OpenZeppelin | security

Uniswap Labs Phoenix Fees Phase 2 Audit

October 10, 2025

Table of Contents

Summary

Type	DeFi
Timeline	From 2025-09-23 To 2025-09-25
Languages	Solidity

Total Issues	13 (10 resolved)
Critical Severity Issues	2 (2 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	0 (0 resolved)
Low Severity Issues	3 (2 resolved)
Notes & Additional Information	8 (6 resolved)

Scope

OpenZeppelin audited the Uniswap/phoenix-fees repository at commit 296a2d2.

In scope were the following files:

```
src
├── UNIMinter.sol
├── UniVesting.sol
├── interfaces
│   ├── IUNI.sol
│   └── IUniVesting.sol
└── libraries
    └── VestingLib.sol
```

System Overview

The audited codebase is a subsystem that manages UNI token issuance and time-based distribution. It separates responsibilities among the following contracts:

- `UNIMinter`: A minter contract that holds the right to mint UNI tokens and to allocate annual inflation to recipients by shares.
- `UniVesting`: A vesting contract that releases tokens over discrete periods aligned with the UNI token's minting cadence.
- `VestingLib`: A small math library that safely mixes signed/unsigned accounting for vesting leftovers.

`UNIMinter`

The `UNIMinter` contract holds the UNI token's minter role and coordinates the annual issuance of new UNI. It lets governance allocate the yearly mint (capped at 2% of the total supply) across multiple recipients using a simple share system. In total, it allows for 10,000 shares to be allocated representing 100% of the annual mint. If not all shares are allocated, the unassigned portion is not minted, directly reducing annual inflation.

Shares are minted on an annual basis following the limitations set by the UNI token contract. The admin is able to revoke shares with a given delay that is decided upon by the admin at the time when the shares are granted. If the removal of shares falls within the next minting period, the recipient will receive a portion of the original amount given the time of the final removal. If the time of the removal falls within the current minting period, the recipient will not receive any shares in the next year.

`UniVesting`

The `UniVesting` contract is responsible for releasing any UNI tokens it receives over the course of the year in discrete periods. With the current expected configuration, it will vest the tokens over the span of a year with releases happening every month. Funds can be claimed by the owner of the `UniVesting` contract using the `claim` function. Any unclaimed leftovers from prior windows are carried forward and remain immediately claimable.

Security Model and Trust Assumptions

The `owner` controls multiple critical parameters regarding the minting of UNI tokens. It is assumed that it will not misuse its permissions which include:

- changing the minter in the UNI token contract through the `UNIMinter` contract
- granting and revoking shares in the `UNIMinter` contract

Critical Severity

C-01 Partially Revoked Shares Can Be Reduced to Zero

When the owner initiates a share revocation through `initiateRevokeShares`, the system is designed to handle two scenarios: complete removal if the revocation completes before the next mint, or proportional reduction if it spans across multiple mint periods.

In cases where a revocation extends into the next mint period, the `initiateRevokeShares` function calculates a proportional reduction of the user's shares based on the remaining time until revocation. It updates the share amount and reduces the total allocated shares accordingly. However, the function can be executed multiple times, each time further reducing the user's shares. Since the function is executable by any caller, a malicious actor can repeatedly call the function to reduce the user's shares to effectively zero.

Consider updating the design to reduce the user's shares in the `initiateRevokeShares` function and removing that part of the logic from the `revokeShares` function.

Update: *Resolved in pull request #61. The Uniswap Labs team added a boolean flag* `adjustedForRevocation` *which prevents multiple proportional revocations from occurring, effectively solving the issue.*

C-02 Wrong Division Factor Leads to UNI Tokens Being Stuck

The `UNIMinter` contract manages UNI token inflation by distributing newly minted tokens proportionally among shareholders. The contract mints a percentage of tokens based on allocated shares relative to the maximum possible shares, then distributes these tokens to individual recipients. While the minting calculation correctly uses `totalShares` as the numerator when determining how much to mint, the individual recipient distribution incorrectly uses `MAX_SHARES` as the denominator instead of `totalShares`. This creates a fundamental mismatch where the sum of all individual distributions will be less than the total minted amount.

As an example, let us consider scenario where 8,000 out of 10,000 possible shares are allocated and that, for simplicity, only one allocation exists. The contract correctly mints 16 million UNI tokens representing 80% of the maximum 20 million token inflation cap. However, when distributing to recipients, each holder receives their share amount divided by the full 10,000 `MAX_SHARES` rather than the actual 8,000 `totalShares`. This means that only 12.8 million tokens are distributed while 3.2 million tokens remain permanently locked in the contract with no recovery mechanism since the contract lacks any sweep or recovery function.

Consider changing the distribution denominator from `MAX_SHARES` to `totalShares` in the `mint` function's recipient calculation. This ensures mathematical consistency between the minting phase and distribution phase, guaranteeing that all minted tokens are distributed proportionally among actual shareholders. In addition, consider implementing a recovery mechanism for previously locked tokens to address any existing accumulated losses from prior minting cycles or the accumulated dust.

Update: *Resolved in pull request #63.*

Low Severity

L-01 Missing Docstrings

Throughout the codebase, multiple instances of missing docstrings were identified:

- In `UniVesting.sol`, the `UniVesting` contract has no docstrings above its own definition.
- In `IUNI.sol`, the interface has no docstrings at all.

Consider thoroughly documenting all functions (and their parameters) that are part of any contract's public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Resolved in pull request #62.*

L-02 Tokens in `UniVesting` Will Vest 5 Days Earlier Than Expected

The `UniVesting` contract vests tokens by periods instead of linearly. With the current configuration, the total vesting period will be 365 days, while each individual period will be 30 days long. Since the total vesting period is not an exact multiple of the period length, the total number of periods will round down to 12. Therefore, the tokens will vest five days earlier than expected.

Consider updating the `UniVesting` contract to vest tokens linearly throughout the total vesting period or adjusting the period duration to be closer to 365 days over the course of 12 periods.

Update: *Acknowledged, not resolved. The Uniswap Labs team stated:*

> *The purpose of the vesting is to smoothen out the circulating supply of newly minted tokens. The legal team did not have issues if the newly minted tokens vest over 360 days instead of 365 days.*

L-03 Removing and Re-Adding the UniVesting Contract as a Recipient Can Be Abused to DoS the Contract for One Vesting Period

In the `UniVesting` contract, anybody can initiate vesting tokens as long as new UNI tokens have been minted. Under the rare circumstance in which the `UniVesting` contract is removed as a recipient from the `UNIMinting` contract for at least one period, such that a mint happens without the `UniVesting` contract being used and being intended to be re-added as a recipient, it is possible for an attacker to front-run the `start` function call to DoS the vesting for a whole period (one year). Before the next mint is executed with the `UniVesting` contract being set as a recipient, the attacker would send a small amount of UNI to the contract and initiate the vesting. At this point, any subsequent call to `start` will revert because the attacker's UNI tokens would not have fully vested yet. Any UNI sent to the vesting contract after that would be locked for the year.

Consider initiating the vesting even if the contract is not in use to update the `mintingAllowedAfterCheckpoint` value or deploying a new `UniVesting` contract in case the previous one was removed from the `UNIMinting` contract.

Update: *Resolved in pull request #70. The team has solved the issue by making it economically unresonable for an attacker to run a DoS attack as described. A new* `MINIMUM_UNI_TO_VEST` *constant variable set to 1000 UNI has been introduced so that the* `start` *function now checks for* `amountVesting` *to be greater or equal than that.*

Notes & Additional Information

N-01 Missing SPDX License Identifier

The `VestingLib.sol` file lacks an SPDX license identifier.

To avoid legal issues regarding copyright and follow best practices, consider adding SPDX license identifiers to files as suggested by the Solidity documentation.

Update: *Resolved in pull request #65 at commit 2d00d94.*

N-02 Missing Security Contact

Providing a specific security contact (such as an email address or ENS name) within a smart contract significantly simplifies the process for individuals to communicate if they identify a vulnerability in the code. This practice is quite beneficial as it permits the code owners to dictate the communication channel for vulnerability disclosure, eliminating the risk of miscommunication or failure to report due to a lack of knowledge on how to do so. In addition, if the contract incorporates third-party libraries and a bug surfaces in those, it becomes easier for their maintainers to contact the appropriate person about the problem and provide mitigation instructions.

Throughout the codebase, multiple instances of contracts not having a security contact were identified:

- The `UNIMinter` contract
- The `UniVesting` contract
- The `IUNI` interface
- The `IUniVesting` interface
- The `VestingLib` library

Consider adding a NatSpec comment containing a security contact above each contract definition. Using the `@custom:security-contact` convention is recommended as it has been adopted by the OpenZeppelin Wizard and the ethereum-lists.

Update: *Resolved in pull request #65 at commit c63aa89.*

N-03 Function Visibility Overly Permissive

The visibility of the `claim` function in `UniVesting.sol` is `public`, but it could be limited to `external`.

To better convey the intended use of functions and to potentially realize some additional gas savings, consider changing a function's visibility to be only as permissive as required.

Update: *Resolved in pull request #65 at commit d6ed20f.*

N-04 Custom Errors in `require` Statements

Since Solidity version `0.8.26`, custom error support has been added to `require` statements. Initially, this feature was only available through the IR pipeline. However, Solidity version `0.8.27` extended support for this feature to the legacy pipeline as well.

In `UNIMinter.sol`, multiple instances of `if-revert` statements were identified that could be replaced with `require` statements:

- The `if (totalShares == 0) revert NoShares()` statement
- The `if (totalShares + _amount > MAX_SHARES) revert InsufficientShares()` statement
- The `if (pendingRevocationTime == 0) revert NotPendingRevocation()` statement

For conciseness and gas savings, consider replacing `if-revert` statements with `require` statements.

Update: *Resolved in pull request #65 at commit 55d44e6.*

N-05 Mismatched Solidity Versions

The `IUniVesting.sol` file is using Solidity version `^0.8.0`, whereas the rest of the contracts use `^0.8.29`.

Consider updating the pragma in the `IUniVesting` interface to be consistent with the other contracts in the project.

Update: *Resolved in pull request #65 at commit 30a6eed.*

N-06 Rounding Error in `mint` Leaves Undistributed UNI Permanently Locked

The calculation of the UNI amount to be distributed to each recipient rounds down, causing dust amounts to be locked in the `UNIMinter` contract. Over successive years, the contract can silently accumulate an ever-growing dust balance, breaking the promise to distribute the full, freshly minted supply proportionally to shareholders and creating an accounting mismatch between UNI supply and distributed tokens.

Consider allowing the owner to retrieve the accumulated dust amounts from the `UNIMinter` contract.

Update: *Acknowledged, not resolved. The Uniswap Labs team stated:*

> *We believe the dust amounts to be immaterial even on very long time horizons.*

N-07 Unbounded Loop Can Run Out of Gas

The `mint` function of the `UNIMinter` contract performs a `for` loop to iterate through all the share allocations. If the number of allocations is high enough, such an operation can consume excessive gas, eventually making any attempt to `mint` to revert.

Consider limiting the number of possible allocations to avoid encountering the aforementioned edge case.

Update: *Acknowledged, not resolved. The Uniswap Labs team stated:*

> *We will not fix; we do not expect the owner to have many allocations leading to OOG. In the event that allocations do OOG, the owner can revoke shares to ensure minting is not permanently DOS'd.*

N-08 Inconsistent Contract Naming

There is an inconsistency in how contracts are named within the codebase. Specifically the `UNIMinter` contract uses upper case format for UNI token, whereas the `UniVesting` contract does not.

Consider using a consistent contract naming approach to improve the overall clarity and readability of the codebase.

Update: *Resolved in pull request #65 at commit 5bf532e.*

Conclusion

The `UniMinter` and `UniVesting` contracts expand the Phoenix Fees system to allow for managing the minting of new UNI tokens as well as their subsequent distribution.

During the audit, two critical-severity issues were identified. One of these allowed a malicious actor to reduce the shares of a user to 0 if they are scheduled to be revoked within the next minting period. The other issue was related to recipients receiving fewer UNI tokens than they should if all the shares were not allocated. Apart from that, several general recommendations aimed at improving the clarity and maintainability of the codebase were also made.

The Uniswap Labs team is appreciated for being very helpful during the course of the review and for timely responding to all the questions posed by the audit team.