



Coinbase: x402 Batch Settlement Security Review

Cantina Managed review by:

Sujith Somraaj, Lead Security Researcher

Cryptara, Security Researcher

May 5, 2026

Contents

1	Introduction	2
1.1	About Cantina	2
1.2	Disclaimer	2
1.3	Risk assessment	2
1.3.1	Severity Classification	2
2	Security Review Summary	3
2.1	Scope	3
3	Findings	4
3.1	Medium Risk	4
3.1.1	Constant <code>DEPOSIT_WITNESS_TYPE_STRING</code> violates EIP-712 alphabetical ordering	4
3.2	Low Risk	4
3.2.1	USDC/USDT blacklist of payer or receiver permanently traps escrow	4
3.2.2	Function <code>initiateWithdraw</code> missing <code>nonReentrant</code> modifier	5
3.2.3	Event <code>ChannelCreated</code> can be re-emitted for the same <code>channelId</code>	6
3.3	Informational	6
3.3.1	Claim-vs-finalizeWithdraw race under censorship	6
3.3.2	Spec/code drift on <code>channelId</code> derivation	7
3.3.3	Partial refund does not clear pending withdrawal	7
3.3.4	Voucher signatures have no expiry; remain valid across top-ups on the same <code>channelId</code>	8
3.3.5	Deposit events emitted before token-receipt verification	9
3.3.6	Function <code>refund()</code> invalidates pre-signed <code>refundWithSignature()</code> authorizations	10
3.3.7	Empty-batch branch in <code>getClaimBatchDigest()</code> is dead code on the on-chain path	11
3.3.8	Param caller in <code>IDepositCollector.collect</code> is unused	11

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 level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

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 are 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 27th to Apr 29th the Cantina team conducted a review of [x402](#) on commit hash [ca60063c](#). The team identified a total of **12** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	1	1	0
Low Risk	3	2	1
Gas Optimizations	0	0	0
Informational	8	6	2
Total	12	9	3

2.1 Scope

The security review had the following components in scope for [x402](#) on commit hash [ca60063c](#):

```
contracts/evm/src
├── x402BatchSettlement.sol
├── interfaces
│   └── IDepositCollector.sol
├── periphery
│   ├── DepositCollector.sol
│   ├── ERC3009DepositCollector.sol
│   └── Permit2DepositCollector.sol
```

3 Findings

3.1 Medium Risk

3.1.1 Constant DEPOSIT_WITNESS_TYPE_STRING violates EIP-712 alphabetical ordering

Severity: Medium Risk

Context: Permit2DepositCollector.sol#L25-L27

Description: Permit2DepositCollector.sol defines:

```
string public constant DEPOSIT_WITNESS_TYPE_STRING =  
    "DepositWitness witness)TokenPermissions(address token,uint256  
    ↪ amount)DepositWitness(bytes32 channelId)";
```

Permit2 prepends its stub PermitWitnessTransferFrom(TokenPermissions permitted,address spender,uint256 nonce,uint256 deadline,to this string and hashes the concatenation to derive the typehash used in permitWitnessTransferFrom.

The resulting on-chain encoded type is:

```
PermitWitnessTransferFrom(TokenPermissions permitted,address spender,uint256  
↪ nonce,uint256 deadline,DepositWitness witness)TokenPermissions(address token,uint256  
↪ amount)DepositWitness(bytes32 channelId)
```

EIP-712's definition of `encodeType` requires that the set of referenced struct types be **"collected, sorted by name and appended to the encoding."** The two referenced types here are `DepositWitness` and `TokenPermissions`. Lexicographically, D (0x44) < T (0x54), so the spec-mandated order is `DepositWitness` first, then `TokenPermissions`.

Spec-compliant signing libraries (ethers `signTypedData`, viem `signTypedData`, MetaMask `eth_signTypedData_v4`, Coinbase Wallet, hardware wallets) accept a JSON types object and reconstruct the encoded type by sorting referenced types alphabetically.

They will sign a digest derived from:

```
PermitWitnessTransferFrom(...,DepositWitness witness)DepositWitness(bytes32  
↪ channelId)TokenPermissions(address token,uint256 amount)
```

Because the two type strings differ, the resulting typehashes and therefore the EIP-712 digests differ. Permit2 will compute a different message hash than the one the wallet signed, `_signatureBased.verify()` returns false, and `permitWitnessTransferFrom` reverts with `InvalidSigner`. No deposit through a normal wallet flow can succeed via this collector.

The existing `x402ExactPermit2Proxy` and `x402UptoPermit2Proxy` are unaffected because their witness struct is named `Witness` (W = 0x57 > T = 0x54), so `TokenPermissions` already precedes `Witness` alphabetically. The bug is specific to choosing a witness struct name that sorts before `TokenPermissions`, which `DepositWitness` does.

Recommendation: Reorder the witness type string so the referenced struct definitions appear in alphabetical order:

```
string public constant DEPOSIT_WITNESS_TYPE_STRING =  
    "DepositWitness witness)DepositWitness(bytes32 channelId)TokenPermissions(address  
    ↪ token,uint256 amount)";
```

Coinbase: Fixed in PR 1950.

Cantina Managed: Fix verified.

3.2 Low Risk

3.2.1 USDC/USDT blacklist of payer or receiver permanently traps escrow

Severity: Low Risk

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

Description: The contract escrows USDC/USDT for a payer and receiver. Every payout path (settle to the receiver, finalizeWithdraw and refund to the payer) sends tokens to addresses fixed in ChannelConfig at channel creation.

USDC and USDT both maintain blacklists. A transfer to a blacklisted address reverts. There's no admin, no rescue function, no way to redirect a payout, and no setReceiver or claimTo. The receiver and payer addresses are hashed into channelId, so they can't be migrated either. receivers[r][token] is keyed by address, so every channel sharing a blacklisted receiver freezes at once.

Once either party gets blacklisted, the funds in that channel are stuck. The honest counterparty has no path to recover their share.

Worth flagging: Coinbase's own audit checklist (SOL-AM-DOSA-3) calls out this exact scenario, citing Real Wagmi 2 as precedent. USDC is the documented default asset for this contract.

Proof of concept: Two ways this plays out:

Receiver blacklisted after claiming. Receiver calls claim(), which bumps receivers[R][USDC].totalClaimed. Then R gets blacklisted. Anyone calling settle(R, USDC) reverts inside USDC.transfer. The claimed funds sit in the contract with no way out.

Payer blacklisted during the withdraw window. Payer deposits, then calls initiateWithdraw(). Payer gets blacklisted before the delay elapses. finalizeWithdraw(), refund(), and refundWithSignature() all revert because they pay the blacklisted address. Escrow is stranded.

Remediation: I see 3 options:

1. Add settleTo(receiver, token, recipient, sig) and equivalents for withdraw and refund. The relevant authorizer signs an EIP-712 redirect to a fresh address. No admin, trust model intact.
2. A bilateral-consent rescue path that requires signatures from both payerAuthorizer and receiverAuthorizer. Either side can block griefing.
3. If neither is acceptable, at minimum document the blacklist exposure in NatSpec and the README so integrators know what they're signing up for.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.2.2 Function initiateWithdraw missing nonReentrant modifier

Severity: Low Risk

Context: x402BatchSettlement.sol#L299

Description: Function initiateWithdraw() is declared as external without the nonReentrant modifier, while every other state-changing entrypoint in the contract (deposit, claim, settle, finalizeWithdraw, refund, etc.) is guarded.

Although the function itself performs no external calls and is unlikely to be exploitable in isolation, the missing guard breaks the consistent reentrancy posture of the contract. If future changes introduce a token hook, callback, or cross-function path that reaches initiateWithdraw during another guarded call's execution, the inconsistency could become exploitable (for example, opening a withdrawal under stale balance/totalClaimed state observed mid-claim).

Recommendation: Add the nonReentrant modifier to initiateWithdraw() so the withdrawal flow has uniform reentrancy protection alongside finalizeWithdraw() and refund():

```
- function initiateWithdraw(ChannelConfig calldata config, uint128 amount) external {
+ function initiateWithdraw(ChannelConfig calldata config, uint128 amount) external
  ↪ nonReentrant {
```

Coinbase: Fixed in PR 1950.

Cantina Managed: Fix verified.

3.2.3 Event ChannelCreated can be re-emitted for the same channelId

Severity: Low Risk

Context: `x402BatchSettlement.sol#L209-L212`

Description: The `deposit()` function detects the "first deposit" on a channel using `ch.balance == amount && ch.totalClaimed == 0`. This heuristic assumes the channel state is freshly initialized, but both fields can legitimately return to zero after creation: if the payer deposits, then fully refunds before any claim or settlement occurs, `ch.balance` is decremented back to 0 and `ch.totalClaimed` remains 0.

A subsequent deposit on the same `channelId` then satisfies the predicate again and re-emits `ChannelCreated(channelId, config)`.

Function `finalizeWithdraw()` produces the same shape (full withdrawal of an unclaimed channel resets balance to 0 with `totalClaimed == 0`).

The contract state is unaffected, but indexers, subgraphs, and off-chain monitors that treat `ChannelCreated` as a one-shot lifecycle event (e.g., to bootstrap channel records, dedupe creation, or trigger notifications) can double-count or overwrite metadata. It also weakens any invariant downstream consumers might rely on, that one `channelId` maps to exactly one `ChannelCreated`.

Recommendation: Stop inferring creation from balances and just track it explicitly. A single bool on `ChannelState` is enough:

```
ChannelState storage ch = channels[channelId];
//...
if (!ch.created) {
    ch.created = true;
    emit ChannelCreated(channelId, config);
}
```

Now the event fires exactly once per `channelId`, no matter how many refund/withdraw/redeposit cycles the channel goes through.

Coinbase: Fixed in `10b35d3`. Instead of gating `ChannelCreated` behind a stored flag, we added a matching `ChannelClosed` event that fires when a `refund()` or `finalizeWithdraw()` leaves the channel with `balance == 0 && totalClaimed == 0`. This makes the lifecycle explicit on-chain: every re-emission of `ChannelCreated` is now preceded by a `ChannelClosed` for the same `channelId`, so indexers can pair them up and treat each create/close cycle as a distinct epoch rather than a duplicate.

Cantina: Verified fix.

3.3 Informational

3.3.1 Claim-vs-finalizeWithdraw race under censorship

Severity: Informational

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

Description: `claim()` and `finalizeWithdraw()` are independent entry points but both read and write `ch.balance / ch.totalClaimed`. Since vouchers live off-chain until claimed, the contract can't see outstanding receiver entitlement when the payer initiates withdraw, `totalClaimed` reflects only what's been claimed onchain so far.

That opens a race during the withdraw window:

- A payer can sign a voucher, hand it to the receiver, and call `initiateWithdraw()` in the same block. The available withdraw amount is computed against the current `totalClaimed`, not against any voucher the receiver is holding.
- If the receiver gets `claim()` onchain before `finalizeWithdraw`, `totalClaimed` rises and the payer's withdraw is capped to whatever remains. Receiver wins.
- If the receiver can't transact for the full window, RPC outage, sanctioned address, builder censorship, infra failure, `finalizeWithdraw` lands first, drains the balance, and the receiver's later `claim` reverts with `ClaimExceedsBalance`. The voucher becomes worthless.

Proof of concept:

1. Channel open with `withdrawDelay = 900`. Payer deposits \$100.
2. Payer signs voucher for \$80, receiver delivers the service.
3. Payer calls `initiateWithdraw()` immediately. Contract sees `totalClaimed = 0`, so available looks like \$100.
4. Receiver's claim infra is offline / censored for the full 15 minutes.
5. Payer calls `finalizeWithdraw()`. Balance drained.
6. Receiver comes back online, calls `claim(voucher)` → reverts `ClaimExceedsBalance`.
7. Payer kept the service and the money.

Remediation: Mostly documentation and tooling:

1. Make the receiver's claim-watching responsibility explicit in NatSpec and integrator docs. Ship a reference claim-watcher in the SDK. Recommend `withdrawDelay` well above the 900s floor for high-value channels (hours or days).
2. Optional: add a `commitVoucher(channelId, maxAmount, sig)` primitive. Cheap storage write, no transfer. `finalizeWithdraw` then caps to `balance - max(totalClaimed, voucherCommitment)`. Lets the receiver reserve a voucher's value in one tx without doing the full claim, neutralizes the race even under temporary censorship.

Coinbase: Fixed in [PR 1950](#).

Cantina Managed: [PR 1950](#) addresses this by adding NatSpec documentation for the race condition and introducing a `x402-batch-settlement-implementers.md` implementers guide. The SDK will ship a reference implementation that prescribes `withdrawDelay` values well above the 900s floor and includes mechanisms to ensure receivers land claims before a withdraw window closes.

3.3.2 Spec/code drift on channelId derivation

Severity: Informational

Context: `x402BatchSettlement.sol#L18-L19`

Description: Spec (`specs/schemes/deferred/scheme_batch_settlement_evm.md` L7, 23, 86, 332, 403) defines `channelId = keccak256(abi.encode(channelConfig))`. The contract (`x402BatchSettlement.sol:403-407`) uses `_hashTypedDataV4(CHANNEL_CONFIG_TYPEHASH, ...)`. The contract is correct, EIP-712 binds the ID to chain + verifying contract, which is what INV-006 asserts.

```
specId      = keccak256(abi.encode(channelConfig))
///  
contractId = _hashTypedDataV4(keccak256(abi.encode(CHANNEL_CONFIG_TYPEHASH, ...)))
```

Remediation: Docs and test only, no contract change:

1. Update spec (L7, 23, 86, 332, 403) to use the EIP-712 form.
2. Fix contract NatSpec L18.
3. Fix fork test L86 to match contract derivation.

Coinbase: Fixed in [PR 2051](#).

Cantina Managed: Fix verified.

3.3.3 Partial refund does not clear pending withdrawal

Severity: Informational

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

Description: The specification and the implementation diverge on how refunds interact with pending timed withdrawals.

Spec (scheme_batch_settlement_evm.md L449) states: *"Any successful refund clears a pending timed withdrawal."*

The contract (x402BatchSettlement.sol:518-526) only clears the pending withdrawal when the refund amount is greater than or equal to pws.amount. Smaller refunds decrement pws.amount but leave the withdrawal active and the timer running.

Test cases at x402BatchSettlement.t.sol:990, 1009, and 1034 confirm this is the intended behavior. Implementing the spec literally would let a receiver grief the payer by submitting repeated 1-wei refunds, indefinitely resetting the payer's exit timer, a denial-of-service against the payer's unilateral liveness guarantee (INV-014). The contract's NatSpec at L501 already describes the behavior accurately ("reduces or clears").

The code is correct; the specification is the source of the drift. The risk is integrator confusion: external implementers and auditors reading the spec will see a contradiction with onchain behavior.

Proof of concept:

```
pendingWithdrawals[channelId].amount = 100
receiver calls refund(channelId, 30)

Spec:      pending withdrawal cleared, timer reset
Contract: pws.amount = 70, timer unchanged
```

Following the spec literally produces a grieving primitive: the receiver repeatedly calls refund(channelId, 1), resetting the payer's withdrawal each time and preventing exit indefinitely.

Remediation: Specification update only; no contract changes required.

1. Revise spec L449 to: *"A refund reduces a pending timed withdrawal by the refunded amount. The withdrawal is cleared only when fully refunded."*
2. Optional: emit distinct WithdrawReduced and WithdrawCancelled events so integrators can distinguish the two outcomes without inspecting storage deltas.

Coinbase: Fixed in PR 2051.

Cantina Managed: Fix verified.

3.3.4 Voucher signatures have no expiry; remain valid across top-ups on the same channelId

Severity: Informational

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

Description: The Voucher struct (x402BatchSettlement.sol:62-65) contains only (channelId, maxClaimableAmount). There is no deadline, nonce, or epoch counter. processVoucherClaim (L470-499) enforces only the cumulative ceiling and monotonicity of totalClaimed. As a result, any voucher signed by the payer remains valid for the full lifetime of the associated channelId.

The spec (scheme_batch_settlement_evm.md:444-450) describes the channel lifecycle as long-lived: deposits and **top-ups** accumulate into the same channel.balance under the same channelId, and the only documented way to "reset" a channel is to fully withdraw and migrate to a new ChannelConfig (different salt or other field). There is no on-chain primitive for invalidating outstanding vouchers without that migration.

This produces two exposures:

Voucher unused ceiling persists across top-ups. Because totalClaimed is cumulative per channelId, any unused headroom on a previously-signed voucher carries forward into later top-ups. A voucher signed for max=200 against which only 50 has been claimed still authorizes 150 more - even after the payer has withdrawn the remaining balance and later topped up for an unrelated purpose. The receiver can sweep the new escrow without the payer signing again.

Unbounded blast radius on payer key compromise. A stolen payer signing key produces a voucher signed with maxClaimableAmount = type(uint128).max that is valid forever, against any current or

future escrow in the affected `channelId`. The contract has no on-chain revocation primitive: no per-channel epoch, no voucher expiry, no allow/deny list. Recovery requires fully draining the channel and migrating to a new `ChannelConfig`.

Severity: LOW. The cumulative-claim model is intentional and replay-safe within a single voucher's lifetime. The intended usage pattern (tight `maxClaimableAmount`, channel migration when intent changes) avoids the issue. The risk surfaces under long-lived channel reuse, loose ceiling signing, and key rotation.

Proof of concept:

```
// Initial deposit
ChannelConfig memory cfg = ChannelConfig({
    payer: P, payerAuthorizer: address(0), receiver: R, receiverAuthorizer: address(0),
    token: USDC, withdrawDelay: 900, salt: SALT
});
bytes32 chId = settlement.getChannelId(cfg);
settlement.deposit(cfg, 100e6, collector, "");

// Payer signs a generous voucher for the ongoing relationship
bytes memory sig = sign(P, voucherDigest(chId, 200e6));

// Receiver claims only 50e6, settles
settlement.claim(VoucherClaim({voucher: Voucher(cfg, 200e6), signature: sig,
    ↪ totalClaimed: 50e6}));
settlement.settle(R, USDC);

// Payer withdraws the remaining unclaimed balance
settlement.initiateWithdraw(chId, 50e6);
// ... withdrawDelay elapses ...
settlement.finalizeWithdraw(chId);

// Later: payer tops up the same channel for a new purpose
settlement.deposit(cfg, 100e6, collector, "");

// The original voucher is still valid. Receiver claims another 150e6 against the new
    ↪ escrow.
settlement.claim(VoucherClaim({voucher: Voucher(cfg, 200e6), signature: sig,
    ↪ totalClaimed: 200e6}));
```

The receiver captures the new escrow against a previously-signed voucher whose unused ceiling carried forward across the withdraw and top-up. The same primitive applies to any voucher signed by a compromised key: it remains valid against all future escrow on the affected `channelId`.

Remediation: Three options, in order of preference:

1. **Add a deadline field to Voucher.** Revert in `_processVoucherClaim` when `block.timestamp > voucher.deadline`. Bounds the time window for any signed voucher and limits the blast radius of a key compromise. Minimal protocol change.
2. **Per-channel epoch counter.** Increment an epoch on `finalizeWithdraw` (or expose an explicit `rotateEpoch` call) and bind vouchers to the epoch at signing time. Reverts on mismatch. More invasive but avoids clock-based reasoning.
3. **Document explicitly** in NatSpec and the integrator guide if neither is acceptable. Specifically: sign vouchers with the tightest feasible `maxClaimableAmount`, treat top-ups as an extension of the same authorization context (do not assume voucher hygiene resets), and treat any payer-key compromise as requiring full migration to a new `channelId` - the contract provides no on-chain revocation path.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.3.5 Deposit events emitted before token-receipt verification

Severity: Informational

Context: [x402BatchSettlement.sol#L213](#)

Description: In `deposit()` (`x402BatchSettlement.sol:209-218`), the `ChannelCreated` and `Deposited` events are emitted before the `balanceOf` delta check that verifies the collector actually pulled the tokens:

```
ch.balance += amount;

if (ch.balance == amount && ch.totalClaimed == 0) {
    emit ChannelCreated(channelId, config);
}
emit Deposited(channelId, msg.sender, amount, ch.balance);

uint256 balBefore = IERC20(config.token).balanceOf(address(this));
IDepositCollector(collector).collect(...);
uint256 balAfter = IERC20(config.token).balanceOf(address(this));
if (balAfter != balBefore + amount) revert DepositCollectionFailed();
```

If the collector fails to deliver the expected amount, the transaction reverts and the events are not persisted on-chain. State is safe.

The risk is off-chain: pending-tx parsers, mempool indexers, and some debugger/trace tools surface emitted events from a transaction's execution trace before the final revert is observed. Integrators reading these intermediate signals may show a deposit confirmation that never lands, leading to UI/accounting drift between off-chain views and final on-chain state.

Remediation: Move the event emissions after the post-pull balance check:

```
ch.balance += amount;

uint256 balBefore = IERC20(config.token).balanceOf(address(this));
IDepositCollector(collector).collect(config.payer, config.token, amount, channelId,
    ↪ msg.sender, collectorData);
uint256 balAfter = IERC20(config.token).balanceOf(address(this));
if (balAfter != balBefore + amount) revert DepositCollectionFailed();

if (ch.balance == amount && ch.totalClaimed == 0) {
    emit ChannelCreated(channelId, config);
}
emit Deposited(channelId, msg.sender, amount, ch.balance);
```

Events now only emit on successful, fully-verified deposits.

Coinbase: Fixed in commit `5c3b8ea0`.

Cantina Managed: Fix verified.

3.3.6 Function `refund()` invalidates pre-signed `refundWithSignature()` authorizations

Severity: Informational

Context: `x402BatchSettlement.sol#L362`

Description: Function `_executeRefund()` unconditionally increments `refundNonce` on every non-reverting entry. Both `refund()` and `refundWithSignature()` flow through it. As a result, a receiver (or `receiverAuthorizer`) calling plain `refund(config, 1)` bumps the channel's refund nonce and silently invalidates any in-flight `refundWithSignature` digest the `receiverAuthorizer` had pre-signed for that `channelId` at the prior nonce. The signed authorization becomes unrelayed and must be re-issued.

In practice both callers sit on the receiver side of the trust boundary, so this is internal grieving between the receiver and its delegated authorizer (or two relayers racing) rather than a cross-party attack. It still produces foot-gun behavior: a relay-friendly signed refund can be cancelled by an unrelated direct refund, wasting authorizer effort and potentially stranding a relayer's gas.

Recommendation: Only consume the refund nonce on the signature-gated path. Move the `refundNonce[channelId]++` out of `_executeRefund()` and into `refundWithSignature()` immediately after the nonce/signature checks pass, so direct refund calls do not invalidate outstanding signed authorizations. The replay-prevention property the comment cites is still preserved because only the signed path reads or writes the nonce.

Alternatively, consider documenting this behavior.

Coinbase: Acknowledged. Documented this behavior in [e13ee4f](#).

Cantina Managed: Verified.

3.3.7 Empty-batch branch in `getClaimBatchDigest()` is dead code on the on-chain path

Severity: Informational

Context: `x402BatchSettlement.sol#L440-L442`

Description: In `getClaimBatchDigest()`, there exists an explicit branch for the empty batch case:

```
if (n == 0) {  
    entriesRoot = keccak256("");  
} else {  
    //...  
}
```

Both functions `claimWithSignature()` and `claim()` revert with `EmptyBatch` when `voucherClaims.length == 0` before the digest is ever computed. So no on-chain caller can produce a digest over an empty batch, and the `entriesRoot = keccak256("")` arm cannot influence any state transition.

The branch is reachable when `getClaimBatchDigest()` is called as a view directly (e.g., off-chain signing tools or client SDKs computing what the authorizer would sign). That is the only path it serves today, and it is reachable by external callers only, not by the contract's own logic.

Recommendation: Consider adding a natspec line stating that the empty-batch digest is defined for off-chain use only and is never accepted by the two on-chain functions.

Coinbase: Fixed in [PR 1950](#).

Cantina Managed: Fix verified.

3.3.8 Param caller in `IDepositCollector.collect` is unused

Severity: Informational

Context: `IDepositCollector.sol#L22`

Description: The function `collect()` in `IDepositCollector` interface declares a caller argument, populated by `x402BatchSettlement.deposit` with `msg.sender`.

However, both the deposit collector implementations (`ERC3009DepositCollector` and `Permit2DepositCollector`) leave the slot named, making the parameter redundant.

Recommendation: Consider removing the caller parameter to improve code quality and gas efficiency.

Coinbase: Fixed in [PR 1950](#).

Cantina Managed: Fix verified.