CANTINA

Unichain contracts

Security Review

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

November 5, 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 31st to Nov 2nd the Cantina team conducted a review of unichain-contracts on commit hash 935f521c. The team identified a total of **6** issues in the following risk categories:

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

3 Findings

3.1 Low Risk

3.1.1 Owner is allowed to revoke itself

Severity: Low Risk

Context: L1Splitter.sol#L12

Description: It was observed that `L1Splitter` contract is configuring multiple contract configuration through `Owner` role. Setting of `Owner` is achieved by implementing `Ownable2Step` contract. The issue here is that `Ownable2Step` also allows `Owner` to revoke its role. If `Owner` ever revoke itself then `L1Splitter` contract will be left with no way to maintain its configuration

Impact: Without `Owner`, configuration like `l1Recipient`, `feeDisbursementInterval`, `minWithdrawalAmount` cannot be altered.

Likelihood: Low, Owner is a trusted entity.

Proof of Concept: Owner of `L1Splitter` contract calls `renounceOwnership` function and renounces its role.

Recommendation: Override `renounceOwnership` function in `L1Splitter` to revert which will prevent `Owner` from renouncing its role.

Uniswap: Acknowledged. Keeping the `renounceOwnership` function is fine.

Cantina Managed: Acknowledged.

3.1.2 On `L1Splitter` anyone can perform an L1 withdrawal just before the configuration functions are about to get executed

Severity: Low Risk

Context: L1Splitter.sol#L36, L1Splitter.sol#L54-L66

Description: The `L1Splitter` contract has `updateL1Recipient`, `updateFeeDisbursementInterval` and `updateMinWithdrawalAmount` configuration functions by which the `owner` of the contract can change withdrawal configuration of the contract. It also has a publicly available `withdraw` function to initiate an L1 withdrawal.

Anyone can perform the L1 withdrawal just before the configuration functions are about to get executed and force a withdrawal according to the older configurations. This can be done by monitoring the upcoming configuration changes for `L1Splitter` contract.

Recommendation: There can be multiple mitigations for this issue:

1. Restrict the `withdraw` function to owner by adding the `onlyOwner` modifier.
2. Try to attempt a withdrawal before every configuration change. This can be done manually or at smart contract level like:

```
function updateMinWithdrawalAmount(uint256 newAmount) public onlyOwner {
    if (
        address(this).balance >= minWithdrawalAmount
        && block.timestamp >= lastDisbursementTime + feeDisbursementInterval
    ) {
        withdraw();
    }
    _updateMinWithdrawalAmount(newAmount);
}
// same for other functions
```

3. Leave the implementation as it is if this is an accepted risk.

Uniswap: Acknowledged.

Cantina Managed: Acknowledged.

3.2 Gas Optimization

3.2.1 `NetFeeSplitter._transfer` should revert early in case of insufficient allocation

Severity: Gas Optimization

Context: NetFeeSplitter.sol#L107

Description: The `NetFeeSplitter._transfer` function performs `_updateFees` before checking sufficient allocation of `oldRecipient`. This check should be done early to revert early and save gas in case of insufficient allocation. It is a good practice to perform all necessary validation checks before updating any contract states.

Recommendation:

```
  function _transfer(address oldRecipient, address newRecipient, uint256 allocation) private {
      if (setterOf(oldRecipient) != msg.sender) revert Unauthorized();
      if (newRecipient == address(0)) revert RecipientZero();
      if (allocation == 0) revert AllocationZero();
+     if (balanceOf(oldRecipient) < allocation) revert InsufficientAllocation();
      _updateFees(oldRecipient);
      _updateFees(newRecipient);

-     if (balanceOf(oldRecipient) < allocation) revert InsufficientAllocation();
      recipients[oldRecipient].allocation -= allocation;
      recipients[newRecipient].allocation += allocation;
      emit AllocationTransferred(msg.sender, oldRecipient, newRecipient, allocation);
  }
```

Uniswap: Fixed in commit fb9024ae.

Cantina Managed: Fixed.

3.3 Informational

3.3.1 Rename references of `admin` to `setter` in `NetFeeSplitter`

Severity: Informational

Context: NetFeeSplitter.sol#L54, INetFeeSplitter.sol#L78

Description: The input parameter `newAdmin` of `NetFeeSplitter.transferAllocationAndSetSetter` functions should be renamed to `newSetter` as all other references to admin has been renamed to setter now.

Recommendation: Rename `newAdmin` to `newSetter` in `NetFeeSplitter` contract and `INetFeeSplitter` interface.

Uniswap: Fixed in commit d7db41e6.

Cantina Managed: Fixed.

3.3.2 Frontrunning can cause DOS on `transferAllocationAndSetSetter`

Severity: Informational

Context: NetFeeSplitter.sol#L59

Description: Contract allows any user with allocation to become admin of any random address. If `newRecipient` is supposed to be a contract, attacker can frontrun `transferAllocationAndSetSetter` call and set himself as `newRecipient` setter

Impact: Attacker wont gain anything since allocation remains `0` but the Victim setter will now need to redeploy a new `newRecipient` contract for usage in `transferAllocationAndSetSetter`, which again could be frontrun.

Likelihood: Low, recipients are known entities

Proof of Concept:

1. User A deploys contract R1 which is supposed to be a recipient.
2. User A calls `transferAllocationAndSetSetter` to become setter for `R1`.

3. User B frontuns `transferAllocationAndSetSetter` and becomes setter for `R1`.
4. User A call fails, forcing user to redeploy a new Recipient contract.

Recommendation: If recipient is a contract then bundle recipient contract creation call with `transferAllocationAndSetSetter` function call so that other's cannot frontrun `transferAllocationAndSetSetter` and become setter of recipient.

Uniswap: Acknowledged. All recipients are known entities, risk of malicious action is low. No risk of loss of allocation. Transaction bundle could be used to avoid frontrunning.

Cantina Managed: Acknowledged.

3.3.3 Recipient Setter cannot be created with `0` allocation amount by valid existing Setter

Severity: Informational

Context: NetFeeSplitter.sol#L103

Description: If existing Setter (with `> 0` allocation) wants to create a new Recipient Setter (using `transferAllocationAndSetSetter`) without sending any initial allocation then it is not possible as transfer enforces `> 0` allocation.

Impact: New Setter creation with `0` allocation wont be possible even with existing setter with `> 0` allocation.

Proof of Concept:

1. Valid Setter S1 (with existing allocation) wants to create Setter S2 for a new Recipient R2.
2. S1 does not want to allocate right away but will be allocating to R2 in future.
3. Thus Setter of R1 simply calls `transferAllocationAndSetSetter` with `0` allocation.
4. Since `0` allocation is not allowed, operation fails.

```
if (allocation == 0) revert AllocationZero();
```

Recommendation: If creation of new Recipient Setter from an existing Setter with >0 allocation is expected then make below changes in `_transfer` function.

```
- if (allocation == 0) revert AllocationZero();
+ if (balanceOf(oldRecipient) == 0) revert InsufficientAllocation();
```

Uniswap: This issue is acknowledged.

Cantina Managed: Acknowledged.