PUBLIC

Code Assessment

of the Franchiser Expiry Smart Contracts

February 11, 2025

Produced for

by

Contents

1 Executive Summary

Dear Uniswap Foundation,

Thank you for trusting us to help Uniswap Foundation with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Franchiser Expiry according to Scope to support you in forming an opinion on their security risks.

Uniswap Foundation's Franchiser system enables multi-level delegation of UNI tokens' voting power. This latest version introduces support for expirations with permissionless recall of funds to the original owner after expiry.

The most critical subjects covered in our audit are asset solvency and front-running resistance. Security regarding all the aforementioned subjects is high.

The general subjects covered are gas efficiency, code complexity, and documentation. The robustness regarding all the aforementioned subjects is satisfactory but can be improved, see Redundant checks in ...Many() functions and Inaccurate NatSpec. Furthermore, this report contains notes highlighting considerations to prevent unexpected behavior during operation.

In summary, we find that the codebase provides a high level of security. No issues were identified that would pose a significant risk to the system.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the `src` folder of the Franchiser Expiry repository:

```
Franchiser.sol
FranchiserExpiryFactory.sol (previously FranchiserFactory.sol in Version 1)
FranchiserLens.sol
base:
    FranchiserImmutableState.sol
interfaces:
    Franchiser:
        IFranchiser.sol
        IFranchiserErrors.sol
        IFranchiserEvents.sol
    FranchiserFactory:
        IFranchiserExpiryFactory.sol (previously IFranchiserFactory.sol in Version 1)
        IFranchiserExpiryFactoryErrors.sol (previously IFranchiserFactoryErrors.sol in Version 1)
    IFranchiserImmutableState.sol
    IFranchiserLens.sol
    IVotingToken.sol
```

This report covers Version 2, where the newly added expire functionality was the primary scope of review. The assessment also ensured that the integration of this functionality does not introduce issues in the existing system.

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	07 September 2024	a9cd24d12ec2c390807a148d9b07ee6e2728aa05	Initial Version
2	03 February 2025	0c4c61844511d03d3fd0e97afdf798b04da4cb76	Franchiser Expiry

For the solidity smart contracts, the compiler version `0.8.15` was chosen.

2.1.1 Excluded from scope

Imported dependencies are not in the scope of this assessment.

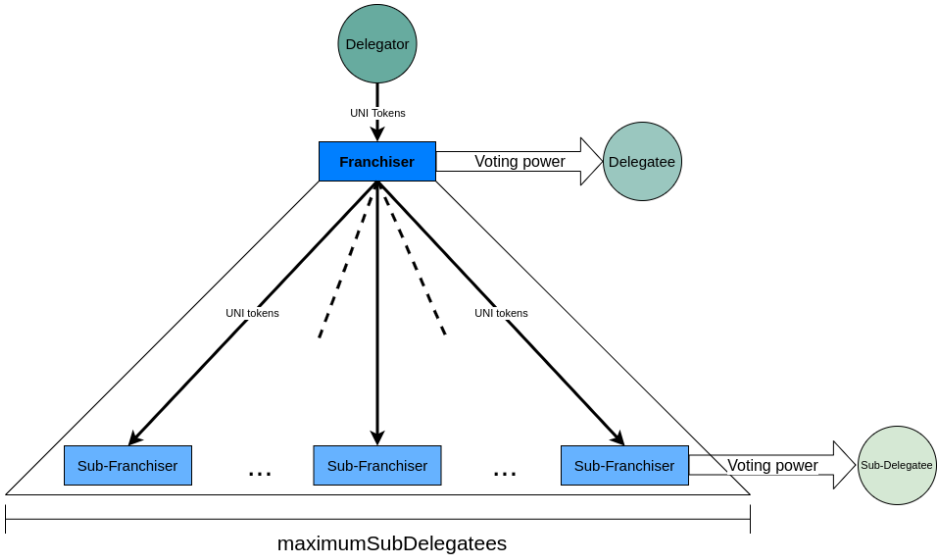
2.2 System Overview

This system overview describes Franchiser Expiry corresponding to Version 2 of the contracts as defined in the Assessment Overview. Version 1 of Franchiser is the base implementation, without the new expiry functionality added in Version 2.

2.2.1 High-level Overview

Uniswap Foundation offers a Franchiser system that enables a token holder to delegate voting power to a delegatee, who can further delegate to subDelegatees. The initial token holder can retrieve tokens at any time. Additionally, once the expiration time of a delegation is reached, anyone can permissionlessly trigger the recall of delegated tokens to the original owner (the delegator).

The contract is intended to be used with the UNI token on Ethereum mainnet.



Franchiser contract visualization

A single Franchiser contract can be visualized as shown in the figure above.

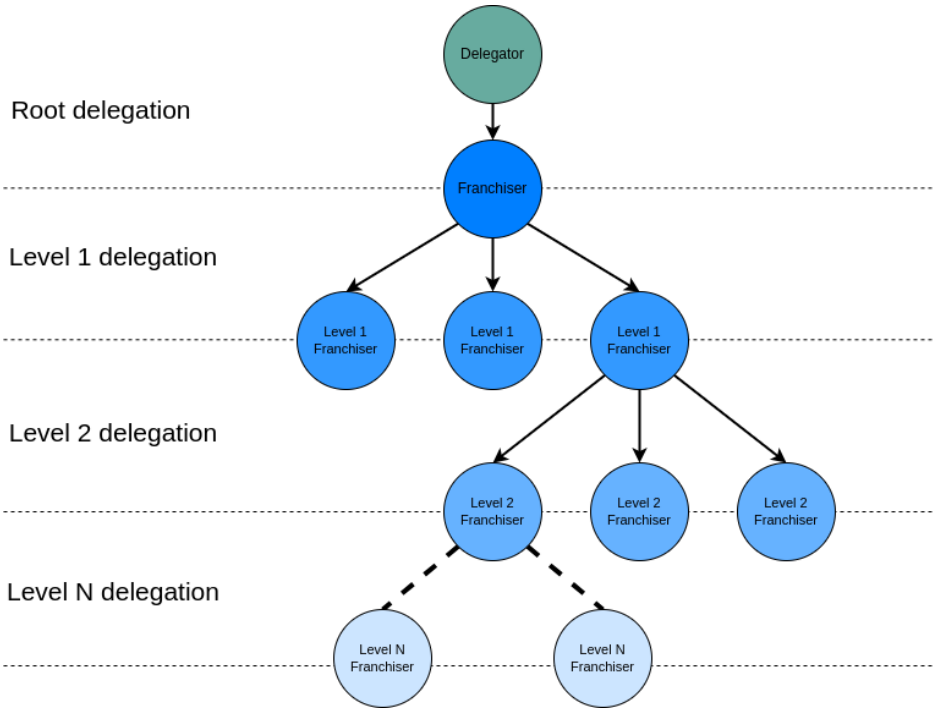
The `Delegator` is the token owner who supplies UNI tokens to the `Franchiser` contract.

The `Delegatee` is the address that receives the voting power from the tokens held by the `Franchiser` contract. The `Delegatee` can choose to further sub-delegate this voting power to multiple `subDelegatees`. For each sub-delegation, the delegatee's `Franchiser` contract deploys a new `Franchiser` contract (Sub-Franchiser) and transfers the corresponding UNI tokens.

The per-address aggregation of delegated voting tokens happens directly in the UNI token contract. Voting mechanics are also handled by the UNI token contract, independently of the Franchiser system.

At any time, the Delegator can retrieve tokens from the Franchiser contract and all its sub-delegations. In this context, the `Franchiser` contract acts as the `Delegator` for its `Sub-Franchiser` contracts. After expiry, anyone can trigger the retrieval of tokens for the Delegator of the root-level Franchiser. Expiration times are tracked by the factory for the root-level Franchiser, Sub-Franchiser contracts do not have their own expiration times.

By layering multiple `Franchiser` contracts, a tree-like delegation structure can be created.



Delegations

2.2.2 Nested Delegation Structure

To ensure that the token `recall` operation does not exceed the gas limit, each `Franchiser` contract has a limited number of subDelegatees, defined by `maximumSubDelegatees`.

The root `Franchiser` contract has a maximum of 8 subDelegatees. Each subsequent level of subDelegatees can have at most half the maximum number of subDelegatees of the previous level. This is controlled by the `INITIAL_MAXIMUM_SUBDELEGATEES` and `DECAY_FACTOR` constants, which by default are set to 8 and 2, respectively. With these default parameters, the maximum depth of delegation is 5 levels (including the root level).

For a single initial delegation, the number of `Franchiser` contracts for each level in the delegation tree can be:

- 1 `Franchiser` contract for the initial token owner
- 8 `Franchiser` contracts for the first level of delegation
- 8 * 4 = 32 `Franchiser` contracts for the second level of delegation
- 8 * 4 * 2 = 64 `Franchiser` contracts for the third level of delegation
- 8 * 4 * 2 * 1 = 64 `Franchiser` contracts for the fourth level of delegation

This results in a maximum of 169 `Franchiser` contracts for one initial delegation.

2.2.3 Smart Contracts Overview

The system consists of the following contracts:

1. `FranchiserExpiryFactory`
2. `Franchiser`
3. `FranchiserLens`

2.2.3.1 `FranchiserExpiryFactory`

The factory contract facilitates the creation and management of Franchiser contracts.

A constant variable, `franchiserImplementation` stores the address of the `Franchiser` contract reference implementation.

Whenever a user calls `FranchiserExpiryFactory.fund()`, the `FranchiserExpiryFactory` deploys a minimal proxy (ERC-1167; OZ: `cloneDeterministic`) of the `Franchiser` contract implementation, unless a contract has already been deployed for that specific user (delegator) and delegatee pair. The `fund()` function then transfers UNI tokens to the newly created `Franchiser` contract and updates the expiry timestamp.

The `INITIAL_MAXIMUM_SUBDELEGATEES` constant is used to set the maximum number of subDelegatees for the root-level `Franchiser`.

The root-level delegator can call the `recall()` function of the `FranchiserExpiryFactory` contract at any time to retrieve the tokens held by the first-level `Franchiser` contract and all of its sub-delegations. Further `recallExpired()` allows anyone to permissionlessly initiate a recall if the expiration for this delegation has been reached. Note that when `recall()` and `recallExpired()` is called with a `Franchiser` contract that does not exist the function returns without effect and does not revert.

For the `fund()`, `recall()` and `recallExpired()` functions multi-action versions are available, allowing multiple actions to be performed in a single transaction: `fundMany()`, `recallMany()` and `recallExpired()`. Note that `fundMany()` takes a single expiration timestamp as a parameter. To fund receivers with different expirations, separate calls are required.

Since the `FranchiserExpiryFactory` needs to transfer UNI tokens from the user to the newly created `Franchiser` contract, the user must approve the transfer of UNI tokens to the `FranchiserExpiryFactory` contract.

The user can call the `Uni.approve()` function to approve the transfer of UNI tokens to the `FranchiserExpiryFactory` contract. Alternatively, to bundle approval and funding in a single transaction, the user can produce a signature for the `Uni.permit()` function and utilize the following functions of the `FranchiserExpiryFactory` contract:

- `permit()`
- `permitAndFund()`
- `permitAndFundMany()`

2.2.3.2 Franchiser

The main functionality of the `Franchiser` contract is provided by the following functions:

- `subDelegate()` and `subDelegateMany()` - Delegates voting power to one or more subDelegatees. When these functions are called, an ERC-1167 minimal proxy of the `Franchiser` contract is deployed (if it does not already exist) and the specified amount of UNI tokens is transferred to it. Each subDelegatees's `Franchiser` contract is owned by the parent `Franchiser` contract and is initialized with `maximumSubDelegatees` set to the parent's `maximumSubDelegatees` divided by the `DECAY_FACTOR`. The addresses of the Franchiser's active subDelegatees are stored in the `_subDelegatees` set.
- `unSubDelegate()` and `unSubDelegateMany()` - Removes one or more subDelegatees from the `_subDelegatees` set and recalls the UNI tokens back to the parent `Franchiser` contract.
- `recall()` - Recalls the UNI tokens from the `Franchiser` contracts of the `_subDelegatees` set and transfers all of the tokens back to the `delegator`.

2.2.3.3 FranchiserLens

The `FranchiserLens` contract is a read-only utility contract that provides a way to retrieve data from the `Franchiser` contract. It is intended to be used by frontend applications and websites.

It has the following functions:

- `getRootDelegation()` - Retrieves information about the root delegation.
- `getVerticalDelegations(Franchiser)` - Traverses the delegation graph up to the root delegation and returns a list of all delegations along the path.
- `getHorizontalDelegations(Franchiser)` - Returns a list of all sub-delegations of the given `Franchiser` contract.
- `getVotes()` - Returns the amount of votes for a given `Franchiser`, along with delegation information.
- `getAllDelegations()` - Returns the entire delegation tree as a list of lists.

FranchiserLens does not provide expiration information which must be read from the FranchiserExpiryFactory directly.

2.2.4 Roles and Trust Model

The system does not have explicit role-based access control. No addresses can change the system parameters in a way that would affect the behavior of all `Franchiser` contracts.

Each `Franchiser` contract has an `owner`. Only the `owner` can call the `recall()` function to retrieve the tokens from the contract and its sub-delegations at any time. Permissionless recall of the funds is possible using `recallExpiry()` once the expiration timestamp has been reached.

For the root level `Franchiser`, the owner is the `FranchiserExpiryFactory` contract. For subDelegatee `Franchiser` contracts, the owner is the parent `Franchiser` contract.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe our findings. The findings are split into these different categories:

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

6 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

6.1 Dangling Franchisers With Tokens

Informational **Version 1**

CS-UNIFND-FRNC-001

Assume the following delegation chain:

Alice -> Bob -> Charlie.

Alice decides to send 100 tokens to Charlie, effectively giving more voting power to Charlie, while still keeping control over the tokens. Bob can front-run this transaction and `unSubDelegate()` Charlie. When Alice's transaction is processed, Charlie's Franchiser will get 100 tokens, but Alice will not be able to recall them anymore unless Bob calls `subDelegate()` on Charlie again.

Sending tokens to a Franchiser's address directly should be avoided.

6.2 Inaccurate NatSpec

Informational **Version 1**

CS-UNIFND-FRNC-002

Some of the NatSpec comments in the code can be improved:

1. The `@notice` of the function `IFranchiserFactory.permitAndFundMany` describes that the function calls `permitAndFund` many times. However, the implementation makes no calls to `permitAndFund`. Instead, it calls `permit` once on the sum of amounts and then calls `fund` many times.
2. The `@dev` of the function `IFranchiser.subDelegate` is inaccurate. If the Franchiser associated with the `subDelegatee` is already active the `amount` of `votingTokens` will also be delegated to `subDelegatee`.

6.3 Redundant Checks in ...Many() Functions

Informational **Version 1**

CS-UNIFND-FRNC-003

`Franchiser.subDelegateMany()` calls `subDelegate()` multiple times. The `subDelegate()` function has the `onlyDelegatee` modifier which checks if the caller is the delegatee. Calling the `onlyDelegatee` modifier multiple times in a loop is a redundant check.

In Version 2, `FranchiserExpiryFactory.fundMany()` uses a fixed expiration time for all delegations. The batch is processed iteratively using `fund()`, which verifies the expiration timestamp. Since the timestamp is the same for the entire batch, these repeated checks are redundant.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Expiration Time Is Overwritten on Fund

Note Version 2

In the functions `fund` and `fundMany`, a delegation expiration time is stored for each franchiser in the mapping `expirations`. Once the expiration time is reached, delegated tokens can be recalled.

Users must be aware that any prior delegation expiration time is overwritten with any new call to `fund` or `fundMany`, an expiry timestamp should not be considered immutable.

7.2 Later Solidity Compiler Version Can Be Used to Avoid Unchecked Blocks

Note Version 1

In version 0.8.22, the Solidity compiler does not use safe math checks for loop increments by default. If the project were to be updated to use this version, the unchecked block would no longer be necessary. Currently, 0.8.15 is used.

More info: https://soliditylang.org/blog/2023/10/25/solidity-0.8.22-release-announcement/

7.3 Notes for Delegating Smart Contracts

Note Version 2

In Version 2 of the project delegation with expiry has been added. After the expiration timestamp of a delegation is reached anyone can recall tokens from the top level Franchiser contract by calling function `recallExpired` or `recallManyExpired` in the `FranchiserExpiryFactory` contract. These functions will delete delegations and transfer all delegated tokens back to the delegator. The owner must be able to handle the incoming funds, which is a reasonable assumption since the funds originally came from them.

If the delegator is a smart contract, it must be able to accept token transfers initiated by other parties and handle them gracefully. For example, it must not rely on internal accounting that separately tracks the balance. Further, if it contains logic to retrieve the funds itself, it must be able to handle the scenario when funds arrive separately from `recallExpired()`.

7.4 Theoretical Hash Collision Attack

Note Version 1

Sub-delegatees have the ability to deploy new Franchiser contracts via sub-delegation. The address of the new Franchiser can be arbitrarily chosen by the sub-delegatee.

Assume the following scenario:

Alice is the Delegator, that delegates tokens to Bob. Bob deploys a Factory contract, that can create an arbitrary number of contracts that are able to sweep UNI tokens. Let's call these "Sweeper" contracts. Bob can as well pre-compute the set of sub-Franchiser addresses that will be used to deploy Franchiser contracts (within Alice's delegation tree). Let's call this the "Franchiser" set of contracts.

If both sets of addresses are large enough, there is a chance that some of the sub-Franchiser address will collide with the Sweeper addresses.

In this case, Bob can:

1. Deploy a Sweeper contract
2. `subDelegate()` to the Sweeper contract, so Alice's delegated tokens are transferred to the Sweeper contract. Due to address collision, the Franchiser contract creation will be skipped.
3. Sweep the tokens from the Sweeper contract.

This is a highly theoretical attack, as it requires a lot of resources to pre-compute the addresses. A 50% chance of address collision is reached after 2^81 addresses (2^80 in each set). With the current cost of hardware, such an attack is in the order of 400 billion USD.

However, in ~50 years, this attack might become feasible.