Security Review

Cantina Managed

ATUM LABS: PRS 49, 50 & 51

Cantina Managed review by:
Jay, Security Researcher

CANTINA

September 21, 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 **5**
- 3.1 Medium Risk . . . 5
  - 3.1.1 Reserve time fee rescale overflows u64 and forces large deposits onto the refund only path on SVM . . . 5
- 3.2 Informational . . . 8
  - 3.2.1 Fee computation diverges between EVM and SVM in rounding direction and at the fee equals amount boundary . . . 8
  - 3.2.2 Zero amount fulfill is allowed on EVM and TVM . . . 8
  - 3.2.3 Misleading comment claims the existence check blocks completed deposits . . . 9

1 Introduction

1.1 About Cantina

Cantina is a security platform that pairs a world-class network of security researchers with agentic AI to help teams building onchain find and fix real risk. Through managed reviews, competitions, and bounties, Cantina secures protocols before vulnerabilities can be exploited.
Learn more at cantina.security

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 Aug 12th to Aug 19th the Cantina team conducted a review of protocol on commit hash e11c58be. The team identified a total of **4** issues:

Issues Found

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

2.1 Scope

The security review had the following components in scope for protocol on commit hash e11c58be:

```
contracts
├── evm
│   └── source
│       └── src
│           ├── Escrow.sol
│           ├── FulfillmentProxy.sol
│           └── interfaces
│               ├── IEscrow.sol
│               ├── IFulfillmentProxy.sol
│               └── ITokenConfig.sol
├── svm
│   └── source
│       └── programs
│           ├── escrow
│           │   └── src
│           │       ├── constants.rs
│           │       ├── errors.rs
│           │       ├── events.rs
│           │       ├── lib.rs
│           │       ├── state.rs
│           │       ├── instructions
│           │       │   └── escrow.rs
│           │       └── utils
│           │           └── signatures.rs
│           └── fulfillment_proxy
│               └── src
│                   ├── errors.rs
│                   ├── events.rs
│                   ├── lib.rs
│                   ├── state.rs
│                   └── instructions
│                       ├── fulfillment.rs
│                       └── protocol.rs
└── tvm
    └── source
        └── contracts
            ├── Escrow.sol
            ├── FulfillmentProxy.sol
```

```
└── interfaces
    ├── IEscrow.sol
    └── IFulfillmentProxy.sol
```

3 Findings

3.1 Medium Risk

3.1.1 Reserve time fee rescale overflows u64 and forces large deposits onto the refund only path on SVM

Severity: Medium Risk.

Context: escrow.rs#L527-L532, fees.rs#L39-L42.

Description: At deposit time the SVM program stores the signed authorization ceiling in `deposit.amount` and a fee snapshot in `deposit.fee_amount`. At reserve time the winning charge `source_amount` is committed, `deposit.amount` is narrowed to it, and the fee is rescaled proportionally so the settler is not underpaid. The rescale performed the multiply in `u64` as `deposit.fee_amount * source_amount / deposit.amount`.

`deposit.fee_amount`, `source_amount`, and `deposit.amount` are all `u64`. The intermediate product `fee_amount * source_amount` was formed in `u64` and overflowed `u64::MAX`, which is about 1.845e19, before the division could bring the value back down under the original fee. On overflow `checked_mul` returns `None`, which is mapped to `ErrorCode::InvalidData`, so the `reserve` instruction reverts.

`reserve` is the only place that sets `deposit.recipient`. It is guarded at the top by `require!(deposit.recipient.is_none())`, so it cannot be retried or split into multiple partial reserves. A deposit that can never be reserved can never be released. The funds are not stolen and not permanently frozen, because the depositor or releaser can still call `refund` to return the entire escrow balance to the payer. But the intended settlement to the recipient can never complete on Solana. The deposit is forced onto a path where only refund works.

The original inline comment reasoned that the result can never exceed the original fee. That is true of the final quotient but not of the product formed before the division, which is exactly the value that overflowed. The comment described the value after the divide while the failure happened before the divide.

This was unique to SVM. The deposit time `compute_fee` widens every multiply to `u128` and is safe from overflow. The EVM contract `Escrow.sol` and the TVM Tron contract `Escrow.sol` never rescale at reserve time at all. They take `sourceAmount` at deposit, store it directly as `amount`, and compute the fee once via `determineFee(sourceAmount)` in `uint256`, whose maximum is about 1.15e77. So only the SVM reserve time rescale was exposed, and it was also a cross-VM parity break. The same payment that reverted on Solana settles normally on EVM and TVM.

The trigger is purely `percentage_fee * deposit_size^2` at the worst case where `source_amount == deposit.amount`, which is a full reserve and the normal case. Higher fee rates and higher token decimals both lower the amount needed to trigger it. A configured `max_fee` clamp shrinks the stored fee and raises the threshold. A configuration with only a minimum fee and a zero percentage removes the quadratic term and is effectively not exposed.

Proof of concept: A reproduction of the exact arithmetic, using the deposit time `compute_fee` in `u128` and then the reserve time `u64` rescale with `source_amount == deposit.amount`, reverts at the following full deposit sizes. Values shown for a token with six decimals such as USDC. A token with nine decimals divides every nominal threshold by about 1000.

Fee rate	Reserve reverts for a full deposit above
0.01%	~429,497 USDC
0.1%	~135,819 USDC
0.5%	~60,740 USDC
1%	~42,950 USDC
2%	~30,370 USDC
5%	~19,208 USDC

10%	~13,582 USDC

Example at 0.1% on a six decimal USDC. A 140,000 USDC deposit stores `fee_amount = 140_000_000` atoms. At full reserve the product `140_000_000 * 140_000_000_000 = 1.96e19` exceeds `u64::MAX = 1.845e19`, so `checked_mul` returns `None` and `reserve` reverts. A 135,000 USDC deposit is just under the boundary and succeeds.

The test `f23_old_u64_rescale_overflows_on_large_full_deposit` reproduces the original `u64` arithmetic and asserts it overflows to `None` on a full reserve of a 140,000 USDC deposit at 0.1%, which documents the bug.

The tests `f23_rescale_preserves_fee_on_large_full_deposit`, `f23_rescale_is_proportional_for_partial_reserve`, and `f23_rescale_survives_extreme_u64_inputs` assert the fixed helper returns the original fee on a full reserve, scales proportionally on a partial reserve, and does not revert even at `u64::MAX` inputs.

```
// F23 regression coverage: the reserve-time fee rescale must not overflow u64 for large but
// legitimate deposits.
//
// A 6-decimal token such as USDC at a 0.1% fee. FEE_PERCENTAGE_SCALE is 1e8, so 0.1% is 100_000.
const USDC_DECIMALS: u8 = 6;
const FEE_0_1_PCT_6: u64 = 100_000;

// The exact arithmetic the reserve handler used before F23 was fixed: the product was formed in
// u64 and overflowed u64::MAX before the divide could bring it back under the original fee.
fn buggy_u64_rescale(fee_amount: u64, source_amount: u64, deposit_amount: u64) -> Option<u64> {
    fee_amount
        .checked_mul(source_amount)
        .and_then(|product| product.checked_div(deposit_amount))
}

#[test]
fn f23_old_u64_rescale_overflows_on_large_full_deposit() {
    // 140,000 USDC deposit => 140_000_000_000 atoms. compute_fee snapshots the fee in u128 and
    // it fits u64 comfortably (140_000_000 atoms).
    let amount: u64 = 140_000 * 1_000_000;
    let fee = compute_fee(amount, USDC_DECIMALS, FEE_0_1_PCT_6, 0, 0).unwrap();
    assert_eq!(fee, 140_000_000);

    // At a full reserve the pre-divide product fee * amount exceeds u64::MAX, so the old u64
    // path returned None, which the handler mapped to a revert. This documents the bug.
    assert_eq!(
        buggy_u64_rescale(fee, amount, amount),
        None,
        "the pre-fix u64 rescale overflows and reverts on a full reserve of a large deposit"
    );
}

#[test]
fn f23_rescale_preserves_fee_on_large_full_deposit() {
    // Same deposit that overflowed the old path. The u128-widened helper must return the
    // original fee unchanged for a full reserve.
    let amount: u64 = 140_000 * 1_000_000;
```

```
    let fee = compute_fee(amount, USDC_DECIMALS, FEE_0_1_PCT_6, 0, 0).unwrap();

    let rescaled = rescale_reserved_fee(fee, amount, amount).unwrap();
    assert_eq!(
        rescaled, fee,
        "a full reserve must preserve the deposit-time fee snapshot"
    );
}

#[test]
fn f23_rescale_is_proportional_for_partial_reserve() {
    let amount: u64 = 140_000 * 1_000_000;
    let fee = compute_fee(amount, USDC_DECIMALS, FEE_0_1_PCT_6, 0, 0).unwrap();

    // Reserving half the ceiling rescales the fee to half, and never exceeds the
    ↪  snapshot.
    let half = amount / 2;
    let rescaled = rescale_reserved_fee(fee, half, amount).unwrap();
    assert_eq!(rescaled, fee / 2);
    assert!(rescaled <= fee);
}

#[test]
fn f23_rescale_survives_extreme_u64_inputs() {
    // Even the largest possible u64 inputs stay inside u128 for the intermediate
    ↪  product, so a
    // full reserve at the u64 ceiling still returns the original fee instead of
    ↪  reverting.
    let max = u64::MAX;
    assert_eq!(rescale_reserved_fee(max, max, max).unwrap(), max);
}
```

Recommendation: Widen the intermediate product to `u128`, matching the overflow-safe pattern already used by `compute_fee`. Since `source_amount <= deposit.amount` is enforced above, the quotient is guaranteed to be less than or equal to the original `u64` fee, so the narrowing cast back to `u64` cannot fail for in-range inputs.

```
let fee_amount = u64::try_from(
    (deposit.fee_amount as u128)
        .checked_mul(source_amount as u128)
        .ok_or(ErrorCode::InvalidData)?
        / deposit.amount as u128,
)
.map_err(|_| ErrorCode::InvalidData)?;
```

Add a test that reserves a full deposit at a size above the overflow threshold (for example 140,000 units of a 6-decimal token at 0.1%) and asserts the rescaled fee equals the original fee, so the regression is pinned in continuous integration.

Atum Labs: Fixed in PR 51.

Cantina Managed: PR 51 fixes this by introducing a `rescale_reserved_fee` helper in `fees.rs` that widens the intermediate product to `u128` before dividing, matching the overflow-safe pattern already used by `compute_fee`. The call site in `instructions/escrow.rs` delegates to this helper, and five unit tests cover the fix — including `f23_old_u64_rescale_overflows_on_large_full_deposit`, which reproduces the original `u64` overflow through a local transcription of the pre-fix arithmetic.

3.2 Informational

3.2.1 Fee computation diverges between EVM and SVM in rounding direction and at the fee equals amount boundary

Severity: Informational.

Context: Escrow.sol#L541, Escrow.sol#L563-L564, fees.rs#L39-L42, fees.rs#L80.

Description: The percentage fee function differs between the two chains in two independent ways, both confirmed in source.

First, rounding direction. EVM rounds up using ceiling division at `Escrow.sol`, while SVM truncates toward zero using integer division at `fees.rs`. For any amount where the product of amount and percentage is not an exact multiple of the scale, the two chains produce fees that differ by one atom, the smallest unit of the token. This is a cross chain reconciliation difference only. It occurs frequently, on ordinary amounts rather than only on edge configurations, so any offchain quoting or settlement layer that assumes exact fee parity across chains will mismatch routinely. It only becomes a real problem if such a layer treats cross chain fee equality as an invariant.

Second, the fee equals amount boundary. EVM rejects the case where the fee is greater than or equal to the amount at `Escrow.sol`, which guarantees the settler always nets at least one unit. SVM instead accepts fee less than or equal to amount at `fees.rs`. On SVM a deposit whose fee equals the amount therefore settles with a net of zero. The settler or recipient receives nothing and the treasury takes the whole amount. The impact is bounded to that single degenerate deposit. Reaching it requires either a one hundred percent percentage configuration, which is admin gated and self evidently misconfigured, or an ordinary minimum fee configuration where the deposit amount is exactly the minimum fee so the minimum clamp raises the fee to equal the amount. On EVM that same deposit reverts up front.

The reason this is worth recording despite the low on chain impact is process and parity. EVM added the greater than or equal guard specifically in response to an auditor proof of concept and pins it with tests, while SVM shipped the less than or equal comparator with no equivalent test, so the exact fix the EVM audit produced was never mirrored to SVM essentially.

Recommendation: Pick one convention and apply it to both chains. For rounding, switch SVM to ceiling division to match EVM, since ceiling is protocol favoring and matches the EVM comment intent, or switch EVM to floor. For the boundary, use the same comparator on both so that a fee equal to the amount is rejected on both chains, which means changing the SVM check from fee less than or equal to amount to fee strictly less than amount. Add a cross VM parity test that pins the fee output for a vector of amounts and configurations, including the non exact amounts and the fee equals amount case, so future drift is caught in continuous integration. Before finalizing, confirm in the SVM model who receives the net amount, the settler or the recipient, so the description of who loses is exact.

Atum Labs: Fixed in PR 51.

Cantina Managed: PR 51 addresses both divergences: `compute_fee` in `fees.rs` now uses ceiling division with the same `(a*b + c - 1) / c` form as `Escrow.determineFee`, and the boundary check is tightened to a strict `<`, rejecting `fee == amount` on SVM as EVM and TVM already do. A new `cross_vm_fee_parity_tests` suite asserts `compute_fee` against an independent transcription of `Escrow.determineFee` over 72 (amount, config) pairs, including non-exact multiples, both clamps, the smallest taxable deposit, and the `fee == amount` boundary case.

3.2.2 Zero amount fulfill is allowed on EVM and TVM

Severity: Informational.

Context: FulfillmentProxy.sol#L25, FulfillmentProxy.sol#L31, FulfillmentProxy.sol#L39, FulfillmentProxy.sol#L37, FulfillmentProxy.sol#L43, FulfillmentProxy.sol#L51.

Description: The `FulfillmentProxy` on EVM does not check that the fulfill amount is positive. On EVM the functions `fulfill`, `fulfillMany`, and `strictFulfill` have no amount greater than zero

guard. On TVM the functions `fulfill` and `fulfillMany` also have no guard, but TVM `strictFulfill` was hardened under CORE-2116 and now rejects a zero amount with `ZeroAmount`, so TVM `strictFulfill` is not affected. These functions are permissionless, marked public and `whenNotPaused`, with no settler allowlist on EVM. The SVM program guards both of its paths in `fulfillment.rs` with a `require` that the amount is greater than zero.

Because ERC20 permits zero value transfers, anyone can call `fulfill` with an amount of zero, or `fulfillMany` with an empty array which is even cheaper, and emit a genuine `Fulfilled` event that carries a zero amount. On its own the impact is low and amounts to spam and observability noise that every indexer must filter.

The larger concern is that this feeds the strict fulfill escalation. A zero value `fulfill` emits a real `Fulfilled` event bound to a quote hash. If the offchain settlement matcher keys on the quote hash alone rather than on recipient, token, amount, and settler, an attacker zero value event could authorize a real release and become theft. The missing amount guard is the cheap primitive that makes that escalation trivial on EVM and TVM.

Recommendation: Add a require that the amount is greater than zero in `fulfill`, and per item in `fulfillMany`, on both EVM and TVM to match SVM. On EVM also add the guard to `strictFulfill`. TVM `strictFulfill` already enforces it. Independently, require the offchain matcher to bind the quote hash to recipient, token, amount, and settler, so that a zero or mismatched `Fulfilled` event can never authorize settlement.

Atum Labs: Fixed in PR 49.

Cantina Managed: PR 49 (EVM) and PR 50 (TVM) add zero-amount guards to `fulfill` and every item of `fulfillMany` on both VMs, and introduce a new `EmptyFulfillmentBatch` error that rejects empty-array calls. Tests that previously asserted zero-amount fulfillment events were inverted rather than deleted to document the change in intent, and the ABI and `abigen` bindings were regenerated on both VMs.

3.2.3 Misleading comment claims the existence check blocks completed deposits

Severity: Informational.

Context: Escrow.sol#L225.

Description: The comment states that the existence check blocks re use of deposits that are either active or completed. The guard only requires that the stored deposit hash equals zero before allowing a new deposit. Completed deposits are deleted back to zero in both the release and refund paths:

```
// release path
delete s_activeDepositHashes[releaseInput.releaseWitness.depositId];

// refund path
delete s_activeDepositHashes[refundInput.refundWitness.depositId];
```

Because a completed deposit hash is zero, the equals zero check cannot detect it. It only rejects a still active duplicate.

Re-use of a completed deposit id is actually prevented by the spent Permit2 nonce, because `permitWitnessTransferFrom` reverts on the consumed nonce. That is a different mechanism than the comment implies. The interface already documents the correct behavior, which directly contradicts the inline comment:

```
// Deposit not found - either never existed or already completed (storage cleared)
```

This is not a vulnerability, because the nonce guarantee holds for every deposit that goes through Permit2. The risk is future maintenance. A later change could remove the nonce dependency in the

belief that this check already covers completed deposits, which would reopen re-use of a completed `depositId`.

Recommendation: Reword the comment to state that it rejects a still active duplicate only, and that re-use of a completed deposit id is prevented by the spent Permit2 nonce. Cross-reference the interface documentation in `IEscrow.sol` so the two descriptions agree.

Atum Labs: Fixed in PR 49.

Cantina Managed: PR 49 rewrites the misleading inline comment in `Escrow.sol` to clarify that the guard rejects only still-active duplicates, and that re-use of a completed deposit id is prevented by the spent Permit2 nonce — since `depositId` is `keccak256(depositor, depositSignature, permit.nonce)` and `permitWitnessTransferFrom` consumes it, replaying the id necessarily replays a spent nonce. The inline comment and `IEscrow.sol` interface documentation are now consistent.