Coinbax EVM Escrow System

Date	July 2026
Auditors	Georgy Kobakhidze, Sergii Kravchenko

1 Executive Summary

This report presents the results of our engagement with **Coinbax** to review their EVM escrow contracts.

The review was conducted from **July 7, 2026** to **July 15, 2026**, by **Georgy Kobakhidze** and **Sergii Kravchenko**.

Coinbax Escrow is a per-workspace stablecoin escrow system providing time-delayed, de-risked transfers with an optional three-way fee split and optional off-chain-attested control gating. Escrow instances are deployed per workspace as EIP-1167 clones that pool funds and read pinned fee configuration from a central factory. The system is described in the System Overview and Security Specification. Importantly, the Coinbax team is at no point in control of the funds-in-flight. Once escrow processes are started, only escrow users, owners and configured privileged actors, such as escrow attestors, may influence the fund flow.

We did not identify any critical or high severity issues, and the findings we raised are limited to configuration, hardening, and documentation. The main protections described in the Security Specification held up under our review. The contracts are well-structured and come with a written specification, threat model, and property-based tests.

2 Scope

Our initial review focused on the coinbax/coinbax-contracts repository at commit `1742290`. During the review, one update was brought into scope:

- `75a2514` (PR #115): collect the platform and workspace fees on `cancelEscrow` and `withdrawRejected` instead of refunding them to the sender.

The in-scope contracts were the EVM escrow contracts `WorkspaceEscrow.sol` and `EscrowFactory.sol` together with their interfaces `IEscrowFactory.sol` and `IERC20WithAuthorization.sol`. The Solana escrow program, the mock token contracts, the test suite, and the deployment scripts were out of scope.

Below are the file hashes as of commit `1742290c7423723d46641bc64c1a6309c61db3b8`:

File	SHA-1 hash
./WorkspaceEscrow.sol	db8f2686ca51ed6efee3504708e5504fd575923d
./EscrowFactory.sol	794657e080592d384a5a63836b7e958af705403e
./interfaces/IEscrowFactory.sol	5d60102020cfef1bcad4c67de2afc32f9ff0f99b

2.1 Objectives

Together with the **Coinbax** team, we identified the following priorities for this review:

1. Correctness of the implementation, consistent with the intended functionality and without unintended edge cases.
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.

3 System Overview

Coinbax Escrow lets a sender transfer stablecoins to a receiver with a time delay. The contract holds the funds until a set release time, then releases them to the receiver. The delay gives time to cancel the transfer, or for an off-chain compliance check to reject it, before it settles. An optional fee is split between Coinbax (the platform) and the workspace that operates the escrow.

The system has two contracts:

- `EscrowFactory` stores each workspace's fee configuration and roles, and deploys the escrows.
- `WorkspaceEscrow` is the escrow itself. It custodies the funds and runs each escrow from start to finish.

The code is Solidity 0.8.34 and built on OpenZeppelin 5.4.0. It targets Base, and requires a chain with EIP-1153 transient storage (Cancun or later), which the reentrancy guard relies on.

Deployment

Every workspace gets its own `WorkspaceEscrow`. The factory deploys it as a small EIP-1167 clone of one shared implementation, which keeps each deployment cheap.

- The clone's address is derived from `keccak256(workspaceId, templateId, version)`. Each combination can be deployed only once.
- A clone has no constructor, so its configuration is set once, in `initialize()`.
- The contracts are not upgradeable. A clone is bound to its implementation permanently.

Funds and fees

One instance holds all of its escrows in a single balance per token. The contract must always hold enough to cover every active escrow. This is enforced per token.

An escrow charges two fees, one for the platform and one for the workspace. Either can be zero.

- Each fee has a rate (up to 10%), a minimum and maximum in USD (up to $1M), and a recipient.
- The fee is read from the factory when the escrow is created, then pinned onto the escrow. Later changes on the factory do not affect escrows that already exist.
- Fees are charged on top of the principal. The sender pays the principal plus both fees.
- If a fee's recipient is the zero address, that fee is refunded to the sender instead of collected.
- The sender sets a ceiling on the fee they accept. If the computed fee is higher, the creation reverts.

An escrow can be created in three ways:

- `createEscrow` : one escrow, using a standard token approval.
- `batchCreateEscrow` : up to 50 escrows in a single call.
- `createEscrowWithAuthorization` : one escrow using an EIP-3009 signature, so no separate approval is needed.

Tokens must be standard ERC-20s. No fee-on-transfer, no rebasing, and 6 to 30 decimals.

Control gating

Some escrows require off-chain checks to pass before creation or release. A template lists the checks it requires as a bitmap. The checks run off-chain. Only the pass or fail result is written on-chain, by the attestor.

There are four phases:

Phase	Bit	Gates	Keyed by
`PRE_ESCROW`	`1<<0`	create	`(sender, transactionId)`
`POST_ESCROW`	`1<<1`	release	`escrowId`
`PRE_RELEASE`	`1<<2`	release	`escrowId`
`POST_RELEASE`	`1<<3`	nothing (audit)	`escrowId`

Lifecycle

An escrow is `ACTIVE` when created. It then ends in exactly one of three ways, and it can only end once.

- **Release**: the receiver receives the principal, and both fees are collected. The sender can release at any time. Anyone can release once `releaseTime` has passed. All required release checks must have passed first.
- **Cancel**: the sender can cancel before `releaseTime` and receive the principal back. As of PR #115, both fees are still collected. The fee is treated as earned once the service has run.
- **Reject**: if a required check fails, the escrow is rejected. `withdrawRejected` returns the principal to the sender and, again, collects both fees.
- **Reclaim**: if a required check is never resolved, the sender can reclaim 30 days after `releaseTime` . This is the only path that also refunds the fees.

If a payout cannot be delivered, for example because the token has blacklisted the recipient, it is stored as a claimable credit. The recipient claims it later with `claimPayout` .

4 Security Specification

This section describes the expected behavior of the system from a security perspective. It lists the security properties the review team validated. It is not a substitute for documentation.

4.1 Actors

- **Administrator** (`DEFAULT_ADMIN_ROLE` , factory): a cold address, such as a multisig. It manages the token whitelist, default durations, workspace deregistration, escrow deprecation, implementation rotation, pause, and roles. It cannot touch the funds or records of a deployed escrow.
- **Mirror signer** (`MIRROR_SIGNER_ROLE` , factory): a hot key. It registers workspaces, writes fee configuration and recipients, and deploys escrows. It picks each escrow's owner and attestor. It cannot change an existing escrow's pinned terms or move funds.
- **Instance owner** (per escrow): sets the local token whitelist and pause, and rotates the attestor. It cannot move funds, and ownership cannot be renounced.
- **Attestor** (per workspace): records the pass or fail result of each check. It never moves funds. Release always pays the pinned receiver, and a rejection always refunds the sender.
- **Sender**: funds the escrow. It can release early, cancel before `releaseTime` , or reclaim after the grace period.
- **Receiver**: receives the principal on release. Fixed at creation.
- **Any caller**: can trigger `releaseAfterDelay` after `releaseTime` , `withdrawRejected` (which pays the original sender), or `claimPayout` for its own credit.

4.2 Trust Model

The model is "trusted attestor, trustless enforcement". Funds cannot move without the required approvals, and a failed check refunds the sender. The four privileged roles are trusted to different degrees. Some of these bounds are weaker in the code than the documentation states.

- **Administrator**: fully trusted, cold. It can change the implementation for future escrows or grant roles. It cannot reach the

caps, the pinning at create time, and the version counter. Two powers go further. It picks the owner and attestor at deploy time and sets the workspace's default escrow, so a compromised key can deploy a fully controlled escrow that off-chain routing may send users to. It can also lock a workspace's fee configuration in a way that rotating the key does not fix.

- **Instance owner**: semi-trusted, per escrow. A compromise cannot be recovered, because no factory or admin function overrides a live escrow's owner. It is trusted not to whitelist a fee-on-transfer or rebasing token, which would break that instance's per-token solvency for its senders. It is also trusted not to make itself the attestor.
- **Attestor**: semi-trusted, hot, and assumed compromisable. It cannot send funds to itself. Its main risk is that passing the release checks on a receiver that cannot receive the token can lock the escrow permanently.
- The roles are not independent. The mirror signer chooses the owner and attestor, and the owner can become the attestor. A compromised mirror signer or owner is therefore more powerful than the single-role view suggests.
- **Off-chain backend**: trusted to give each escrow a unique, unpredictable transaction ID. This is the only thing that stops front-running. It is also trusted to screen the escrow that is actually created.
- **Whitelisted tokens**: trusted to be standard ERC-20s with no fee-on-transfer, no rebasing, and 6 to 30 decimals. The only on-chain check is the decimal range at whitelist time. The fee min and max are set in USD but scaled to token units by decimals only, never by price. So the token is also assumed to trade at about $1. A depeg mis-scales the fee floor and cap, but only the fee, never the principal.
- **No fund rescue**: no role, not even the administrator, can move or seize escrowed funds. This is deliberate. The trade-off is that stuck funds cannot be rescued either. If a payout target becomes permanently unable to receive the token, its funds stay as a claimable credit forever, and no one can redirect them.

4.3 Security Properties

The properties below are guaranteed by the code. Each is listed because it is easy to assume otherwise.

- **Pinned per escrow.** The factory's fee configuration can change at any time, but each escrow records its own fee amounts, recipients, and config version when it is created, and every later payout uses those stored values. A change on the factory applies only to escrows created afterward. It cannot re-price or re-route an escrow that already exists.
- **Workspace isolation.** Each workspace has its own escrow contract, and each one holds its own funds. One workspace cannot touch another's funds. Inside a single instance, funds are tracked separately per token, and the balance always covers every active escrow in that token. A problem in one workspace, or in one token, stays contained to it.
- **A compromised admin cannot reach existing escrows.** The administrator has broad control over the factory, but it has no way into an escrow that already exists. It cannot move, redirect, or freeze funds that are already escrowed. The most it can do is change what future escrows look like, for example by pointing new deployments at a different implementation. Escrows that already hold funds are untouched.
- **The sender can cancel until `releaseTime`, even after the checks pass.** When the checks pass, the escrow is allowed to release. That does not stop the sender. The sender can still cancel and get the funds back any time before `releaseTime`, even if every check already passed. Cancel only closes at `releaseTime`. So a passed check does not mean the receiver will be paid.
- **Funds can only be stuck for a bounded time.** The one way an escrow can get stuck is a gated escrow whose attestor never records a result: it cannot release (no approval) and cannot be cancelled after `releaseTime`. But 30 days after `releaseTime` the sender can reclaim it and get everything back. The worst-case lock is the escrow's own duration plus 30 days, and it always ends in the sender's favor.
- **Clones are fixed once deployed.** Each escrow instance is a clone created for one `(workspace, template, version)` combination, and that combination can be deployed only once. A clone is not upgradeable and stays tied to the implementation it was created from. The admin can point new clones at a new implementation, but existing clones never change, and a clone's address is known in advance.

4.4 Dependencies

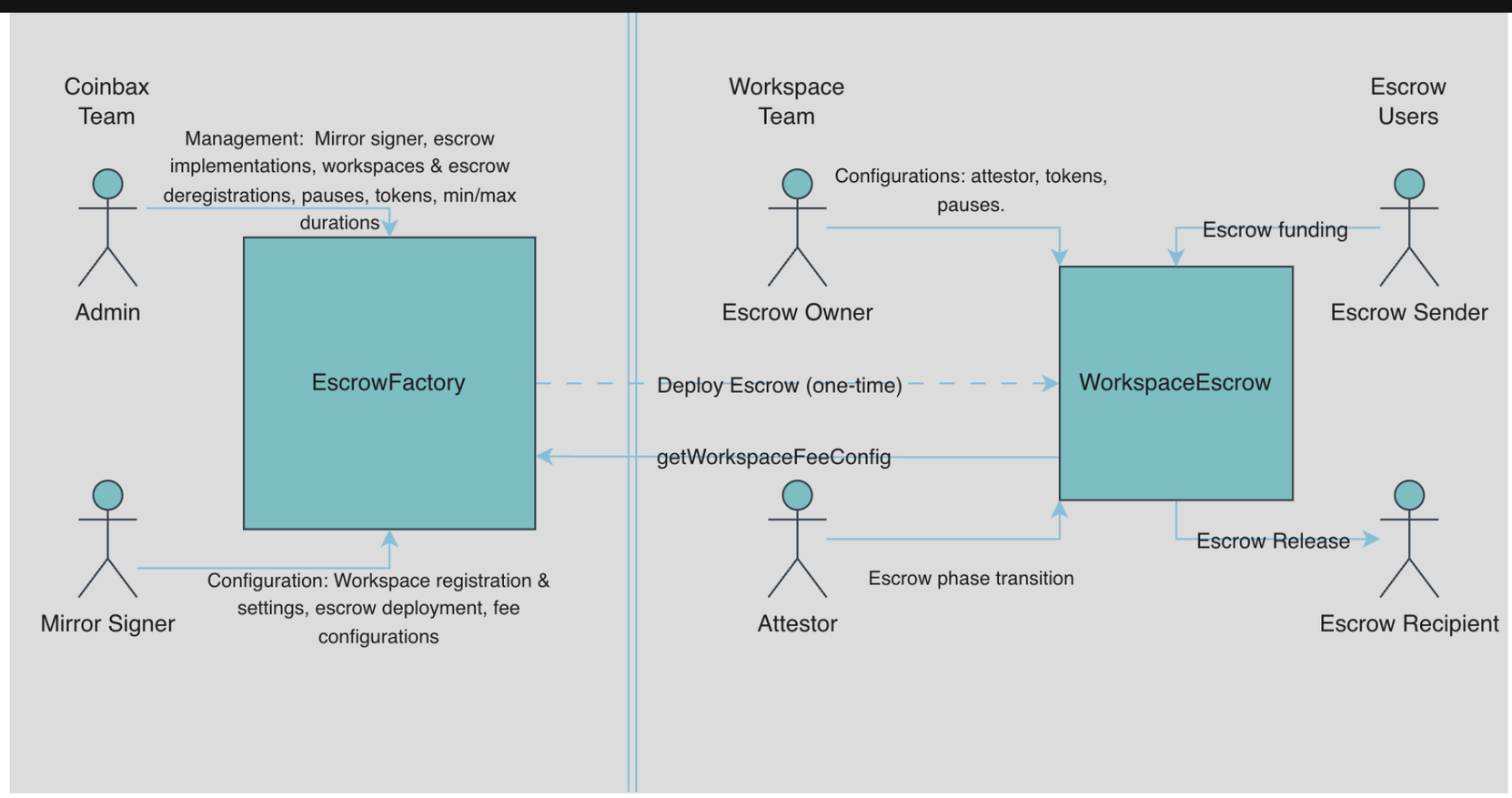
- OpenZeppelin 5.4.0: `AccessControl`, `Ownable2StepUpgradeable`, `PausableUpgradeable`, `Initializable`, `ReentrancyGuardTransient`, `SafeERC20`, and `Clones`.
- `ReentrancyGuardTransient` uses EIP-1153 transient storage, so the chain must be Cancun or later.
- `createEscrowWithAuthorization` needs the token to implement EIP-3009 `receiveWithAuthorization`.

4.5 Fund flow control

An important consideration in escrow systems is which entities control the flow of funds. In this case, the Coinbax team supports escrow operations through its factory contract and other supporting products but does not control funds in transit.

We reviewed the relevant behavior in the codebase and confirmed that, due to the separation between the administrative entities governing `EscrowFactory` and the individually deployed `WorkspaceEscrow` contracts, the flow of funds is controlled exclusively by the privileged actors defined in each escrow contract and its configuration.

This structure is illustrated in the diagram below:



5 Findings

Each issue has an assigned severity:

- Critical issues are directly exploitable security vulnerabilities that need to be fixed.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Issues without a severity are general recommendations or optional improvements. They are not related to security or correctness and may be addressed at the discretion of the maintainer.

5.1 Inaccurate Natspec for `deregisterWorkspace` Minor ✓ Fixed

Resolution

Fixed in PR-120.

Description

The `deregisterWorkspace()` function is used by the admins of the system to remove configurations and active registrations of any one workspace from the `EscrowFactory` contract. It should allow all the escrows to proceed as they were as they are already deployed. In particular, the natspec says:

ethereum/contracts/EscrowFactory.sol:L462-L467

```
/// @inheritdoc IEscrowFactory
/// @dev Wipes registration metadata. Existing escrow contracts deployed
///      for this workspace keep operating from their own state; this
///      only stops new mirror writes from landing and clears the
///      canonical-active pointer.
function deregisterWorkspace(bytes32 workspaceId) external override onlyRole(DEFAULT_ADMIN_ROLE) {
```

This isn't strictly correct. The natspec implies that `WorkspaceEscrow` escrow contracts that are deployed prior to deregistration will behave entirely as they were. However, upon trying to create new escrows - or actually in any function that calls the internal `_computeFees()` - the contract calls will always revert due to the `getWorkspaceFeeConfig()` call:

ethereum/contracts/WorkspaceEscrow.sol:L1008-L1014

```
function _computeFees(
    address token,
    uint256 amount,
    uint256 maxAcceptablePlatformFee,
    uint256 maxAcceptableWorkspaceFee
) internal view returns (ComputedFees memory result) {
    IEscrowFactory.WorkspaceFeeConfig memory cfg = factory.getWorkspaceFeeConfig(workspaceId);
```

ethereum/contracts/EscrowFactory.sol:L547-L550

```
function getWorkspaceFeeConfig(bytes32 workspaceId) external view override returns (WorkspaceFeeConfig memory) {
    if (!_workspaces[workspaceId].isRegistered) revert EscrowFactory__WorkspaceNotRegistered();
    return _workspaceFeeConfig[workspaceId];
}
```

always produce a revert. At the same time, all previously created escrows within the `WorkspaceEscrow` contract will indeed go on as they were; they do no other calls to the `EscrowFactory` contract.

Recommendation

Adjust natpsec or deregistration functionality so they are in sync.

5.2 An Escrow Can Look Gated but Enforce No Release Gate Minor ✓ Fixed

Resolution
Fixed in PR-121.

Description

`requiredPhases` is a `uint8` bitmap set once when the escrow is deployed. Only four bits are defined: PRE_ESCROW (bit 0), POST_ESCROW (bit 1), PRE_RELEASE (bit 2), and POST_RELEASE (bit 3). Release is gated only by POST_ESCROW and PRE_RELEASE. PRE_ESCROW gates creation, and POST_RELEASE gates nothing.

The only check at deploy is that a non-zero `requiredPhases` is paired with a non-zero attestor. The value of the bitmap itself is never validated. So an escrow can be deployed with `requiredPhases` set to a value that gates nothing, for example POST_RELEASE only (`0x08`), or any bit above bit 3. Such an escrow has a non-zero `requiredPhases` and a real attestor, so from the outside it looks gated. But `_release` only checks POST_ESCROW and PRE_RELEASE, neither of which is set, so no gate is enforced. Anyone can release it after `releaseTime` with no attestation.

This mainly affects off-chain systems. There is no view that reports whether an escrow's release is gated; the only signals are the raw `requiredPhases` bitmap and the presence of an `attestor`. A system that treats a non-zero `requiredPhases`, or a set attestor, as "this escrow is gated" will assume the escrow is protected when it is not.

Examples

The only validation at deploy is that a non-zero `requiredPhases` has an attestor; the bitmap value is not checked:

ethereum/contracts/EscrowFactory.sol:L317

```
if (requiredPhases != 0 && attestor == address(0)) revert EscrowFactory__AttestorNotSet();
```

`_release` gates only on POST_ESCROW and PRE_RELEASE, so any other bit enforces nothing:

ethereum/contracts/WorkspaceEscrow.sol:L1148-L1157

```
if (
    _isPhaseRequired(Phase.POST_ESCROW) && phaseResult[escrowId][uint8(Phase.POST_ESCROW)] != PhaseResult.Passed
) {
    revert WorkspaceEscrow__ReleaseControlNotPassed(Phase.POST_ESCROW);
}
if (
    _isPhaseRequired(Phase.PRE_RELEASE) && phaseResult[escrowId][uint8(Phase.PRE_RELEASE)] != PhaseResult.Passed
) {
    revert WorkspaceEscrow__ReleaseControlNotPassed(Phase.PRE_RELEASE);
}
```

Recommendation

Validate `requiredPhases` at deploy: reject any bit outside the defined range (require `requiredPhases <= 0x0F`), and if a gated template is intended, require at least one release-gating bit (POST_ESCROW or PRE_RELEASE). If PRE_ESCROW-only or POST_RELEASE-only are meant to be valid configurations, document that they do not gate release, and consider adding a view that reports the real release-gating state so off-chain systems do not have to interpret the bitmap themselves.

5.3 `EscrowFactory` Constructor Does Not Enforce Distinctness Between `initialAdmin` and `initialMirrorSigner` Minor ✓ Fixed

Resolution
Fixed in PR-121.

Description

The `EscrowFactory` constructor grants `DEFAULT_ADMIN_ROLE` and `MIRROR_SIGNER_ROLE` without checking that `initialAdmin` and `initialMirrorSigner` are different addresses.

The mirror signer runs frequent, routine operations (workspace registration, fee configuration, escrow deployment), so it operates as a hot key. The admin is meant to be a cold multisig. If a single address holds both roles, the admin key must sign the mirror signer's frequent transactions, which forces the cold admin to become a hot wallet and exposes it to the same risk of compromise.

This is a deployment-time configuration risk, not a runtime-exploitable bug.

Recommendation

factory is used in production.

5.4 Lack of Existing Escrow Deposit Pause Functionality by the Coinbax Team Minor ✓ Fixed

Resolution
Fixed in PR-118.

Description

Both `EscrowFactory` and `WorkspaceEscrow` include pause mechanisms by their respective admin teams intended for exceptional situations, such as a discovered bug, an incorrect configuration, or a compromised downstream token.

The Coinbax team that will be handling the `EscrowFactory` administration intentionally does not want their pausing abilities to interrupt an already-established flow of funds. If escrows are already created, the fund flow should only be paused by the escrow's administrators.

Indeed, in both cases we can observe the `whenNotPaused` modifiers on deploying new escrow contracts or creating new escrows, for example:

ethereum/contracts/EscrowFactory.sol:L303-L310

```
function deployDedicatedEscrow(
    bytes32 workspaceId,
    bytes32 templateId,
    uint256 version,
    uint8 requiredPhases,
    address instanceOwner,
    address attestor
) external override onlyRole(MIRROR_SIGNER_ROLE) whenNotPaused nonReentrant returns (address escrowAddress) {
```

ethereum/contracts/WorkspaceEscrow.sol:L541-L546

```
function createEscrow(address token, CreateLeg calldata leg)
    external
    whenNotPaused
    nonReentrant
    returns (bytes32 escrowId)
{
```

And the pausing itself can only be done by the admins of the respective contracts:

ethereum/contracts/EscrowFactory.sol:L535-L537

```
function pause() external override onlyRole(DEFAULT_ADMIN_ROLE) {
    _pause();
}
```

ethereum/contracts/WorkspaceEscrow.sol:L954-L956

```
function pause() external onlyOwner {
    _pause();
}
```

However, as the experts and service providers of this protocol, the Coinbax team may be the more informed team on when to execute this pause functionality to safeguard people's funds. While pausing existing fund flows may indeed be inappropriate, it may be valid for the Coinbax team to pause creation of new escrows, i.e. prevent new deposits from being created in the event of an issue. Indeed, even today the Coinbax team has this functionality by deregistering workspaces, which would prevent new escrows from being created, as discussed in issue 5.1 in this report.

A more explicit safeguarding `pauseEscrows` function may be an effective solution for the Coinbax team to further secure workspace escrows without getting in the middle of the fund flows.

Recommendation

Consider having a pause functionality executable by the Coinbax team to pause new deposits on existing escrows.

5.5 Payout Recovery Mechanism Consistency and Improvements Minor ✓ Fixed

Resolution
Fixed in PR-116.

Description

Release-side transfers use a try-transfer pattern: if the transfer fails, the amount is credited to the recipient for later withdrawal:

ethereum/contracts/WorkspaceEscrow.sol:L1200-L1206

```
    pushed = token.trySafeTransfer(to, amount);
    if (!pushed) {
        pendingWithdrawals[to][address(token)] += amount;
        emit PayoutDeferred(escrowId, to, address(token), amount);
    }
}
```

Refund paths - `cancelEscrow`, `withdrawRejected`, and `reclaimStuckEscrow` instead use a direct `safeTransfer` to the original sender:

ethereum/contracts/WorkspaceEscrow.sol:L695-L710

```
function cancelEscrow(bytes32 escrowId) external nonReentrant {
    Escrow storage escrow = escrows[escrowId];
    if (!escrow.isActive) revert WorkspaceEscrow__EscrowNotActive();

    address sender = escrow.sender;
    if (msg.sender != sender) revert WorkspaceEscrow__OnlySenderCanCancel();
    if (block.timestamp >= escrow.releaseTime) revert WorkspaceEscrow__CannotCancelAfterReleaseTime();

    escrow.isActive = false;
    escrow.isRefunded = true;
    activeEscrowCount--;

    uint256 refundAmount = escrow.amount + escrow.platformFee + escrow.workspaceFee;
    IERC20(escrow.token).safeTransfer(sender, refundAmount);

    emit EscrowRefunded(escrowId, sender, refundAmount);
```

ethereum/contracts/WorkspaceEscrow.sol:L742-L745

```
uint256 refundAmount = escrow.amount + escrow.platformFee + escrow.workspaceFee;
IERC20(escrow.token).safeTransfer(sender, refundAmount);

emit EscrowReclaimed(escrowId, sender, refundAmount);
```

ethereum/contracts/WorkspaceEscrow.sol:L906-L909

```
uint256 refundAmount = escrow.amount + escrow.platformFee + escrow.workspaceFee;
IERC20(escrow.token).safeTransfer(sender, refundAmount);

emit EscrowRefunded(escrowId, sender, refundAmount);
```

The refund behavior is safe because a failed transfer reverts the entire transaction without changing state or losing funds. However, it creates a liveness and UX asymmetry: the sender must retry the full refund operation once they can receive the tokens. Using the same transfer-or-credit mechanism as the release path would provide consistent behavior and avoid repeated failed transactions.

A second, separate improvement could be made for both recipients and senders credited through this mechanism. `claimPayout` is currently callable only by the credited account:

ethereum/contracts/WorkspaceEscrow.sol:L922-L924

```
function claimPayout(address token) external nonReentrant returns (uint256 amount) {
    amount = pendingWithdrawals[msg.sender][token];
    if (amount == 0) revert WorkspaceEscrow__NothingToClaim();
```

Smart-contract accounts, gasless accounts, or operationally slow entities such as multisig-controlled organizations may be able to receive tokens but unable to promptly submit the claim transaction themselves.

Allowing other parties to submit credit requests on their behalf would strongly improve the UX of such entities.

Recommendation

1. Consider changing `claimPayout` to accept an account parameter and allow anyone to trigger the claim
2. Extend the try-transfer and credit fallback to the refund paths for consistency with release-side payouts.

5.6 Escrows Can Be Deployed for Fee-Unconfigured Workspaces, Contradicting the Natspec Minor

✓ Fixed

Resolution
Fixed in PR-111.

Description

A workspace can be registered and have escrows created against it before a fee configuration has been set for that workspace. The natspec states this should not be possible (no escrows before fees are set):

ethereum/contracts/EscrowFactory.sol:L236-L242

```
 * @inheritdoc IEscrowFactory
 * @dev Registration sets only duration bounds. Fee config is a separate
 *      write via `setWorkspaceFeeConfig`. A workspace can exist without
 *      a fee config (it just can't have escrows created against it yet).
 */
function registerWorkspace(bytes32 workspaceId, uint256 minDuration, uint256 maxDuration)
```

but nothing in `createEscrow` (or the registration flow) enforced it - escrows deployed in this state simply carry zero fees. There is no check for fee configurations:

ethereum/contracts/EscrowFactory.sol:L303-L344

```
function deployDedicatedEscrow(
    bytes32 workspaceId,
    bytes32 templateId,
    uint256 version,
    uint8 requiredPhases,
    address instanceOwner,
    address attestor
) external override onlyRole(MIRROR_SIGNER_ROLE) whenNotPaused nonReentrant returns (address escrowAddress) {
    WorkspaceMeta storage ws = _workspaces[workspaceId];
    if (!ws.isRegistered) revert EscrowFactory__WorkspaceNotRegistered();
    if (instanceOwner == address(0)) revert EscrowFactory__InvalidInstanceOwner();
    // A gated escrow needs an attestor (the workspace's Utila wallet); the
    // clone's initialize would revert anyway, but fail early with a clear
    // factory error.
    if (requiredPhases != 0 && attestor == address(0)) revert EscrowFactory__AttestorNotSet();

    // One escrow per (workspace, template, version). The deterministic salt
    // makes the address predictable off-chain and the mapping rejects a
    // duplicate before the CREATE2 would.
    bytes32 salt = _escrowSalt(workspaceId, templateId, version);
    if (escrowOfSalt[salt] != address(0)) revert EscrowFactory__DedicatedEscrowAlreadyExists();

    // NOTE (static-analysis): Slither flags `reentrancy-no-eth` /
    // `reentrancy-benign` here because the `initialize` external call
    // precedes the `escrowOfSalt` / `dedicatedEscrow` / `_deployments`
    // writes below. Reviewed and safe: this function is `nonReentrant`, and
    // the callee is a *freshly* `Clones.clone`d instance of the
    // admin-set-only `workspaceEscrowImplementation` — not attacker-supplied
    // code — whose `initialize` runs once and makes no callback into the
    // factory. Deploy + initialize are one atomic tx (no front-run window).
    escrowAddress = Clones.cloneDeterministic(workspaceEscrowImplementation, salt);
    WorkspaceEscrow(escrowAddress)
        .initialize(
            workspaceId,
            ws.minEscrowDuration,
            ws.maxEscrowDuration,
            address(this),
            _tokenList,
            instanceOwner,
            requiredPhases,
            attestor
        );
```

Funds are not at risk, but the invariant the docs promise is violated.

Recommendation

Adjust the natspec/documentation or introduce checks for the existence of fee configurations.

5.7 Misleading `dedicatedEscrow` / `getEscrowForWorkspace` Pointer ✓ Fixed

Resolution
Fixed in PR-119.

Description

A workspace does not have a single “dedicated” escrow. `deployDedicatedEscrow` creates one escrow per `(workspaceId, templateId, version)` triple, and each deploy overwrites the workspace’s `dedicatedEscrow` field with the newest one. `getEscrowForWorkspace` returns that single field, so a workspace can have many live escrows at once while the view surfaces only the last one deployed. Deploying a new template or version silently repoints it, and the older escrows stay active but are no longer returned.

Deprecating the current escrow clears the pointer to the zero address, even when other escrows for the workspace are still active. After deprecating the most-recent escrow, `getEscrowForWorkspace` returns the zero address, as if the workspace had no escrow at all, while its older `(templateId, version)` escrows are still live.

The field is never read by on-chain logic; it is only an off-chain convenience. `getEscrowForTemplate(workspaceId, templateId, version)` and `getWorkspaceDeployments(workspaceId)` already provide precise and complete lookup. The naming (`dedicatedEscrow` , `getEscrowForWorkspace`) suggests one escrow per workspace, which does not match the per-triple model.

Examples

`deployDedicatedEscrow` overwrites the pointer with the most recent escrow:

ethereum/contracts/EscrowFactory.sol:L346-L347

```
ws.dedicatedEscrow = escrowAddress; // most-recent; precise lookup via getEscrowForTemplate
```

`getEscrowForWorkspace` returns that single field:

ethereum/contracts/EscrowFactory.sol:L573-L575

```
function getEscrowForWorkspace(bytes32 workspaceId) external view override returns (address escrowAddress) {
    return _workspaces[workspaceId].dedicatedEscrow;
}
```

`deprecateEscrow` clears it to the zero address even when other escrows for the workspace are still active:

ethereum/contracts/EscrowFactory.sol:L497-L500

```
if (ws.dedicatedEscrow == escrowAddress) {
    ws.dedicatedEscrow = address(0);
    ws.lastUpdatedAt = block.timestamp;
}
```

Recommendation

The field is unused on-chain, and `getEscrowForTemplate` and `getWorkspaceDeployments` already cover lookup. Either rename it for clarity (for example `latestEscrow` / `getLatestEscrowForWorkspace`) so callers do not read it as "the workspace's escrow", or remove the field and view entirely. In both cases, correct the NatSpec.

5.8 Structs and Events for `WorkspaceEscrow` Can Be Isolated to an Interface File Acknowledged

Resolution
Acknowledged by the team but chosen not to be pursued at this time due to lack of security impact.

Description

`WorkspaceEscrow` declares its structs and events directly in the implementation contract rather than in a dedicated interface, unlike `EscrowFactory`, which already separates its interface. Without this separation, the file can become unnecessarily large which could hinder its maintainability.

For example, the following can be isolated to an interface:

Structs:

ethereum/contracts/WorkspaceEscrow.sol:L172-L195

```
/// @notice EIP-3009 authorization fields the user signs off-chain.
///         The token contract is the authority on signature validity;
///         this contract only threads the values through to
///         `receiveWithAuthorization`.
struct Authorization {
    uint256 validAfter;
    uint256 validBefore;
    bytes32 nonce;
    uint8 v;
    bytes32 r;
    bytes32 s;
}

/// @dev Internal struct returned from `_computeFees`. Bundles the
///      computed amounts with the recipients-and-version snapshot
///      so the calling function can pin them onto the Escrow record
///      in a single assignment.
struct ComputedFees {
    uint256 platformFee;
    uint256 workspaceFee;
    address platformFeeRecipient;
    address workspaceFeeRecipient;
    uint64 schedulerVersion;
}
```

Events:

ethereum/contracts/WorkspaceEscrow.sol:L314-L334

```
///         Off-chain reconciliation correlates the two via
///         `escrowId`. Carries the audit trail that lets a watcher
///         pin which fee config version + recipient pair the escrow
///         was created against.
event EscrowFeeSnapshot(
    bytes32 indexed escrowId,
    address platformFeeRecipient,
    address workspaceFeeRecipient,
    uint64 feeConfigVersionAtCreation
);

event EscrowReleased(bytes32 indexed escrowId, address indexed receiver, uint256 amount, bool earlyRelease);

event EscrowRefunded(bytes32 indexed escrowId, address indexed sender, uint256 totalRefunded);

/// @notice Emitted ONLY when a fee leg pays out to a real recipient.
///         Zero-recipient fees that get refunded to the sender are
///         not "collected revenue" and don't emit this — they appear
///         as part of the principal-side movement instead.
event FeeCollected(bytes32 indexed escrowId, address indexed recipient, uint256 amount, FeeKind kind);
```

Recommendation

Consider creating an interface file for `WorkspaceEscrow` contracts for the sake of code organization.

Appendix 1 - Disclosure

Consensys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via Consensys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any third party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any third party by virtue of publishing these Reports.

A.1.1 Purpose of Reports

The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

A.1.2 Links to Other Web Sites from This Web Site

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Consensys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that Consensys and CD are not responsible for the content or operation of such Web sites, and that Consensys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that Consensys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. Consensys and CD assumes no responsibility for the use of third-party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

A.1.3 Timeliness of Content

The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice unless indicated otherwise, by Consensys and CD.