AtumLabs: EVM - TVM update

Security Review

Cantina Managed review by:
Alireza Arjmand, Lead Security Researcher
Jonatas Martins, Security Researcher

April 14, 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
- 2.2 Codebase context . . . 3
- 2.3 Trust assumptions . . . 6

3 Findings **9**
- 3.1 Low Risk . . . 9
  - 3.1.1 Fee can cause zero-amount transfers and potentially blocking release . . . 9
  - 3.1.2 Token decimals misconfiguration can lead to incorrect accounting . . . 11
  - 3.1.3 Unbounded Gas Costs Can Make Settlement Actions Economically Unsafe . . . 11
- 3.2 Informational . . . 12
  - 3.2.1 `EIP-7702` delegated EOAs are incompatible with `Permit2` and `SignatureChecker.isValidSignatureNow` . . . 12
  - 3.2.2 Inaccurate Comment Regarding Admin Delay and Role Grants . . . 12
  - 3.2.3 Suboptimal Initial Admin Delay Configuration . . . 13
  - 3.2.4 Improve Readability of Fee Representability Checks . . . 13
  - 3.2.5 Missing Target Chain and Target Recipient in Deposit On Chain Footprint . . . 13
  - 3.2.6 Unrestricted Reserver and Releaser Selection . . . 14

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

From Feb 9th to Feb 12th the Cantina team conducted a review of evm-contracts and tvm-contracts on commit hashes eea3ecb0 and 689387e1 respectively. The team identified a total of **10** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	3	2	1
Gas Optimizations	0	0	0
Informational	7	1	6
Total	**10**	**3**	**7**

2.1 Scope

The security review had the following components in scope for evm-contracts and tvm-contracts on commit hashes eea3ecb0 and 689387e1 respectively:

```
├── evm-contracts
│   └── src
│       ├── Escrow.sol
│       ├── FulfillmentProxy.sol
│       └── interfaces
│           ├── IEscrow.sol
│           ├── IFulfillmentProxy.sol
│           └── ITokenConfig.sol
└── tvm-contracts
    └── contracts
        ├── Escrow.sol
        ├── FulfillmentProxy.sol
        ├── ITIP712.sol
        ├── Migrations.sol
        ├── TIP712.sol
        ├── interfaces
        │   ├── IEscrow.sol
        │   ├── IFulfillmentProxy.sol
        │   └── ITokenConfig.sol
        └── test
            ├── MockERC20.sol
            └── Permit2Extension.sol
```

2.2 Codebase context

Protocol and Code Overview: Atum is a cross chain stablecoin payments protocol built around an off chain RFQ flow. A Depositor signs a Permit2 authorization (and related witness data) off chain. A Reserver runs an off-chain auction, selects a Settler and signs a reservation attesting to the chosen Settler for a specific `paymentId`. The Settler fulfills the payment on the destination chain, after which a Releaser authorizes the final outcome on the source chain by signing either a release (pay Settler, keep fee) or a refund (return funds to Depositor). The smart contracts primarily enforce correct custody and one time settlement on the source chain, while correctness of the cross chain fulfillment is enforced by off chain validation and the Releaser's decision making.

On chain, the system is composed of two contracts:

- **Escrow (source chain custody and settlement)**. Escrow receives funds via Permit2 `permitWitnessTransferFrom`, stores a compact commitment to the deposit (a keccak hash of the

`ActiveDeposit` fields), and later allows a single terminal action: `release` or `refund`. Both terminal actions delete the stored deposit hash before transferring tokens, providing replay protection and reducing reentrancy risk. Release and refund require a valid EIP 712 signature from the Releaser over all relevant witness fields. Escrow also manages an allowlist of supported tokens and per token fee configuration, and accrues collected fees for withdrawal by a treasurer role.

- **FulfillmentProxy (destination chain payment helper)**. FulfillmentProxy simply transfers tokens from the caller (typically the Settler) to the intended recipient and emits an event that off chain actors can observe. It also offers `strictFulfill`, which adds a deadline check and a per `(paymentId, sender)` replay guard for higher risk flows. The escrow contract does not read FulfillmentProxy state; it is an off chain signal and convenience layer.

The Tron and EVM variants are largely identical at the application layer, with these relevant differences:

- Tron code uses a TIP 712 domain separator pattern (masked chain id) and includes a special case for Tron USDT behavior (transfer returns `false`) by calling `transfer` directly instead of `safeTransfer`. The EVM version uses OpenZeppelin EIP 712 (`_hashTypedDataV4`) and relies on `SafeERC20` for transfers.

Threat Model:

What the contracts guarantee: If the Reserver and Releaser behave correctly and signatures are validated correctly, Escrow enforces:

- Funds can only be pulled from a Depositor via a valid Permit2 witness transfer to the Escrow contract.
- Each deposit identified by `paymentId` can be settled at most once (release or refund) because the active deposit hash is deleted before transfer.
- Release sends `amount - feeAmount` to the Settler, and refund sends `amount` back to the Depositor.
- Token fees are applied according to the current on chain token configuration at the time the deposit executes.

FulfillmentProxy provides:

- An on chain token transfer and event emission that off chain systems can use as proof of an attempted fulfillment.
- Optional strict mode protections (deadline and duplication guard) for payments that require higher assurance.

What the contracts do not guarantee:

- Escrow does not verify that a fulfillment happened on the destination chain, to the correct recipient, for the correct amount. This is intentionally off loaded to off chain validation and to the Releaser.
- Escrow does not enforce any off chain deadlines. FulfillmentProxy deadlines are local to the destination chain helper and are not enforced by Escrow.
- There is no on chain notion of partial fulfillment. Any notion of splitting or aggregating fills would be an off chain policy implemented by the Releaser and Settlers.

Adversarial capabilities considered:

- Malicious Depositors attempting to grief or exploit off chain infrastructure.
- Malicious or careless Settlers attempting to get paid without correct fulfillment.
- Attackers attempting signature replay, ambiguity, or misuse of contract wallet signature verification.
- Token level behaviors that break assumptions: blacklist, fee on transfer, rebasing, non standard return values.
- Chain instability: reorgs, temporary forks, timestamp variability, and cross chain finality mismatch.
- Admin misuse or front running through token configuration changes.

Actors and Trust Model:

Releaser (trusted): The Releaser is the primary safety assumption. It is the only entity that can authorize terminal settlement in Escrow.

Responsibilities and assumptions:

- Verifies that the destination chain fulfillment occurred correctly and completely. It must not rely solely on an emitted event; it should validate the actual token transfer semantics on the destination chain, including token contract behavior and recipient correctness.
- Signs **exactly one** terminal action per `paymentId` (release or refund). It must never sign both for the same deposit.
- Handles chain instability. It must account for reorg and finality on both chains and only sign once the relevant condition is final enough for the risk tolerance of the system.
- For release: waits until the fulfillment transaction is finalized on the destination chain before signing, to avoid paying the Settler for a fulfillment that is later reorged out.
- For refund: waits until the protocol's off chain deadline has passed and the order is truly not settled, including handling last second fulfillment races.
- If implemented as a smart contract account, it must implement EIP 1271 (`isValidSignature`) correctly and defensively, and must ensure the signature it validates binds all fields required by the protocol, including chain specific replay protections.

Reserver (trusted): The Reserver is the gatekeeper for Settler selection and for binding the parties for a `paymentId`.

Responsibilities and assumptions:

- Selects the intended Settler for the RFQ and signs the reservation witness, binding `paymentId` and `settler`.
- Ensures the chosen Releaser is correct for the order and that off chain coordination uses the intended trusted Releaser set.
- Ensures the signature scheme is not replayable or ambiguous across chains or contracts. The EIP 712 domain separation must bind chain id and verifying contract correctly for the environment.
- If implemented as a smart contract account, it must implement EIP 1271 correctly and verify all fields it is expected to verify, matching the off chain EOA signature checks.

Settler (untrusted): The Settler provides liquidity and performs destination chain fulfillment but is not trusted for safety.

Assumptions and constraints:

- The Settler only receives funds if the Releaser signs a valid release that matches the stored deposit commitment.
- The Settler must treat on chain events as non authoritative signals. Before fulfilling, it should verify the deposit is valid and active by reconstructing expected `paymentId` and checking Escrow state (e.g., `getDepositHash`) rather than relying only on a `Deposited` event.
- The Settler must ensure the Releaser configured for the order is one it trusts. Since the Depositor can effectively choose the Reserver and can influence which Releaser is referenced off chain, Settlers should refuse to fulfill orders that route to unknown or malicious Releasers.
- The Settler must call the correct fulfillment mode on the destination chain based on risk classification:
  - `fulfill` for normal payments.
  - `strictFulfill` for higher risk payments where deadlines and duplicate protection are required.

Depositor (untrusted): Depositors are not trusted for safety and should not be able to cause unauthorized release or refund if Reserver and Releaser behave correctly.

Assumptions and constraints:

- Depositors must correctly sign Permit2 messages and must manage Permit2 approvals and nonces safely.
- Depositors should ensure the Permit2 nonce they use for Atum is not concurrently used in other applications and remains available, to avoid unintended invalidation or confusion.
- Depositors should understand that off chain deadlines are enforced by off chain actors and FulfillmentProxy strict mode, not by Escrow itself.

System Level Assumptions and Operational Requirements:

1. **Releaser correctness is the primary safety assumption**. If the Releaser mishandles reorgs, finality, signature validation, or fulfillment verification, deposits can be incorrectly released, incorrectly refunded, or stuck.
2. **Reserver authorization is the gatekeeper assumption**. If the Reserver signs the wrong data, selects an incorrect Settler, or signatures are replayable or ambiguous across chains or contracts, unauthorized parties could be approved and the off chain flow could be subverted.
3. **Untrusted roles should only affect liveness, not safety**. Settler and Depositor behavior should only determine whether a payment completes. They should not be able to cause loss, unauthorized release, or incorrect refunds if the Releaser and Reserver behave correctly.

2.3 Trust assumptions

Permit2 approval and nonce management:

- Depositors must have granted the Permit2 contract sufficient allowance for their intended payment amounts. If approval is insufficient, deposits will fail and funds will not move, affecting liveness.
- Depositors should ensure the Permit2 nonce used in the Atum permit is not already consumed by other applications and will not be consumed concurrently. `paymentId` is derived from the permit nonce and the deposit signature, so nonce reuse or racing can cause deposits to fail or become unrecoverable operationally.

Reserver and Releaser account type and signature validation:

- The system assumes Reserver and Releaser are EOAs for the simplest and safest signature verification path.
- If either is a contract account, they must implement EIP 1271 correctly and must ensure their signature validation logic binds and checks all fields that an EOA signature would bind, including the full EIP 712 domain separation. In practice, this means ensuring the signed digest includes the correct chain id and verifying contract address, and that there is no cross chain replay surface.
- For Tron deployments, cross chain replay resistance must be carefully reviewed given the TIP 712 chain id masking behavior. Any off chain signer or contract wallet must match the same domain separator construction used on chain.

Destination chain fulfillment amount:

- Settlers must fulfill on the destination chain for **`amount - feeAmount`**, not the full deposit amount. The fee is retained in Escrow and accrued to `s_tokenFeeRevenue`. Fulfilling the full amount makes the Settler economically whole only if it separately recovers the fee, and can create reconciliation issues or economic loss.
- Releasers must verify that the fulfilled amount matches the expected net amount and not just that a fulfillment occurred.

Deadlines and timing:

- The Escrow contract does not enforce the RFQ or fulfillment deadline. Deadlines are an off chain control plane measure and, optionally, a destination chain enforcement through `strictFulfill`.
- Fillers and off chain matching must compare the intended deadline against the relevant chain timestamps. Because FulfillmentProxy uses `block.timestamp`, Settlers must ensure they are fulfilling within the deadline according to the destination chain timestamp, and must account for timestamp differences between chains.
- The Releaser must handle last second fulfillment races: it should avoid signing a refund immediately at the deadline boundary without confirming that no valid fulfillment transaction has landed and finalized shortly before or at the deadline.

Finality and reorg resistance:

- Release signing should occur only after the destination chain fulfillment is finalized to an acceptable depth, otherwise the system can pay out escrowed funds for a fulfillment that is later reorged out.

- Refund signing should occur only after the Releaser is confident the payment will not settle, which includes waiting beyond the deadline and checking for late inclusion and finality.
- The Releaser must not sign both release and refund for the same `paymentId` and must avoid signing a terminal action until finality is guaranteed at the chosen security level.

Token behavior assumptions and allowlisting policy:

- Supported tokens must be carefully allowlisted. Fee on transfer and rebasing tokens are generally incompatible with the escrow invariants because Escrow records a fixed `amount` and later transfers based on that amount. Fee on transfer tokens can cause actual received amounts to be less than recorded, breaking settlement. Rebasing tokens can change balances over time, breaking the accounting assumptions.
- If a fee on transfer token is ever allowed, protocol fees should be configured to zero and the system must account for the token's transfer tax separately. Even then, correctness remains fragile.
- Centralized stablecoins may implement blacklist or pause features. Token issuers can blacklist Escrow or participant addresses (Depositor, Settler, Reserver, Releaser), freezing funds and breaking liveness. This is an external dependency outside the protocol's control.

Token fee configuration and admin trust:

- Token fee configuration is governed by `TOKEN_CONFIG_ROLE` and is trusted not to front run users. Fee determination happens at deposit execution time, so transactions in flight can land after a fee update and pay the new fee. This can create user facing surprise and economic unfairness if the role is abused.
- Operationally, fee updates should be announced and possibly rate limited or time delayed off chain to reduce this risk, since there is no on chain timelock for fee updates in the code shown.

Off chain infrastructure and DoS resilience:

- The RFQ and matching system must be resilient to spam and invalid orders. A centralized Reserver or matching service should assume adversarial Depositors can submit many invalid signatures, invalid permits, or malformed orders to waste resources.
- A practical mitigation is a priority or reputation based queue that separates known valid users and returning users from untrusted new senders, and that performs cheap pre checks before expensive signature and chain simulation work.

Event reliance and independent verification:

- Releasers must not rely solely on FulfillmentProxy events to determine that a fulfillment occurred. They must verify the underlying token transfers and relevant call data semantics.
- Settlers must not rely solely on `Deposited` events. Before fulfilling, they should verify the deposit is active and matches expectations by reconstructing `paymentId`, checking Escrow state, and verifying that the configured trusted Releaser is acceptable.

Partial fulfillment policy:

- The on chain protocol is designed for a single settlement per `paymentId` and does not implement partial fills. Any attempt to support partial fulfillment would be an off chain convention where the Releaser aggregates multiple destination chain fills and only signs release once the net requirement is satisfied.
- If such a policy is adopted, it becomes an additional trust and correctness burden on the Releaser and off chain accounting, and should be treated as out of scope for on chain enforcement.

Tron specific operational assumptions:

- A compatible Permit2 deployment on Tron must exist and be correct, since Escrow relies on `permitWitnessTransferFrom`.
- The Tron implementation includes a USDT special case because USDT on Tron is known to return `false` for `transfer`. This implies USDT should be explicitly and intentionally allowlisted by admins, and the system should treat it as a special compatibility case rather than assuming all tokens behave like standard ERC 20.

- Even with the special case, Tron USDT transfers that return `false` do not provide a revert based failure signal, so operational monitoring should detect failures through balance deltas rather than relying on return values.

Admin role and governance assumptions:

- `AccessControlDefaultAdminRules` introduces an admin transfer delay mechanism. The initial delay configured at deployment should be chosen intentionally as part of the security posture, not left at an overly permissive value for production.
- If the default admin delay is increased after deployment, the change is itself delayed by the current delay rules, meaning the system must wait the full configured period for the increase to take effect. Deployment planning must account for this.
- Ownership and admin roles should not be renounced in a way that prevents emergency response. In particular, `PAUSER_ROLE`, `TOKEN_CONFIG_ROLE`, and `TREASURER_ROLE` should remain held by operationally controlled entities, ideally with appropriate multisig and key management.

3 Findings

3.1 Low Risk

3.1.1 Fee can cause zero-amount transfers and potentially blocking release

Severity: Low Risk

Context: Escrow.sol#L478

Description: The current validation uses `if (feeAmount > amount)` which allows the edge case `feeAmount == amount`. This may happen when minimum fee is the amount of tokens transfered. In that scenario, the net amount to transfer after fees becomes 0.

Some ERC20 tokens revert on `transfer(0)/transferFrom(0)`, so the "post-fee" transfer to the settler may fail.

Additionally, a zero amount transfer should be blocked by the `deposit` function.

Proof of Concept: Here is a PoC that shows zero transfer can happen. Below the amount that is being deposited is equal to `minFee`:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.27;

import {Test, console2} from "forge-std/Test.sol";
import {Escrow} from "../src/Escrow.sol";
import {IEscrow} from "../src/interfaces/IEscrow.sol";
import {ITokenConfig} from "../src/interfaces/ITokenConfig.sol";
import {IPermit2, ISignatureTransfer} from "permit2/src/interfaces/IPermit2.sol";
import {DeployCoreContracts} from "../script/deploy/DeployCoreContracts.s.sol";
import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";


// forge test --match-test test_checkMinFeeIsAmount
↪  --rpc-url=https://eth-mainnet.g.alchemy.com/v2/key -vvvv
contract EscrowPoCTest is Test {

    Escrow escrow;
    address fakeReserver = address(new saysYesContract());
    address fakeReleaser = address(new saysYesContract());
    address fakeSettler = address(new EmptyContract());
    address randomExecutor = address(0x1234567);


    address immutable permit2 = address(0x000000000022D473030F116dDEE9F6B43aC78BA3);
    ERC20 immutable USDT = ERC20(0xdAC17F958D2ee523a2206206994597C13D831ec7); //
    ↪  https://etherscan.io/token/0xdac17f958d2ee523a2206206994597c13d831ec7#code

    function setUp() public {
        escrow = new Escrow(IPermit2(permit2), address(this));
        escrow.grantRole(escrow.TOKEN_CONFIG_ROLE(), address(this));
    }

    function test_checkMinFeeIsAmount() public {
        uint256 minFeeAmount = 1e2;
        escrow.allowTokenAndSetFeeConfig(USDT, USDT.decimals(), 0, uint80(minFeeAmount),
        ↪  0);

        address Alice = address(new saysYesContract());
        deal(address(USDT), Alice, 1e10);

        console2.log(Alice);


        // Create deposit input
        ISignatureTransfer.PermitTransferFrom memory permit =
            ISignatureTransfer.PermitTransferFrom({
```

```
        permitted: ISignatureTransfer.TokenPermissions({
            token: address(USDT),
            amount: minFeeAmount
        }),
        nonce: 1,
        deadline: block.timestamp + 1 days
    });

IEscrow.DepositWitness memory depositWitness =
    IEscrow.DepositWitness({
        reserver: fakeReserver,
        releaser: fakeReleaser
    });

// dummy paymentId (real one is computed inside escrow)
bytes32 dummyPaymentId = keccak256(
    abi.encode(Alice, bytes(""), permit.nonce)
);

IEscrow.ReserveWitness memory reserveWitness =
    IEscrow.ReserveWitness({
        paymentId: dummyPaymentId,
        settler: fakeSettler
    });

IEscrow.DepositInput memory input = IEscrow.DepositInput({
    permit: permit,
    depositor: Alice,
    depositWitness: depositWitness,
    depositSignature: "",     // empty
    reserveWitness: reserveWitness,
    reserveSignature: ""      // empty
});


vm.startPrank(Alice);
MockERC20(address(USDT)).approve(address(permit2), type(uint256).max);
vm.stopPrank();

vm.startPrank(randomExecutor);
escrow.deposit(input);
vm.stopPrank();

IEscrow.ReleaseWitness memory witness =
IEscrow.ReleaseWitness({
    paymentId: dummyPaymentId,
    depositor: Alice,
    token: address(USDT),
    amount: minFeeAmount,
    feeAmount: minFeeAmount,
    reserver: fakeReserver,
    releaser: fakeReleaser,
    settler: fakeSettler
});

uint256 settlerBalanceBefore = USDT.balanceOf(fakeSettler);

vm.startPrank(randomExecutor);
escrow.release(IEscrow.ReleaseInput({
    releaseWitness: witness,
    releaseSignature: ""
}));
vm.stopPrank();

uint256 settlerBalanceAfter = USDT.balanceOf(fakeSettler);
```

```
        vm.assertEq(settlerBalanceBefore, settlerBalanceAfter);
    }
}

contract saysYesContract {
    function isValidSignature(bytes32, bytes memory) external view returns (bytes4){
        return saysYesContract.isValidSignature.selector;
    }
}

contract EmptyContract {

}

interface MockERC20 {
    function approve(address, uint256) external; // Since USDT does not return a good
    ↪  value
}
```

Recommendation: Change the check to revert when `feeAmount >= amount`. This avoids zero-amount transfer edge cases. And block zero amount deposits in `deposit`.

Atum Labs: Fixed in evm-contracts/pull/76 and tvm-contracts/pull/3.

Cantina Managed: Verified the fixes. Now if `fee - feeAmount == 0` the user can not deposit into the contract.

3.1.2 Token decimals misconfiguration can lead to incorrect accounting

Severity: Low Risk

Context: Escrow.sol#L519

Description: The contract currently relies on a configured/stored decimals value for each ERC20, rather than reading the token's actual `decimals()` on-chain. This creates a misconfiguration risk (wrong decimals set during onboarding, upgrades, or role-based configuration), which can break normalization and accounting logic that depends on decimals (e.g., amount conversions, limits, fee calculations, escrow release amounts). A wrong decimals value can lead to incorrect fee calculation and may cause funds to be over/under-payment.

Recommendation: Read decimals directly from the token using a bounded staticcall to `IERC20Metadata.decimals()`. For example:

```
function _safeDecimals(address token) internal view returns (uint8) {
    (bool success, bytes memory data) =
        token.staticcall(abi.encodeWithSelector(IERC20Metadata.decimals.selector));
    if (success && data.length >= 32) {
        uint256 value = abi.decode(data, (uint256));
        if (value <= MAX_TOKEN_DECIMALS) {
            return uint8(value);
        }
    }
    revert InvalidTokenDecimals();
}
```

Atum Labs: Fixed in evm-contracts/pull/76 and tvm-contracts/pull/3.

Cantina Managed: Verified the fixes. The contract now enforces that the `decimals()` return value of the token matches the `decimals` input parameter of `_allowToken`.

3.1.3 Unbounded Gas Costs Can Make Settlement Actions Economically Unsafe

Severity: Low Risk

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

Description: The protocol does not ensure that the gas cost incurred by callers of `deposit`, `release`, and `refund` is covered by protocol fees or otherwise compensated. This can reduce participation and harm liveness if actions become consistently unprofitable.

In particular, signature verification may involve contract accounts via EIP 1271. A contract based signer can consume an unexpectedly large amount of gas in `isValidSignature`, causing the transaction submitter (often a trusted operator) to bear outsized costs. Similar concerns apply to external calls such as the Permit2 transfer path.

Recommendation:

- Add explicit gas bounds where possible for external calls, especially around the `Permit2` transfer and signature verification paths (for example, using capped gas calls or enforcing EOA only for trusted roles if acceptable).
- Ensure actors are economically incentivized to perform required actions, either off chain or on chain. Practically, model worst case gas for `deposit`, `release`, and `refund`, and set fee parameters and fee distribution so that expected costs are covered and the party performing the action receives compensation.

Atum Labs: Acknowledged.

Cantina Managed: Acknowledged.

3.2 Informational

3.2.1 `EIP-7702` delegated EOAs are incompatible with `Permit2` and `SignatureChecker.isValidSignatureNow`

Severity: Informational

Context: Escrow.sol#L54

Description: The protocol's authorization flow relies on `Permit2` and/or OpenZeppelin's `SignatureChecker.isValidSignatureNow` for signature validation. Under `EIP-7702` (Delegation), an EOA can delegate execution to contract code, which can change how signatures are validated and how an "EOA" is treated by signature-checking logic. As a result, EOAs using `EIP-7702` delegation may fail signature verification and be unable to interact with the protocol via these paths. This is primarily a compatibility limitation that can block a subset of users (delegated EOAs) from using the protocol, even though they control the account.

Recommendation: Document this limitation explicitly (`EIP-7702` delegated EOAs not supported) and, if supporting them is a goal, add an alternative authentication path designed for delegated accounts.

Atum Labs: Acknowledged.

Cantina Managed: Acknowledged.

3.2.2 Inaccurate Comment Regarding Admin Delay and Role Grants

Severity: Informational

Context: Escrow.sol#L73-L74

Description: The constructor comment states that a low initial admin delay allows the deployer to quickly grant operational roles. However, in `AccessControlDefaultAdminRules`, the admin delay only applies to transfers of `DEFAULT_ADMIN_ROLE` and delay updates. It does not affect `grantRole` or `revokeRole`.

As long as the deployer holds `DEFAULT_ADMIN_ROLE`, role grants are immediate regardless of the configured delay. The comment is therefore inaccurate and may cause confusion about the security model.

Recommendation: Update the comment to clarify that the delay only affects `DEFAULT_ADMIN_ROLE` transfers and delay changes, and does not impact `grantRole` calls.

Atum Labs: Fixed in evm-contracts/pull/76 and tvm-contracts/pull/3.

Cantina Managed: Verified the fixes, the comments are now corrected.

3.2.3 Suboptimal Initial Admin Delay Configuration

Severity: Informational

Context: Escrow.sol#L83

Suboptimal Initial Admin Delay Configuration:

Description: The contract initializes `AccessControlDefaultAdminRules` with a delay of `1`. If the intention is to keep the system flexible at deployment and later increase the delay to a more practical value, this configuration introduces unnecessary friction.

When increasing the admin delay to a new value `X`, the change itself is subject to the delay mechanism. This means the system must wait `X` seconds before the new delay becomes effective. For example, increasing the delay to 2 days requires waiting 2 days before the new delay is active.

If ownership is intended to be transferred to a multisig, the fastest and cleanest approach is to set the multisig as the default admin at deployment and grant operational roles through the multisig directly, rather than relying on a short initial delay and subsequent transition.

Recommendation: If early flexibility is desired, consider setting the initial admin delay to `0`.

Alternatively, if a multisig will control the system, set it as `DEFAULT_ADMIN_ROLE` at deployment and configure operational roles through it directly.

Document clearly that increasing the delay later requires waiting the full new delay period before it takes effect.

Atum Labs: Acknowledged.

Cantina Managed: Acknowledged. Per `AccessControlDefaultAdminRules` delay logic, increasing the admin delay to a new value requires waiting `min(newDelay, 5 days)` before it takes effect — for example, setting `newDelay = 3 days` would only take effect after 3 days.

3.2.4 Improve Readability of Fee Representability Checks

Severity: Informational

Context: Escrow.sol#L561-L576

Description: The current representability checks for `minFee` and `maxFee` rely on scaling the value to token decimals and verifying that the result is non zero. While correct, this makes the rounding boundary implicit and slightly harder to read.

The condition can be expressed more directly:

- If `tokenDecimals >= 6`, all `minFee` and `maxFee` values are representable.
- If `tokenDecimals < 6`, then `minFee` and `maxFee` must be at least `10 ** (6 - tokenDecimals)`; otherwise they will round down to zero.

Making this explicit improves clarity and makes the rounding constraint easier to audit.

Recommendation: Refactor the checks to use a threshold based on `tokenDecimals`.

- If `tokenDecimals < 6`, compute `uint256 threshold = 10 ** (6 - tokenDecimals)` and require that any non zero `minFee` or `maxFee` is greater than or equal to this threshold.
- If `tokenDecimals >= 6`, skip the check.

Atum Labs: Acknowledged.

Cantina Managed: Acknowledged.

3.2.5 Missing Target Chain and Target Recipient in Deposit On Chain Footprint

Severity: Informational

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

Description: The Escrow `deposit` flow does not record or emit the destination chain and destination recipient that the off chain system intends to fulfill. As a result, if someone interacts with the contract

directly, observers cannot determine the intended target chain or target address from on chain state or events.

This reduces auditability and monitoring, makes incident response and dispute investigation harder, and increases reliance on off chain logs and proprietary RFQ data to reconstruct intent.

Recommendation: Include the destination chain identifier and destination recipient in the on chain deposit footprint, ideally by emitting them in deposit events. It would be beneficial to add `targetChainId` and `targetRecipient` fields to the deposit-related structs and include them in the `Deposited` and `DepositedMany` events.

Atum Labs: Acknowledged.

Cantina Managed: Acknowledged. This mechanism is supposed to be fully handled off-chain by Atum Labs.

3.2.6 Unrestricted Reserver and Releaser Selection

Severity: Informational

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

Description: Depositors can set any address as `reserver` and `releaser`, including themselves. Since these roles are assumed to be trusted in the protocol model, allowing arbitrary addresses increases the attack surface and can expose Settlers to malicious flows.

Recommendation: Explore introducing allowlists for Reserver and Releaser addresses and enforce them during deposit.

Atum Labs: Acknowledged.

Cantina Managed: Acknowledged. Off-chain solutions are supposed to protect Settlers.