_FEE_BPS` is 100% (100_00 bps) allows the payer to configure their payment to allow a fee up to 100% of the payment. When the operator sends the payment they are able to set a fee between the min and max. Further, if the payer did not specify the `feeReceiver`, then the operator can also specify an address it owns as the receiver, allowing it to capture the full payment. This means a malicious or compromised operator could profit when there are large payments or many small payments and they can set the `feeReceiver` to an address they control.

Recommendation: As operators are essentially permissionless (upon agreement between the payer and payee), it is extremely important systems are built to protect payers and payee from misbehaving or compromised operators and that the trust assumptions of working with an operator are well documented for all parties (payer, operator, and payee).

This means there need to be systems to monitor operators and prevent bad actors from being selected by payers. Wherever possible, UI and infrastructure systems should set thoughful parameters for `maxFeeBps` and `feeReceiver`.

Coinbase: Acknowledged. We will be including relevant documentation outlining the security posture with respect to operators via in-repo documentation and additional longer formats such as a technical blog post. Partially addressed in PR 70.

Cantina Managed: Acknowledged.

3.3.6 Reset `paymentInfoHash` to prevent reuse

Severity: Informational

Context: PreApprovalPaymentCollector.sol#L66

Description: The `PreApprovalPaymentCollector.preApprove` function sets `isPreApproved` to true for a payment info hash. While `isPreApproved` is validated in `_collectTokens` to process the transfer, it isn't reset afterward. Although this isn't currently an issue since `PaymentEscrow` controls the calls, the `isPreApproved` status could potentially be reused if future implementations have problems. As a best practice, setting `isPreApproved` to false after validation would prevent any possible reuse.

Recommendation: Consider setting isPreApproved to false after validation in _collectTokens, or change the implementation to an enum status to prevent the isPreApproved mapping from being reused. Example:

```
  /// @inheritdoc TokenCollector
  /// @dev Requires pre-approval for a specific payment and an ERC-20 allowance to this collector
  function _collectTokens(
      PaymentEscrow.PaymentInfo calldata paymentInfo,
      address tokenStore,
      uint256 amount,
      bytes calldata
  ) internal override {
      // Check payment pre-approved
      bytes32 paymentInfoHash = paymentEscrow.getHash(paymentInfo);
      if (!isPreApproved[paymentInfoHash]) revert PaymentNotPreApproved(paymentInfoHash);
+     isPreApproved[paymentInfoHash] = false;
      // Transfer tokens from payer directly to token store
      SafeERC20.safeTransferFrom(IERC20(paymentInfo.token), paymentInfo.payer, tokenStore, amount);
  }
```

Coinbase: We chose to comment the choice to explicitly NOT reset this value, given that the `PaymentEscrow` enforces unique, single-use payments. Included in PR 65.

Cantina Managed: Fix verified.

3.3.7 The `getHash` function doesn't follow EIP712 hashing

Severity: Informational

Context: PaymentEscrow.sol#L365

Description: According to the EIP712 documentation, all hashStructs must follow this pattern:

```
hashStruct(s : S) = keccak256(typeHash ‖ encodeData(s))` where `typeHash = keccak256(encodeType(typeOf(s)))
```

In the `PaymentEscrow` contract, the `getHash` function deviates from this pattern. Instead, it only returns the hash of the encoded data, as shown in the implementation:

```
function getHash(PaymentInfo calldata paymentInfo) public view returns (bytes32) {
        //@audit the paymentInfoHash follow EIP712 hashStruct
    bytes32 paymentInfoHash = keccak256(abi.encode(PAYMENT_INFO_TYPEHASH, paymentInfo));
    //@audit the return doesn't follow EIP712 hashStruct
    return keccak256(abi.encode(block.chainid, address(this), paymentInfoHash));
}
```

Recommendation: Consider add another constant variable for the typeHash as the return of the `getHash` function, like this:

```
+ bytes32 public constant PAYMENT_INFO_HASH = keccak256(
+     "PaymentInfoHash(uint256 chainid,address contract,PaymentInfo paymentInfo)"
+ );

  function getHash(PaymentInfo calldata paymentInfo) public view returns (bytes32) {
-     bytes32 paymentInfoHash = keccak256(abi.encode(PAYMENT_INFO_TYPEHASH, paymentInfo));
-     return keccak256(abi.encode(block.chainid, address(this), paymentInfoHash));
+     return keccak256(abi.encode(PAYMENT_INFO_HASH, PAYMENT_INFO_TYPEHASH, block.chainid, address(this),
↪  paymentInfo));
  }
```

Coinbase: This was actually intentional. We'll revisit this design decision, but we were concerned that making the hash 712 would confuse developers that they should have users sign the PaymentInfo instead of something deriving from a payment collector.

Cantina Managed: Acknowledged.