# Unichain contracts

## Security Review

Cantina Managed review by:
**Akshay Srivastav**, Security Researcher
**Anurag Jain**, Security Researcher

October 16, 2024

# Contents

# 1 Introduction

## 1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

## 1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

## 1.3 Risk assessment

| Severity | Description |
|---|---|
| **Critical** | *Must* fix as soon as possible (if already deployed). |
| **High** | Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users. |
| **Medium** | Global losses <10% or losses to only a subset of users, but still unacceptable. |
| **Low** | Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies. |
| **Gas Optimization** | Suggestions around gas saving practices. |
| **Informational** | Suggestions around best practices or readability. |

### 1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

# 2 Security Review Summary

Uniswap is an open source decentralized exchange that facilitates automated transactions between ERC20 token tokens on various EVM-based chains through the use of liquidity pools and automatic market makers (AMM).

From Oct 3rd to Oct 7th the Cantina team conducted a review of unichain-contracts on commit hash b7d5bbb0. The team identified a total of **9** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 1
- Medium Risk: 2
- Low Risk: 2
- Gas Optimizations: 1
- Informational: 3

# 3 Findings

## 3.1 High Risk

### 3.1.1 `recipient` loses all his allocation on their admin revocation

**Severity:** High Risk

**Context:** NetFeeSplitter.sol#L48-L51

**Description:** A `recipient` can lose all their allocation if their admin has revoked itself.

**Impact:** `recipient` loses all his allocation and hence funds.

**Likelihood:** Medium as this will require `recipient` admin to have revoked itself

**Proof of Concept:**

1. Admin revokes itself using `transferAdmin` function for its recipient `R1` having an allocation of say `10%`. So now `adminOf(R1) = address(0)`:

```
function transferAdmin(address recipient, address newAdmin) external {
    // TODO: allow newAdmin == address(0)?
    address currentAdmin = adminOf(recipient);
    if (currentAdmin != msg.sender) revert Unauthorized();
    recipients[recipient].admin = newAdmin;
    emit TransferAdmin(recipient, currentAdmin, newAdmin);
}
```

2. Immediately Admin of recipient `R2` transfers `0` allocation amount to `R1`.
3. Since `R1` admin is now `address(0)` so this function incorrectly assumes that `R1` is a new recipient and reinitializes it with `0` allocation:

```
function transfer(address from, address recipient, uint256 allocation) external {
    // ...
    if (adminOf(recipient) == address(0)) {
        // recipient does not exist yet, make recipient the admin
        recipients[recipient] = Recipient(recipient, 0);
    }
```

4. The `10%` allocation for `R1` will be reset to `0` and is lost forever with no way of recovery.
5. Any further payment distribution causes 10% of amount to remain stuck in contract

- Test case:

```
function test_POC() public {
    address admin_ = makeAddr('admin');
    address admin2_ = makeAddr('admin2');
    address recipient_ = makeAddr('recipient');
    address recipient2_ = makeAddr('recipient2');
    address[] memory recipients = new address[](2);
    recipients[0] = recipient_;
    recipients[1] = recipient2_;
    INetFeeSplitter.Recipient[] memory recipientData = new INetFeeSplitter.Recipient[](2);
    recipientData[0] = INetFeeSplitter.Recipient({admin: admin_, allocation: 5_000});
    recipientData[1] = INetFeeSplitter.Recipient({admin: admin2_, allocation: 5_000});
    NetFeeSplitter splitter = new NetFeeSplitter(recipients, recipientData);

    address newAdmin_ = address(0);
    vm.prank(admin_);
    splitter.transferAdmin(recipient_, newAdmin_);
    assertEq(splitter.adminOf(recipient_), newAdmin_);

    assertEq(splitter.balanceOf(recipient_), 5_000);

    vm.prank(admin2_);
    splitter.transfer(recipient2_, recipient_, 0);

    // Bug, below passes and confirms recipient_ becomes 0
    assertEq(splitter.balanceOf(recipient_), 0);
}
```

**Recommendation:**

1. `recipient` should not be created if `recipients[recipient].allocation > 0`:

```
function transfer(address from, address recipient, uint256 allocation) external {

--        if (adminOf(recipient) == address(0)) {
++        if (adminOf(recipient) == address(0) && recipients[recipient].allocation==0) {
            // recipient does not exist yet, make recipient the admin
            recipients[recipient] = Recipient(recipient, 0);
        }
```

2. OR, if admin revocation is not actually required then modify `transferAdmin` to disallow `address(0)`:

```
function transferAdmin(address recipient, address newAdmin) external {
    if (newAdmin == address(0)) revert AdminZero();
    // ...
}
```

**Uniswap:** Fixed in commit 5992d6d3.

**Cantina Managed:** Fixed. Admin rights cannot be renounced now.

## 3.2 Medium Risk

### 3.2.1 `NetFeeSplitter`: admin rights can be recovered after renouncement

**Severity:** Medium Risk

**Context:** NetFeeSplitter.sol#L48-L51

**Description:** The `NetFeeSplitter.transfer` function initializes the `admin` of a recipient when the recipient's admin is `address(0)`.

This creates an edge case for a recipient whose admin rights have been renounced. For admin renounced recipients this behaviour can restore their admin rights again.

**Scenario:**

- The admin rights of recipient `A` gets renounced by calling `transferAdmin(A, address(0))`.
- Admin of any other recipient performs an allocation transfer of any amount to recipient `A`.
- The admin of recipient `A` gets set to A's address itself.

**Recommendation:** Consider implementing an additional flag to differentiate between an existing and fresh recipient. Then initialize the admin for a fresh recipient only.

**Uniswap:** Fixed in 5992d6d3.

**Cantina Managed:** Fixed. Admin rights cannot be renounced now.

### 3.2.2 Net fee will be misinterpreted as L1 fee

**Severity:** Medium Risk

**Context:** FeeSplitter.sol#L58-L60

**Description:** `LOCK_STORAGE_SLOT` is set to `1` in transient storage and is never reset back to `0` even after `distributeFees` completes. This means this value will only reset after transaction completes. This can cause incorrect accounting where Net fees goes to L1 fee recipients.

**Impact:** Net fees recipients lose their funds and L1 fees recipient gets incorrectly funded.

**Likelihood:** Low, since this will require User to follow an unconventional call route.

**Proof of Concept:** User calls a custom contract which executes below in a single transaction:

1. Calls `distributeFees` function which first set `LOCK_STORAGE_SLOT` to `1` and then distributes all pending fees:

```
assembly ("memory-safe") {
    tstore(LOCK_STORAGE_SLOT, 1)
}
```

2. Observe that it does not reset `LOCK_STORAGE_SLOT` back to `0`
3. Next Contract donates a Grant to Sequencer Vault by transferring funds to it directly:

```
receive() external payable { }
```

4. Next contract calls `withdraw` function for `SEQUENCER_FEE_WALLET` directly:

```
function withdraw() external {
    // ...
}
```

5. This succeeds since `LOCK_STORAGE_SLOT` is still `1` from step 2 (transient storage only reset once transaction completes).
6. This increases `NET_REVENUE_STORAGE_SLOT` to withdrawn `SEQUENCER_FEE_WALLET` fees say `X`:

```
receive() external payable virtual {
    uint256 unlocked;
    assembly ("memory-safe") {
        unlocked := tload(LOCK_STORAGE_SLOT)
    }
    if (unlocked == 0) revert Locked();

    if (msg.sender == Predeploys.SEQUENCER_FEE_WALLET || msg.sender == Predeploys.BASE_FEE_VAULT) {
        // ...
        assembly ("memory-safe") {
            tstore(NET_REVENUE_STORAGE_SLOT, add(tload(NET_REVENUE_STORAGE_SLOT), amount))
        }
    } else if (msg.sender == Predeploys.L1_FEE_VAULT) {
        // ...
    } else {
        // ...
    }
}
```

7. Transaction completes and `NET_REVENUE_STORAGE_SLOT` is reset to `0`.
8. Now after some time when Admin genuinely calls `distributeFees` then `X` will be part of `L1 fees` instead of `Net fees`.

**Flow:**

User → Contract A → `distributeFees` → `LOCK_STORAGE_SLOT` Unlock → `distributeFees` completes → Grant X amount to Sequencer Vault transferred → withdraw on `SEQUENCER_FEE_WALLET` → `NET_REVENUE_STORAGE_SLOT` becomes X → Transaction completes → `NET_REVENUE_STORAGE_SLOT` is reset to 0.

User/Admin → `distributeFees` → X which is part of `msg.value` becomes L1 fees instead of sequencer fees.

**Proof of Concept:**

```
function test_POC(
    uint256 sequencerFee,
    uint256 baseFee,
    uint256 l1Fee
) public {
    sequencerFee = bound(sequencerFee, 1, 100 ether);
    baseFee = bound(baseFee, 1, 100 ether);
    l1Fee = bound(l1Fee, 1, (sequencerFee + baseFee) * 150 / 25 - sequencerFee - baseFee);

    vm.deal(Predeploys.SEQUENCER_FEE_WALLET, sequencerFee);
    vm.deal(Predeploys.BASE_FEE_VAULT, baseFee);
    vm.deal(Predeploys.L1_FEE_VAULT, l1Fee);

    uint256 expectedOpShare = (sequencerFee + baseFee) * 150 / 1000;
    uint256 expectedNetRevenueShare = (sequencerFee + baseFee) - expectedOpShare;
    uint256 expectedL1Share = l1Fee;

    vm.expectEmit(true, true, false, true);
    emit IFeeSplitter.FeesDistributed(expectedOpShare, expectedL1Share, expectedNetRevenueShare);
    bool feesDistributed = feeSplitter.distributeFees();
    assertTrue(feesDistributed, 'Fees were not distributed');
    assertEq(opWallet.balance, expectedOpShare, 'Op wallet balance is not expected');
    assertEq(l1Recipient.balance, expectedL1Share, 'L1 recipient balance is not expected');
    assertEq(netRecipient.balance, expectedNetRevenueShare, 'Net recipient balance is not expected');

    vm.deal(Predeploys.SEQUENCER_FEE_WALLET, 1);
    vm.prank(Predeploys.SEQUENCER_FEE_WALLET);
    (bool success, bytes memory revertData) = address(feeSplitter).call{value: 1}('');
    // Bug, this shows that Lock is not revoked and funds can be given directly to feeSplitter
    assert(success);
}
```

**Recommendation:** Reset `LOCK_STORAGE_SLOT` to `0` once `withdraw` from all vault is complete:

```
  // unlock
  assembly ("memory-safe") {
      tstore(LOCK_STORAGE_SLOT, 1)
  }
  _feeVaultWithdrawal(Predeploys.SEQUENCER_FEE_WALLET);
  _feeVaultWithdrawal(Predeploys.BASE_FEE_VAULT);
  _feeVaultWithdrawal(Predeploys.L1_FEE_VAULT);

+ // lock
+ assembly ("memory-safe") {
+     tstore(LOCK_STORAGE_SLOT, 0)
+ }
```

**Uniswap:** Fixed in commit 9aec0878.

**Cantina Managed:** Fixed as recommended.

## 3.3 Low Risk

### 3.3.1 `NetFeeSplitter.withdrawFees` can be called by non-recipients resulting in their `_indexOf` state change

**Severity:** Low Risk

**Context:** NetFeeSplitter.sol#L101-L104, NetFeeSplitter.sol#L71-L73

**Description:** Non-recipient users with `0` allocation can also call the `withdraw` function. This function internally calls the `_updateFees` function which updates the `_indexOf` state of `msg.sender`.

Non-recipient accounts can call `withdraw` and update their `_indexOf` state with latest `_index` value.

**Recommendation:** Consider reverting in `withdraw` for users whose `earned[msg.sender]` and `recipients[msg.sender].allocation` both are `0`.

**Uniswap:** Acknowledged.

**Cantina Managed:** Acknowledged.

### 3.3.2 `L1Splitter` does not honour `FEE_DISBURSEMENT_INTERVAL` delay for the first withdrawal

**Severity:** Low Risk

**Context:** L1Splitter.sol#L28-L33

**Description:** The `L1Splitter` contract intends to enforce a minimum delay of `FEE_DISBURSEMENT_INTERVAL` for every `withdraw` operation. But this delay is not honoured for the first withdrawal transaction.

The `lastDisbursementTime` is `0` for the first withdrawal which means that first withdrawal can be triggered as soon as the `L1Splitter` contract is deployed and contains `0.1` ETH.

**Recommendation:** Consider initializing the `lastDisbursementTime` in the constructor.

```
  constructor(address l1Wallet, uint256 feeDisbursementInterval) {
      L1_WALLET = l1Wallet;
      FEE_DISBURSEMENT_INTERVAL = feeDisbursementInterval;
+     lastDisbursementTime = block.timestamp;
  }
```

**Uniswap:** Acknowledged.

**Cantina Managed:** Acknowledged.

## 3.4 Gas Optimization

### 3.4.1 Return if `grossFeeRevenue` is `0`

**Severity:** Gas Optimization

**Context:** FeeSplitter.sol#L66

**Description:** Fee distribution will happen even with `0` amount.

1. `minWithdrawalAmount` can be `0` for all 3 Vaults.
2. This means `distributeFees` will collect `0` fees and will try sending `0` fees to splitter.
3. This causes unnecessary gas since no fees is actually distributed in this case.

**Recommendation:** Return if `grossFeeRevenue==0`:

```
  uint256 grossFeeRevenue = address(this).balance;
  assembly ("memory-safe") {
      netFeeRevenue := tload(NET_REVENUE_STORAGE_SLOT)
      tstore(NET_REVENUE_STORAGE_SLOT, 0)
  }
+ if(grossFeeRevenue==0) { emit NoFeesCollected(); return false; }
```

**Uniswap:** Fixed in commit 4a9abf72.

**Cantina Managed:** Fixed as recommended.

## 3.5 Informational

### 3.5.1 `NetFeeSplitter:` Rename mapping `earned` to `_earned`

**Severity:** Informational

**Context:** NetFeeSplitter.sol#L14

**Description:** Since the `earned` state variable is defined as `private`, its name should start with an `underscore (_)` so that the same naming convention gets followed across contracts.

**Recommendation:**

```
- mapping(address recipient => uint256 earned) private earned;
+ mapping(address recipient => uint256 earned) private _earned;
```

**Uniswap:** Fixed in commit d9dffca7.

**Cantina Managed:** Fixed as recommended.

### 3.5.2 `L1Splitter`: Additional delay of `withdraw` operation

**Severity:** Informational

**Context:** L1Splitter.sol#L28-L33

**Description:** The `L1Splitter` contract intends to enforce a minimum delay of `FEE_DISBURSEMENT_INTERVAL` duration for every `withdraw` operation.

This delay enforcement can be misused to cause an additional delay for withdrawal execution.

**Scenario:**

- Suppose the `FEE_DISBURSEMENT_INTERVAL` is set as `1` day in `L1Splitter`.
- `L1Splitter` contains `0.1` ETH and is about to receive a significant amount of more ETH (let's say 10 ETH).
- `withdraw` gets called by anyone before those `10` ETH arrive.
- Hence forcing an additional withdrawal delay of `1` day for the `10` ETH.

**Uniswap:** Acknowledged.

**Cantina Managed:** Acknowledged.

### 3.5.3 Fee distribution logic can be tricked by either party

**Severity:** Informational

**Context:** FeeSplitter.sol#L79-L86

**Description:** Fee distribution will be borne by `Net fee recipient+L1 fee recipient` or `Net fee recipient`. `L1 fee recipient` will always try to create situation where they don't participate in sharing fees to Optimism. Similarly, `Net fee recipient` will always try to create situation where `L1 fee recipient` participate in fee distribution to Optimism

Below is one of the ways:

1. If `grossRevenueShare > netRevenueShare` and `l1 fee` recipient see that donating some amount to `Base or Sequencer vault` will reverse the situation such that `amount donated < 2.5% of l1 fees` then L1 can trick the fee system. Now all fees to optimism will be borne only by sequencer and base fee vault.
2. Similarly, if `Net fee recipient` see that donating amount to `L1 fee vault` will cause `L1 fee vault` to also participate in fees to optimism and `donated amount < Net fee recipient gain from L1 fee vault participation` then `Net fee recipient` can trick the fee system.

**Uniswap:** Acknowledged.

**Cantina Managed:** Acknowledged.