PUBLIC

Code Assessment

of the veYFI and RewardPool Smart Contracts

November 22, 2022

Produced for

yearn.finance

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Yearn Team,

Thank you for trusting us to help Yearn with this security audit. Our executive summary provides an overview of subjects covered in our review of the latest reviewed contracts of veYFI and RewardPool according to Scope to support you in forming an opinion on their security risks.

Yearn implements a voting escrow contract and a reward distribution contract based on Curve's implementation. However, some features were added including the support of locks longer than four years and early withdrawals a with penalty. All admin features present in Curve's implementation were removed.

The communication with Yearn's team was professional, but the delivered code had an unusually high ratio of severe issues. The custom code parts have multiple critical issues and the code base itself was delivered in a state not ready for a proper review. The reasons are (1) issues to compile the code, (2) missing test cases, (3) very limited documentation and specification, (4) obvious issues like function calls to non-existing functions of the other contract. For this reason, the review had to make implicit assumptions about how the code is supposed to work.

The most severe subjects covered in our review were:

1. Missing slope change at the end of locks with a kink described in End Slope Changes Not Set
2. Incorrect calls from `RewardPool` to `veYFI` described in Incorrect Interface Definition and Calls to veYFI in RewardPool
3. Incorrect voting power calculation due to multiple issues described in getPriorVotes Does Not Replay Slope Changes, Kink timestamps are too early and RewardPool Calculates Incorrect Balances

The severity of all the issues mentioned above is high or critical and, again, highlights that the code was not ready to review when submitted.

In the current version, all issues have been fixed or acknowledged. Given the high amount of issues, the likelihood of remaining issues in the code base is higher than usual and we would recommend to take additional steps to ensure the security of the project. 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.

Overall, we rate the security of the current code base good.

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	5
• Code Corrected	4
• Specification Changed	1
High-Severity Findings	3
• Code Corrected	3
Medium-Severity Findings	1
• Code Corrected	1
Low-Severity Findings	10
• Code Corrected	6
• Specification Changed	1
• Code Partially Corrected	1
• Acknowledged	2

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 `VotingYFI.vy` and `RewardPool.vy` source code files inside the repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	20 June 2022	fbda9ac523252920bf3295557f9f764725a23f41	Initial Version
2	28 June 2022	696bb76be86a601f25cda577bfb9dc14daa91079	Version 2
3	01 Sep 2022	1ac8c33bdb76dd541a1d80056bd40181cc2b72f6	Version 3
4	14 Nov 2022	bb9d8ac9dd90a9a9772b9663ce4fa232fda7bce2	Version 4

For the vyper smart contracts, the compiler version `0.3.7` was chosen.

2.1.1 Excluded from scope

All other contracts in the repository are out of scope except for `VotingYFI.vy` and `RewardPool.vy`.

2.2 System Overview

This system overview describes Version 2 of the contracts, as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Yearn implements veYFI, a locking system for its YFI governance token. Users can lock up their YFI tokens and initially receive veYFI proportionally to how long they locked, reaching the maximum at 4 years. For example, with a lock that is 4 years or longer, they will receive exactly 1 veYFI per YFI, with a 1-year lock they will receive 0.25 veYFI per YFI. The initial balance decreases linearly with time until it should be zero at the specified lock end time stamp.

The system is based on Curve Finance's veCurve and FeeDistributor contracts.

Balances are tracked using checkpoints, in which a `Point` is created. A `Point` contains a `bias` and a `slope`. The `bias` is the balance at the time of the `Point`. The `slope` is the amount that the balance decays every second. To calculate the balance at a certain time, the last `Point` is taken and then the `slope` is multiplied by the number of seconds that have passed. This is subtracted from the bias to receive the current balance.

There are 2 cases where the `slope` of a lock will change over time:

1. When a lock expires, the `slope` will become zero.
2. If a lock was previously locked for more than 4 years and crosses the 4-year threshold, its `slope` will go from zero to a positive value, which will cause its balance to start decaying. This is called a "kink" in Yearn's implementation.

The `slope_changes` are saved in a mapping and can only occur at the end of a week. Whenever the `checkpoint` function is called, the `slope_changes` of all weeks that have passed are applied to the global `Point`, which keeps track of the total supply of veYFI.

2.2.1 Differences to Curve

veYFI differs from veCRV in the following ways:

1. Users may lock for longer than 4 years. They will still receive at most 1 veYFI per YFI.
2. A user can decrease the lock duration but only if it's longer than 4 years and to no less than 4 years.
3. Nobody can create a lock for another user.
4. A user can withdraw YFI before their lock has expired, suffering a penalty.
5. The penalty is a linear function of remaining lock time, capped at 75%, so it's a constant 75% penalty from 3 to 4 years remaining.
6. Penalties are sent to the Reward Pool and queued using the `burn` call.
7. The `balanceOfAt` function has been replaced with `getPriorVotes` in order to be compatible with GovernorAlpha.
8. The restriction that only whitelisted smart contracts can lock tokens has been removed. This means it is possible to permissionlessly wrap veYFI.

RewardPool differs from Curve's FeeDistributor in the following ways:

1. All admin roles, related access controls, and functions were removed.
2. The `claim_many` function was removed.
3. Users can relock their tokens when claiming.
4. Users can give another address permission to relock their tokens when claiming for them.

2.2.2 veYFI

The voting escrow token contract has 3 functions which are externally callable and modify state:

1. `checkpoint` - To record the latest global bias and slope, given the latest user interaction and recorded slope changes between the last checkpoint and this one.
2. `modify_lock` - Callable by a user to lock tokens until a specified time and receive voting power in veYFI, or to change a locked amount or time (within some restrictions).
3. `withdraw` - Withdraw the locked YFI early, before the actual unlock time, by paying a penalty.

2.2.3 Reward Pool

The reward pool contract has five external state-changing functions:

1. `burn` - Called by a contract to send in new YFI rewards, e.g. the voting escrow contract.
2. `claim` - Used by veYFI token holders to claim their rewards or claim on behalf of a veYFI holder.
3. `toggle_allowed_to_relock` - If someone else is going to claim on behalf of a veYFI holder, this holder can allow the claiming account to relock their rewards by calling this function first.
4. `checkpoint_token` - Tracks incoming rewards and distributes them between weeks.
5. `checkpoint_total_supply` - This function checkpoints the total supply of veYFI.

2.2.4 Trust assumptions

The contract has no permissioned roles. We assume that only the current implementation of `VotingYFI` is used as `VEYFI` in `RewardPool`. We assume that only the implementation of Yearn's `YFI` token currently deployed to Ethereum mainnet is used as `YFI` in `RewardPool` and in `VotingYFI`.

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 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

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	3

5.1 Unspecified Behavior in Balance Functions

Design Low Version 2 Code Partially Corrected

The global points, which keep track of the sum of user balances in `VotingYFI` are saved in `point_history[self]`. This means that any function used to access user balances can also be used to access global balances by passing the address of the `veYFI` contract as user address.

This leads to the following unspecified behavior, which is not present in Version 1

- `balanceOf(veYFI,ts)` returns the total supply of `veYFI` at timestamp `ts`.
- `getPriorVotes(veYFI,height)` returns the total supply of `veYFI` at block `height`.

This behavior is equivalent to the behavior of `totalSupply` and `totalSupplyAt`.

This makes `totalSupply` and `totalSupplyAt` redundant. They are, however, more gas efficient.

Code partially corrected

The code has been partially corrected to remove redundancy.

The `_balanceOf` function is now internal and is called by both `balanceOf(user,ts)` and `totalSupply(ts)`, removing the duplicated code in `totalSupply`.

The NatSpec of the new `balanceOf(user,ts)` states "Get the current voting power for `user`". However, it is not the current voting power but the voting power at the time `ts` if the argument `ts` is supplied.

The NatSpec of `getPriorVotes` has been adjusted to clarify that `user` can be self to get `totalSupply` at `height`. This is equivalent to the functionality of `totalSupplyAt`.

The `totalSupplyAt` function has not been changed. It contains duplicated functionality, similar to that which was removed from the `totalSupply` function.

5.2 Points Could Be Packed

Design Low Version 1 Acknowledged

The `VotingYFI` and `RewardPool` contracts both use the `Point` struct.

```
struct Point:
bias: int128
slope: int128
ts: uint256
blk: uint256
```

Vyper `0.3.3` does not automatically do tight variable packing for structs, so the struct elements are each stored in a separate storage slot.

By manually packing the variables and reducing the size of `ts` and `blk` to 128-byte values, the `Point` struct could fit into 2 256-byte storage slots instead of 4.

This would represent significant gas savings when loading and storing Points.

Acknowledged

The issue is acknowledged by Yearn.

5.3 Redundant Calculation

Design Low Version 1 Acknowledged

Each time `checkpoint_token` is called in `RewardPool`, the following `assert` is checked:

```
assert block.timestamp > self.last_token_time + TOKEN_CHECKPOINT_DEADLINE
```

In the function `_checkpoint_token`, `self.last_token_time` is loaded again to perform a similar calculation:

```
t: uint256 = self.last_token_time
since_last: uint256 = block.timestamp - t
```

The storage load and the subtraction are always performed twice in the call path.

Acknowledged

Yearn acknowledged the issue.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

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

Critical-Severity Findings	5

- End Slope Changes Not Set Code Corrected
- Incorrect Interface Definition and Calls to veYFI in RewardPool Code Corrected
- Kink Timestamp Can Be Set to a Past Timestamp Specification Changed
- getPriorVotes Does Not Replay Slope Changes Code Corrected
- Incorrect bias and slope for Locks Longer Than 4 Years Code Corrected

High-Severity Findings	3

- Kink Timestamps Are Too Early Code Corrected
- RewardPool Calculates Incorrect Balances Code Corrected
- Reward Pool Not Initialized Code Corrected

Medium-Severity Findings	1

- Rewards Can Become Unclaimable Code Corrected

Low-Severity Findings	7

- Incorrect Natspec Specification Changed
- Long Locks Lead to Incorrect Balances Code Corrected
- Inconsistent Use of last_checkpoint Code Corrected
- Missing Explicit View Decorator in _find_timestamp_epoch Code Corrected
- Redundant Functionality in RewardPool Code Corrected
- Unused Function Argument ve in RewardPool Code Corrected
- Unused State Variable total_received in RewardPool Code Corrected

6.1 End Slope Changes Not Set

Correctness Critical Version 2 Code Corrected

The `slope_changes` that occur at the end of a lock are set in the `_checkpoint_user` function. The `slope_change` is set to the `new_point.slope`.

```
if old_point.slope != 0 and old_lock.end > block.timestamp:
    self.slope_changes[self][old_lock.end] += old_point.slope
    self.slope_changes[user][old_lock.end] += old_point.slope
if new_point.slope != 0 and new_lock.end > block.timestamp:
    self.slope_changes[self][new_lock.end] -= new_point.slope
    self.slope_changes[user][new_lock.end] -= new_point.slope
```

However, if `new_point` has a `lock.end` that is more than 4 years away, the `slope` will be `0`.

```
# the lock is longer than the max duration
   if lock.end > block.timestamp + MAX_LOCK_DURATION:
       point.slope = 0
       point.bias = convert(lock.amount, int128)
```

Hence, for a position that is longer than 4 years, `slope_changes[new_lock.end]` will be set to `0`, meaning the locked position never stops decaying, even once it becomes negative. This will lead to an incorrect global `bias` and `slope`, which will make totalSupply smaller than it should be. The user balance will be `0`, as there is a check that ensures a balance cannot be negative.

This problem does not resolve itself, even if `_checkpoint_user` is called again after crossing the 4 year threshold, as the `old_point` will have a non-zero `slope` at this time and will counteract the `slope_change` of the `new_point`.

Code corrected

`_checkpoint_user` was changed to correctly account for the slope change by adding the slope adjustements at the end of the lock period.

```
# schedule kinks for locks longer than max duration
if old_kink.slope != 0:
    self.slope_changes[self][old_kink.ts] -= old_kink.slope
    self.slope_changes[user][old_kink.ts] -= old_kink.slope
    self.slope_changes[self][old_lock.end] += old_kink.slope
    self.slope_changes[user][old_lock.end] += old_kink.slope
if new_kink.slope != 0:
    self.slope_changes[self][new_kink.ts] += new_kink.slope
    self.slope_changes[user][new_kink.ts] += new_kink.slope
    self.slope_changes[self][new_lock.end] -= new_kink.slope
    self.slope_changes[user][new_lock.end] -= new_kink.slop
```

This is only performed in case a kink needs to be set.

6.2 Incorrect Interface Definition and Calls to `veYFI` in `RewardPool`

Design | Critical | Version 2 | Code Corrected

There are multiple calls to the `veYFI` contract that fail due to incorrect interface definitions and/or incorrect function calls. These are:

The functions `user_point_epoch` and `user_point_history` are part of the `veYFI` interface definition in `RewardPool`. These functions are used in the code base of `RewardPool` multiple times, but the functions do not exist in the current implementation of `veYFI`.

The interface definition for `epoch` is incorrect because `epoch` is a mapping in `veYFI`. The automatically generated getter function needs an `address` to look up and return the value stored in the mapping. Hence, the call to this function in `RewardPool` is also incorrect as it is done without the address argument.

Similarly, `point_history` is defined and used incorrectly. In `veYFI` it is a mapping that maps an `address` and a `uint` input to a `Point`. The interface definition only defines one argument, and the later code only uses one argument.

Consequently, all calls to these functions will fail.

Code corrected

The interface definition in `RewardPool` has been adjusted to correctly reflect the interface of `VotingYFI`.

6.3 Kink Timestamp Can Be Set to a Past Timestamp

Correctness Critical Version 2 Specification Changed

Kink timestamp is set in the `lock_to_kink` function of `VotingYFI`.

```
if lock.amount > 0 and lock.end > self.round_to_week(block.timestamp + MAX_LOCK_DURATION):
        kink.ts = self.round_to_week(lock.end - MAX_LOCK_DURATION)
```

`kink.ts` is rounded down, so it can be set to a past timestamp. This happens for `lock.end` where `lock.end - MAX_LOCK_DURATION` is larger than `block.timestamp`, but still in the same week.

The functions that get a user's balance always start from the most recent user point, so they will never apply slope changes that are in the past. The user's position will never start decaying and stay at the maximum value, unless `_checkpoint_user` is called on that user again.

If `checkpoint` has not been called during the week in which the position is modified, the `slope_change` belonging to the past kink will be applied to the global slope, but not to the user slope. This means global bias will be smaller than the sum of user balances and global slope will be larger than the sum of user slopes. This will lead to a `totalSupply` that is as expected, but inconsistent with user balances.

If `checkpoint` has already been called during the week in which the position is modified, the `slope_change` belonging to the past kink will not be applied to the global slope. This means global slope will be smaller than it should be. This will lead to a `totalSupply` that is larger than it should be, but it is consistent with the sum of user balances.

Specification changed

`MAX_LOCK_DURATION` has been changed to be a multiple of `WEEK`.

This makes it impossible for the conditions of this bug to happen.

6.4 getPriorVotes Does Not Replay Slope Changes

Correctness Critical Version 2 Code Corrected

The `getPriorVotes` function in `VotingYFI` calculates a user's bias based on the last recorded point's slope. It does not call `replay_slope_changes`, which would apply slope changes (kinks) that happened since the last user checkpoint.

As a result, `getPriorVotes` will incorrectly return a voting power that is too high if the user has a kink that is in-between the last user checkpoint and the block height on which `getPriorVotes` is called.

Code corrected

The function `getPriorVotes` was fixed, and calls `replay_slope_changes` to calculate the `upoint` and finally returns `upoint.bias`.

6.5 Incorrect `bias` and `slope` for Locks Longer Than 4 Years

Correctness Critical Version 1 Code Corrected

The `VotingYFI` contract allows users to lock their tokens for an unlimited amount of time. However, their voting power, i.e., `bias`, is capped to four years, even if the user chooses to lock for a longer period. The relevant code is in the `_checkpoint` function:

```
time_left: uint256 = min(new_locked.end - block.timestamp, MAX_LOCK_DURATION)
u_new.bias = u_new.slope * convert(time_left, int128)
```

The capping above is not taken into consideration when the balance of a user needs to be calculated. For illustration, assume a user locks his tokens at time `t0` for 8 years and does not modify his lock for the following `4 years`, i.e., `t1 = t0 + 4 years`. Calling the function `balanceOf` at time `t1` would calculate the balance of the user with the following formula:

```
upoint.bias -= upoint.slope * convert(ts - upoint.ts, int128)
```

The statement above returns `0`, although the user has his tokens locked for the next 4 years and should have the maximum voting power for the tokens locked.

Similarly, the function `balanceOfAt` uses the same formula to calculate the voting power of a user.

The functions `supply_at` and `_checkpoint` have the same issue when calculating the global balance. They also assume that balances start decaying immediately, even if a user has locked for more than 4 years. After 4 years, a user's bias can become negative. At this point, the global bias will also be reduced by the negative amount and the `totalSupply` will no longer be equal to the sum of `balanceOf` over all users.

```
last_point.bias -= last_point.slope * convert(t_i - last_point.ts, int128)
```

The root cause for these issues is that `slope_changes` are only set for the time when a position's balance decay ends, (negative slope change) but the slope change when a balance decay starts is always made right away, even if the lock is longer than 4 years.

Code corrected:

Version 2 introduces "kinks", which are positive slope changes that mark when a position hits the point where it is locked for exactly 4 years and starts decaying. If a position is longer than 4 years, its slope is zero until it hits its kink.

6.6 Kink Timestamps Are Too Early

Correctness High Version 2 Code Corrected

In `VotingYFI`, positions that are locked for more than `MAX_LOCK_DURATION` have a bias of `1 * lock.amount` and a slope of 0. They also have a kink, which sets the slope to a positive value once `MAX_LOCK_DURATION` is crossed.

```
if lock.amount > 0 and lock.end > self.round_to_week(block.timestamp + MAX_LOCK_DURATION):
    kink.ts = self.round_to_week(lock.end - MAX_LOCK_DURATION)
    kink.slope = convert(lock.amount / MAX_LOCK_DURATION, int128)
```

The slope is set to `lock.amount / MAX_LOCK_DURATION`. The timestamp of the kink is rounded to the beginning of the week. However, `MAX_LOCK_DURATION` is not a multiple of weeks (it is 208.57 weeks). So, the time from `kink.ts` to `lock.end` is 209 weeks, but the slope is such that the balance would reach zero after 208.57 weeks. The kink's timestamp is more than `MAX_LOCK_DURATION` away from `lock.end`.

The user's balance will start decaying sooner than it should. During the last 0.43 weeks of the lock, the user's bias would be negative. The global bias is also incorrectly reduced by this amount.

Depending on interpretation of the specification, either the slope is too large, or the kink's timestamp is too early. The current implementation seems to assume that "4 years" means `4*365*86400` seconds, *not* rounded to weeks. Using this interpretation, the kink's timestamp is too early.

As there are checks that set negative biases to zero, the incorrect user balance problem has a medium severity. However, the global bias also becomes wrong, and this is not detected, which leads to the `totalSupply` becoming incorrect and subsequently increasing the severity.

Code corrected

The `MAX_LOCK_DURATION` has been changed to `4 * 365 * 86400 / WEEK * WEEK`, which is exactly 208 weeks. The slope is still set to `lock.amount / MAX_LOCK_DURATION`, which is now correct and will bring the user balance to zero after 208 weeks, which is where the `slope_change` is scheduled.

6.7 RewardPool Calculates Incorrect Balances

Correctness High Version 2 Code Corrected

The `_claim` and `ve_for_at` functions of `RewardPool` calculate veYFI user balances themselves instead of using `veYFI`'s functions for calculating balances. They do this by getting the last `user_point` and then subtracting the `slope` multiplied by time.

```
balance_of: uint256 = convert(max(old_user_point.bias - dt * old_user_point.slope, zero), uint256)
```

This does not apply `slope_changes` and leads to incorrect balances for any lock that has a kink between the last user checkpoint and `_claim` or `ve_for_at` being called.

Code corrected

The `_claim` function has been changed to use `veYFI.balanceOf` instead of calculating balances itself. This fixes the issue, as balanceOf correctly applies `slope_changes`.

The `ve_for_at` function has been removed. `veYFI.balanceOf` can be used instead for the same functionality.

The `_find_timestamp_user_epoch` function has been removed, as it was only used in the two functions above, and was unused after the changes.

6.8 Reward Pool Not Initialized

Correctness | High | Version 1 | Code Corrected

In the `VotingYFI` contract, the `reward_pool` variable is not initialized in `__init__`.

Code corrected:

As of Version 2, the Reward Pool gets correctly initialized in `__init__`

6.9 Rewards Can Become Unclaimable

Design | Medium | Version 1 | Code Corrected

The `_checkpoint_token` function in `RewardPool` keeps track of how many tokens to distribute as rewards every week. It loops through each week that has passed since it was last called and fills `tokens_per_week`. This loop is limited to 20 iterations, so if the last call was more than 20 weeks ago, the most recent weeks will not be filled.

`_checkpoint_token` and `claim` always set `last_token_time` to `block.timestamp`, even if not all weeks up to the current time have been filled. When a user calls `claim`, they will claim for up to 50 weeks by increasing `week_cursor` up to `last_token_time`.

```
for i in range(50):
    if week_cursor >= last_token_time:
        break
    #[...]
    if balance_of > 0:
            to_distribute += balance_of * self.tokens_per_week[week_cursor] / self.ve_supply[week_cursor]

    week_cursor += WEEK
#[...]
self.time_cursor_of[addr] = week_cursor
```

For the weeks that were not filled by `_checkpoint_token`, `tokens_per_week` will be `0` instead of the number of tokens that should be available to claim that week. The user's `to_distribute` will end up smaller than it should be and they will receive fewer rewards than they should. Since `self.time_cursor_of` is updated to `last_token_time`, the contract will not let the user claim the incorrect weeks again, even if they make another call to `claim`. The missed rewards will be stuck in the contract with no way to recover them.

This problem will only occur if `_checkpoint_token` is not called for more than 20 weeks, which is unlikely.

Code corrected

The `_checkpoint_token` loop has been increased to a maximum of 40 iterations.

The same issue is still present, but now after not calling `_checkpoint_token` for 40 weeks instead of 20.

This makes it even more unlikely that the issue will happen in practice.

6.10 Incorrect Natspec

Design | Low | Version 2 | Specification Changed

The NatSpec for the `find_epoch_by_block` and `find_epoch_by_timestamp` functions in `VotingYFI` is:

```
@notice Binary search to estimate timestamp for height number
```

In the case of `find_epoch_by_block` this is incorrect, because the function returns an epoch, not a timestamp, given a block height number.

In the case of `find_epoch_by_timestamp` this is incorrect, because the function returns an epoch, not a timestamp. It does this given a timestamp, not a block height number.

The NatSpec for the `balanceOf` function in `VotingYFI` is:

```
@notice Get the current voting power for `msg.sender`
@param user User wallet address
@param ts Epoch time to return voting power at
@return User voting power
```

The `@notice` is incorrect, because the function returns the voting power for the `user`, not for `msg.sender`.

Specification changed

The NatSpec for all functions was corrected accordingly.

6.11 Long Locks Lead to Incorrect Balances

Correctness Low Version 2 Code Corrected

In order to query a user's balance in `VotingYFI`, the last user `Point` is taken and then `replay_slope_changes` is used to loop through all weeks since that `Point` and calculate the `bias` at a certain time stamp.

The loop in `replay_slope_changes` does at most 500 iterations. This means that if a user's balance is queried at a time stamp `t` more than 500 weeks after the last user `Point` was created, it will always return the balance from exactly 500 weeks after `Point.ts` instead of at time `t` if no gas limits are exceeded.

This may return a balance that is too high. It will have no effect on the global balance, so `totalSupply` will no longer return the sum of user balances.

Code corrected

A maximum lock time named `MAX_N_WEEKS` has been added and set to 522 weeks. The relevant loop was changed and iterates over 522 weeks now. Calling `modify_lock` will revert if `unlock_time` is set to more than `MAX_N_WEEKS`.

6.12 Inconsistent Use of `last_checkpoint`

Design Low Version 1 Code Corrected

In `_checkpoint_global`, the variable `last_checkpoint` is set to `last_point.ts`.

```
last_checkpoint: uint256 = last_point.ts
[...]
if block.timestamp > last_point.ts:
    block_slope = SCALE * (block.number - last_point.blk) / (block.timestamp - last_point.ts)
```

The variable is later used inconsistently. In some cases, `last_point.ts` is used and in other cases `last_checkpoint`. Both variables are set to the same value `t_i` in the loop.

Code corrected

`last_point.ts` and `last_checkpoint` are now used consistently.

6.13 Missing Explicit View Decorator in `_find_timestamp_epoch`

Design | Low | Version 1 | Code Corrected

The `_find_timestamp_epoch` function in `RewardPool` does not modify state but is missing the `@view` decorator.

Code corrected

The `@view` decorator has been added.

6.14 Redundant Functionality in `RewardPool`

Design | Low | Version 1 | Code Corrected

The contract `RewardPool` implements some functions which are already present and working in `veYFI`.

- `ve_for_at` is equivalent to `veYFI.balanceOf`.
- `_find_timestamp_epoch` is equivalent to `veYFI.find_epoch_by_timestamp(veYFI.address)`.
- `_find_timestamp_user_epoch` is equivalent to `veYFI.find_epoch_by_timestamp`.

Without good reasons we do not see a necessity to have redundant implementations in both contracts.

Code corrected

All redundant functions mentioned above have been removed.

6.15 Unused Function Argument `ve` in `RewardPool`

Design | Low | Version 1 | Code Corrected

The functions `_find_timestamp_epoch` and `_find_timestamp_user_epoch` use `ve` as argument for the voting YFI contract but do not use the argument in the function, as the immutable `VEYFI` is used.

Code corrected

The ve argument has been removed.

6.16 Unused State Variable `total_received` in `RewardPool`

Design Low Version 1 Code Corrected

The state variable `total_received` in `RewardPool` is defined but not used.

Code corrected

The unused state variable `total_received` has been removed from the contract.

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 Checkpoint Can Run Out of Gas

Note Version 1

`_checkpoint_global` has a `for` loop that is capped at 255 iterations. In each loop iteration, there is a cold `SSTORE` operation, which is expensive in terms of gas. This can cause the function to have a gas cost that is higher than the Ethereum block gas limit even at less than 255 iterations.

If this happens, the `_checkpoint` function will always revert. This will also cause `withdraw` to revert, which will make it impossible for users to withdraw their locked tokens.

The number of loop iterations depends on how long it has been since the last call to `_checkpoint`. There is one loop iteration per week. This problem will only occur if `_checkpoint` is never called for a very long time, which is unlikely.

7.2 Gas Heavy Lookup

Note Version 1

If a user simply locks an amount and often votes (e.g. calling `getPriorVotes`), voting gets more expensive over time due to the fact that all weeks that have passed since the lock point need to be iterated through to check for slope changes. The same calculations are repeated each time the balance is queried and not saved to storage.

This operation is relatively costly. In some cases, it might even be beneficial in terms of gas to simply top up the locked amount with some dust to create a new checkpoint and prevent looping over many weeks when querying balance.

Hence, a user should be aware of the fact that simply locking for a long time and voting will get more expensive after certain time than setting new checkpoints with dust amounts.

7.3 Possible Non-Claimable Rewards

Note Version 1

If `_checkpoint_token` is not called for more than 40 weeks, the rewards of the first user that calls `claim` will become unclaimable for the weeks between 40 and 50.

The function gets called regularly with normal use, so this is unlikely to happen in practice.

7.4 Slope Rounding Can Leave Dust

Note Version 2

When a position with a kink is created, its `bias` is set to `lock.amount` and `slope` is set to `0`. At the time of the kink, the slope is set to `lock.amount / MAX_LOCK_DURATION`.

Let us assume that the kink's `slope_change` happens exactly `MAX_LOCK_DURATION` seconds before `lock.end`. If this is the case, the `bias` at `lock.end` will be `lock.amount - (lock.amount / MAX_LOCK_DURATION) * MAX_LOCK_DURATION`. Due to rounding in integer arithmetic, the bias will contain a small positive value if `lock.amount` is not divisible by `MAX_LOCK_DURATION` without remainder.

User balances and global balances will keep this non-zero `bias` after `lock.end` and it will not decay over time, as `slope` will be `0`.

The specified behavior is that the bias at `lock.end` should be zero.

Due to the Kink timestamps are too early issue, the assumption that kink's `slope_changes` happen exactly `MAX_LOCK_DURATION` before `lock.end` does not hold as of **Version 2**.