OpenZeppelin | security

Uniswap Labs Phoenix Fees Audit

September 29, 2025

Table of Contents

Summary

Type	DeFi
Timeline	From 2025-09-05 To 2025-09-10
Languages	Solidity

Total Issues	15 (11 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	1 (1 resolved)
Medium Severity Issues	1 (1 resolved)
Low Severity Issues	1 (0 resolved)
Notes & Additional Information	12 (9 resolved)

Scope

OpenZeppelin performed an audit of the Uniswap/phoenix-fees repository at commit 146d367.

In scope were the following files:

```
src
├── AssetSink.sol
├── Deployer.sol
├── base
│   ├── Nonce.sol
│   └── ResourceManager.sol
├── feeControllers
│   └── V3FeeController.sol
├── interfaces
│   ├── base
│       ├── INonce.sol
│       └── IResourceManager.sol
│   ├── IAssetSink.sol
│   ├── IFirepitDestination.sol
│   ├── IReleaser.sol
│   ├── IUniswapV3FactoryOwnerActions.sol
│   ├── IUniswapV3PoolOwnerActions.sol
│   └── IV3FeeController.sol
└── releasers
    ├── ExchangeReleaser.sol
    └── Firepit.sol
```

System Overview

The Phoenix Fees system is a set of smart contracts designed to manage, collect, and release protocol fees from different Uniswap versions and pools. The architecture separates the fee lifecycle into distinct, modular components: a central controller that governs the fee policies for specific Uniswap versions, a passive sink where fees accumulate, and a releaser mechanism that gates the withdrawal of those fees.

A `Deployer` contract handles the initial setup, deploying the core contracts, linking them together, and transferring ownership to a designated trusted owner that is intended to be the owner of the Uniswap V3 Factory.

`AssetSink`

The `AssetSink` contract serves as a simple, stateful vault for holding protocol fees collected from Uniswap pools. It is designed to passively accumulate assets without complex internal logic.

- Its primary function is to hold the ERC-20 tokens and native currencies that are sent to it.
- Withdrawals are restricted to a single, authorized `releaser` address.
- The contract owner holds the exclusive privilege to set or change the `releaser` address, giving them ultimate control over who can initiate the fee release process.

`V3FeeController`

The `V3FeeController` contract is the administrative core of the system. It handles fees for the Uniswap V3 protocol and is intended to be set as the owner of the Uniswap V3 Factory. It manages all the fee-related parameters for V3 pools, including enabling new fee tiers and setting protocol fee rates. Specifically, the `V3FeeController` has the following characteristics:

- It provides functions to collect the accrued protocol fees from any V3 pool and deposit them directly into the configured `AssetSink` contract.
- The contract owner can enable new fee tiers in the Uniswap V3 Factory.

- A privileged `feeSetter` role is responsible for setting the default fee rates and, most critically, for setting a Merkle root. This root acts as an allowlist, defining which pools are eligible for automated fee updates.
- The system allows any external actor to trigger a fee update on a pool by supplying a valid Merkle proof, effectively decentralizing the execution of fee changes while keeping the policy-setting centralized.

Update: *The team changed the way in which fees are updated in pools by grouping by pairs instead of individual pools. This is to make more efficient the process of updating several pools at once by improving the merkle tree. These changes have been introduced in pull request #50.*

Releasers (`Firepit` and `ExchangeReleaser`)

The `ExchangeReleaser` is an abstract contract that defines a mechanism for withdrawing fees from the `AssetSink` contract. It acts as a gatekeeper, requiring callers to pay a fee in a designated `RESOURCE` token before it will authorize the `AssetSink` to release its funds. The `Firepit` contract is a concrete implementation of this model.

- The release process uses a nonce to prevent race conditions.
- To trigger a release, a caller must transfer a `threshold` amount of the `RESOURCE` token. In the `Firepit` implementation, these tokens are burned by sending them to the `0xdead` address.
- The `Firepit` contract also imposes a limit on the number of tokens that can be released in a single transaction.
- Privileged roles control the economics of this mechanism: an `owner` can appoint a `thresholdSetter`, who in turn can adjust the `threshold` amount required for a release.

Design Review - MEV implications

The system relies on a fixed `threshold` of UNI tokens to be spent by searchers in order to withdraw the accumulated fees in the sink contract, independently of the amount of UNI tokens.

Complex swaps, where multiple pools are hit with large-volume swaps, can increase the protocol fees by a significant amount. This allows searchers to bundle large swaps with a fee release in order to underpay for the fee tokens, causing a loss of revenue for the UNI token holders. Furthermore, it can be expected that searchers will not just bundle a single swap

transaction but will bundle as many as possible since each transaction increases the protocol fees. With Uniswap being the market leading DEX, complex, large-volume swaps happen regularly, and will likely keep increasing as the overall volume of Uniswap increases

Analyzing the last 100,000 blocks and aggregating the total swap fees generated per block, it can be seen that about 15% of these blocks generate more than $10,000 in swap fees which would demonstrate a significant bump in protocol fees to potentially be incorporated in the fee release. However, the applicability of this scenario is not straightforward, as large-volume swaps would need to coincide with the moment where accumulated fees are near the threshold. Given the current volumes, this alignment is not likely to occur on a regular basis.

Overall, the current design does not maximize the value increase for UNI holders as it might leak value on rare occasions. Instead, an auction system where the fee amount is static or an oracle based system for a dynamic `threshold` parameter should be able to maximize the value at a higher rate. We encourage the Uniswap team in reviewing their design choices in light of this analysis.

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

> - *It is something we've discussed internally, and value loss is not constrained to large swaps (as noted in the report). We theorized potential value loss when UNI sharply declines in value or when the assets sharply increase in value.*
> - *We have considered a dynamic threshold, but for the initial version we believe a semi-static value will be more predictable for searchers.*
> - *In the Dune query, you identify that 15% of blocks produce $10,000 of fee revenue for LPs. In the worse case, the protocol fee can only capture a 4th of that ($2,500):*
> - *Our current expectation is to set the burn threshold at ~100,000 USD.*
> - *Because there is a 15% chance the protocol leaks $2,500 on $100K, the predicted value loss is 0.375% (15% of 2500/100K) which is sort of within our expectations.*
> - *Assuming protocol fee accrual is linear/predictable, the best searchers are the one that can forecast inflow of protocol fees to capture the reward before other participants.*
>
> *We mostly agree though, value leakage is something we intend to monitor very closely on launch!*

Security Model and Trust Assumptions

During the audit, the following special observations and trust assumptions were made:

- The `owner` controls multiple critical parameters across multiple contracts. It can:
  - change the `releaser` on the `AssetSink` contract, allowing them to redirect all future fee claims
  - appoint the `feeSetter` on the `V3FeeController` contract, controlling who defines fee policies
  - appoint the `thresholdSetter` on the `Releaser` contract, controlling the cost of releasing fees
  - enable new fee tiers on the Uniswap V3 factory
- The `feeSetter` role controls the Merkle root for fee updates in `V3FeeController`.
- The `thresholdSetter` role controls the economic cost of releasing fees from the `AssetSink` contract, and could set it to an arbitrary value.

High Severity

H-01 Factory Ownership is Permanently Locked

The `V3FeeController` contract is designed to be the owner of the Uniswap V3 Factory, allowing it to manage fee-related permissions such as enabling new fee tiers. The controller itself is owned and managed by a separate address. This ownership delegation pattern allows the controller to perform its specific administrative duties without holding broader owner privileges.

The `V3FeeController` contract does not expose a function to call the `setOwner` method of the Uniswap V3 factory. Once the ownership of the factory is transferred to the `V3FeeController` contract, there is no mechanism to ever transfer it to another address. This permanently locks the factory's ownership to this specific `V3FeeController` instance, removing the ability to upgrade the controller or recover from a critical issue by transferring ownership.

To ensure long-term upgradeability and administrative flexibility, consider adding a new, owner-restricted function to the `V3FeeController` contract. This function should allow the owner of the `V3FeeController` contract to call the `setOwner` function of the `FACTORY` contract, enabling them to transfer factory ownership to a new address when necessary.

Update: *Resolved in pull request #43.*

Medium Severity

M-01 Incorrect Leaf Hashing for the `MerkleProof` Library

The `V3FeeController` contract uses Merkle trees to specify the pools that will enable the protocol fee. However, depending on the library used for the Merkle tree creation, the provided proofs will not be usable. The contract uses OpenZeppelin's `MerkleProof` library to verify proofs on-chain. Its counterpart, the merkle-tree JS library creates trees by hashing leaves twice. The `V3FeeController` contract instead hashes the leaves only once which will

prevent the proofs from being successfully verified. Specifically, this happens in the `triggerFeeUpdate` and `batchTriggerFeeUpdate` functions of the `V3FeeController` contract.

Consider updating the `V3FeeController` contract to hash each leaf twice.

Update: *Resolved in pull request #42.*

Low Severity

L-01 Users Can Overpay if `threshold` Changes

Whenever the releaser's fee setter adjusts the `threshold`, the price to release the fees will not match the expected amount of a user who has already submitted their transaction by calling the `release` function. Therefore, it is possible that the user ends up overpaying if the threshold is set higher than before.

Consider increasing the nonce when the `threshold` is adjusted so that any existing `release` transactions in the mempool are invalidated.

Update: *Acknowledged, not resolved. An important consideration that was out of scope for the audit is that the nonce needs cross-chain synchronization, and this invalidates the proposed solution. The Uniswap Labs team has improved the docstrings in pull request #52 to raise awareness and improve clarity.*

Notes & Additional Information

N-01 Use `calldata` Instead of `memory`

When dealing with the parameters of `external` functions, it is more gas-efficient to read their arguments directly from `calldata` instead of storing them to `memory`. `calldata` is a read-only region of data that contains the arguments of incoming `external` function calls. This makes using `calldata` as the data location for such parameters cheaper and more efficient

compared to `memory`. Thus, using `calldata` in such situations will generally save gas and improve the performance of a smart contract.

Throughout the codebase, two instances where function parameters should use `calldata` instead of `memory` were identified:

- In the `ExchangeReleaser` contract, the `assets` parameter
- In the `Firepit` contract, the `assets` parameter

Consider using `calldata` as the data location for the parameters of `external` functions to optimize gas usage.

Update: *Resolved in pull request #44.*

N-02 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.

Throughout the codebase, multiple instances of `if-revert` statements were identified that could be replaced with `require` statements:

- In line 18 of the `AssetSink` contract
- In line 26 of the `ResourceManager` contract
- In lines 36, 76, 83 and 106 of the `V3FeeController` contract
- In line 23 of the `Firepit` contract

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

Update: *Resolved in pull request #45.*

N-03 Functions Updating State Without Event Emissions

Throughout the codebase, multiple instances of functions updating the state without an event emission were identified:

- The `setReleaser` function of the `AssetSink` contract

- The `setThresholdSetter` and `setThreshold` functions of the `ResourceManager` contract
- The `setMerkleRoot`, `setDefaultFeeByFeeTier`, `triggerFeeUpdate`, `setFeeSetter` and the `_setProtocolFee` functions of the `V3FeeController` contract

Consider emitting events whenever there are state changes to improve the clarity of the codebase and make it less error-prone.

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

> *We do not intend to update the functions to emit events.*

N-04 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, the majority of in-scope contracts (like `AssetSink` or `ResourceManager`) do not have a security contact.

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 #55.*

N-05 Mismatched Solidity Versions

The `V3FeeController.sol` file is using Solidity version `^0.8.0`, whereas the rest of the files use `^0.8.29`.

Consider updating the pragma in the `V3FeeController` contract to match the other contracts in the project.

Update: *Resolved in pull request #46.*

N-06 `FeesClaimed` Event Can Emit an Incorrect Amount

In line 34 of the `AssetSink` contract, inside the `release` loop, the `amount` variable is recorded **before** calling `asset.transfer(recipient, amount)` and then reused in the `FeesClaimed` event. For standard ERC-20 tokens, this is harmless. However, for assets that implement transfer fees, burns, or any deflationary mechanism, the actual tokens delivered to `recipient` will be *less* than `amount`. The event will therefore over-report how many tokens were claimed, misleading off-chain indexers.

Consider using a safer pattern like computing the delta between the contract's balance before and after the call and emitting that figure instead.

Update: *Resolved in pull request #51. The event has been removed, and the off-chain services will now rely on normal transfer events instead.*

N-07 Inconsistent Hashing Implementation

The `V3FeeController` contract implements a gas-efficient, assembly-based `_hash` function to compute the `keccak256` hash of a pool address. This function is intended for generating leaf nodes for Merkle proofs, which are used to verify that a pool is authorized for a fee update. The `V3FeeController` contract uses this hashing logic in both its single and batch fee update functions.

However, an inconsistency exists in how pool addresses are hashed between different functions. The `batchTriggerFeeUpdate` function correctly uses the optimized `_hash` function when preparing the leaves for multi-proof verification. However, the `triggerFeeUpdate` function uses the less gas-efficient `keccak256(abi.encode(pool))` pattern to generate the node for its single-proof verification. This results in higher gas costs for single-fee updates and introduces an unnecessary inconsistency in the codebase.

To improve gas efficiency and ensure consistent implementation, consider replacing the `keccak256(abi.encode(pool))` call in the `triggerFeeUpdate` function with the provided `_hash(pool)` function.

Update: *Resolved in pull request #42.*

N-08 No Range Validation for Packed `defaultFeeValue`

Within the `V3FeeController` contract, the `setDefaultFeeByFeeTier` function stores any `uint8` value without confirming that each 4-bit nibble lies in the allowed range (i.e., 0 or 4-10). An out-of-range value (e.g., `0xFF`) will later make the `_setProtocolFee` call `setFeeProtocol` with invalid denominators, causing Uniswap V3 pools to revert.

Consider adding proper input validation when calling the `setDefaultFeeByTier` function.

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

> *Since a permissioned actor sets the default fees, we expect them to properly set valid values. More so, invalid values cannot be pushed to the pools, so any unacceptable value (from the permissioned actor) is assumed to be a user error.*

N-09 External Call Could Be a Reentrancy Attack Vector

One of the major dangers of calling external contracts is that they can take over the control flow and perform actions not expected by the calling function. The external call in line 33 of `AssetSink.sol` could potentially be a reentrancy attack vector.

Even if this does not currently have a direct impact on the in-scope codebase, in order to prevent future reentrancy attacks, consider avoiding external calls to untrusted contracts by using the OpenZeppelin `nonReentrant` modifier.

Update: *Acknowledged, not resolved.*

N-10 Misleading and Missing Documentation in `AssetSink`

The `AssetSink` contract's documentation contains inaccuracies and omissions. The NatSpec comment for the constructor states that it creates a new `AssetSink` instance with a specified `releaser`, which is incorrect. The constructor takes no arguments and does not set the `releaser`. This is handled later by a call to `setReleaser`.

To improve the clarity and correctness of the code, consider updating the documentation for the `AssetSink` contract.

Update: *Resolved in pull request #47.*

N-11 Unconventional Import Paths

The `ResourceManager` contract utilizes indirect import paths for two of its dependencies. The `ERC20` interface is imported via the `SafeTransferLib.sol` file instead of being imported directly from its own source file within the Solmate library. Similarly, the `IResourceManager` interface is imported from the `IReleaser.sol` file, which itself imports `IResourceManager`. This transitive import obscures the true location of the interface definition.

While these indirect imports do not affect runtime behavior, they reduce code clarity and can be confusing for developers and auditors trying to trace dependencies. To improve code quality and maintainability, consider refactoring these statements to use direct import paths.

Update: *Resolved in pull request #48.*

N-12 Unused Custom Error

The `IV3FeeController` interface defines a custom error named `AmountCollectedTooLow`. The presence and description of this error imply an intention to validate that a sufficient amount of fees has been collected during the `collect` operation.

The `V3FeeController` contract, which implements the `IV3FeeController` interface, never utilizes the `AmountCollectedTooLow` error. The `collect` function executes the fee collection call and returns the collected amounts without ever performing a check against an expected minimum. This leaves a piece of dead code in the interface and creates a discrepancy between the interface's declared behavior and the implementation's actual logic, which could mislead developers or integrators.

To align the implementation with the interface, consider either implementing logic within the `collect` function to validate the collected amounts and revert with `AmountCollectedTooLow` if the check fails, or removing the unused error definition from the `IV3FeeController` interface if this check is not intended to be performed.

Update: *Resolved in pull request #49. The unused error has been removed.*

Conclusion

The Phoenix Fees system has been designed to collect fees from revenue sources and exchange them for a specific amount of a designated token. For now, the Uniswap Labs team uses it to collect protocol fees from Uniswap V3 pools and exchange them for UNI tokens that are consequently burned. In the future, it will be expanded to support both V2 and V4 of the protocol, as well as to allow for collecting revenue from other chains.

During the audit, one high-severity issue related to the system's upgradeability was identified. In addition, an analysis of the MEV implications of the current design was conducted, which has been included in the introduction along with alternative mechanism proposals to mitigate potential UNI value leakage from large-swap bundling paired with fee releases. Apart from this, several general recommendations aimed at improving the clarity and maintainability of the codebase were also made.

The Uniswap 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.