PUBLIC

Code Assessment

of the Yearn yETH Smart Contracts

June 26, 2023

Produced for

yearn

by

CHAINSECURITY

Contents

1 Executive Summary 3
2 Assessment Overview 5
3 Limitations and use of report 9
4 Terminology 10
5 Findings 11
6 Resolved Findings 18
7 Open Questions 27
8 Informational 28
9 Notes 30

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 audit of the latest reviewed contracts of Yearn yETH according to Scope to support you in forming an opinion on their security risks.

Yearn implements a modified StableSwap pool for liquid staking derivatives and a staking vault. The pool token is yETH and can be staked into the Staking contract to earn rewards.

The most critical subjects covered in our review are asset solvency, functional correctness, access control and front-running. The security regarding functional correctness and front-running still has some potential to improve, see Implementation Mismatch With ERC-4626 and Possible to Frontrun the First Deposit in Pool. The security regarding other subjects is good.

Although we did not identify critical or highly severe issues during this review, we highlight that sandwiching attacks are important for the system as the curve's shape changes when Pool parameters get updated by privileged accounts, or when rates of underlying assets change significantly. Possible sandwiching attacks are described in section Notes.

Given the complexity of the system, we highly recommend extending significantly the test suite and only apply changes to the system after rigorous testing.

In summary, we currently find that the codebase provides a high level of security.

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	6
• Code Corrected	5
• Code Partially Corrected	1
Low-Severity Findings	19
• Code Corrected	8
• Specification Changed	2
• Code Partially Corrected	2
• Risk Accepted	2
• Acknowledged	5

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 `Pool.vy` and `Staking.vy` source code files inside the Yearn yETH 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	10 Apr 2023	4ba5ccdad2d12903a702593c728d1363f81e6695	Initial Version
2	22 May 2023	564ef429a338e55ef35a1f59a33e1a82e6cf528b	Version 2
3	08 Jun 2023	2c77af241db5a783274a185da757d5fbbd07d690	Version 3

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

2.1.1 *Excluded from scope*

All other contracts and associated third-party contracts that are not in the repository. We only assessed the technical risks, and we did not assess the economic sanity of the contracts.

2.2 System Overview

This system overview describes the initially received version (**Version 1**) 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.

The reviewed contracts allow users to put liquid staking derivatives (LSDs) in a modified Curve Stable Swap pool as liquidity. The pool token that users receive is yETH. In contrast to Curve Pools, the users will not receive any fees or staking rewards from the LSDs unless they stake their yETH in a staking contract. The total number of yETH tokens shall always equal `D`. Whereas `D` is the current value of the pool and calculated by the pool's invariant. Yearn modified Curve's invariant by adding weights to each token in the pool with a pool weight that defines when the pool is balanced. The invariant is:

$$A\frac{\prod_{i=1}^{n} x_i^{w_i n}}{(D\prod_{i=1}^{n} w_i^{w_i})^n} D^{n-1} \sum_{i=1}^{n} x_i + \prod_{i=1}^{n} x_i^{w_i n} = A\frac{\prod_{i=1}^{n} x_i^{w_i n}}{(D\prod_{i=1}^{n} w_i^{w_i})^n} D^n + (D\prod_{i=1}^{n} w_i^{w_i})^n$$

`x_i` is the virtual balance of token `i`. The value `x` is the balance multiplied by the rate. n is the number of tokens. A is a defined amplification factor.

The pool's management can change the pool weights, the amplification factor, and the number of different LSD tokens in the pool. When `D` changes, the pool mints or burns an amount of yETH tokens to/from the staking contract to keep the total supply and `D` in sync. Through minting to the staking contract, the staking contract receives the tokens to distribute to the users who staked. The tokens to be distributed are put into a queue. The queue consists of three buckets. A pending bucket, a streaming bucket and unlocked bucket. On a new week the queue progresses and funds from the pending bucket are put into the streaming bucket and then distributed to the users (unlocked buckets). Tokens to be distributed this week from the streaming bucket are distributed linearly over the week. The distribution increases the share of underlying asset to the users proportionally to their stake. Vice versa, when `Pool` makes losses and tokens are burned, the contract tries to remove tokens from the pending and streaming buckets first and as a last resort will touch the unlocked bucket, which affects directly the balance of users staking yETH.

The staking contract is a vault implementing the ERC20 and ERC4626 standards. Additionally, users who staked will have a voting weight. In principle, the voting weight will increase with the time a user has staked. The implemented voting weight calculation becomes more complex when the vault shares are transferred.

The main functionalities of the two contracts are:

Pool:

- `add_liquidity`, `remove_liquidity`, `remove_liquidity_single`: Add and remove liquidity from the pool.
- `swap` and `swap_exact_out`: The user can swap tokens and define either the input and a minimum output or the desired output and a maximum input amount.
- `update_rates`: A function that can be called by anyone to update the token values with the current rate. The rate is pulled from a rate provider that must be defined for each token. Sudden rate changes above a threshold can only be applied by management.
- `set_ramp`: Allows the management to change the target weights (weights that define a balanced pool) and the amplification factor. As these changes change the Curve the changes are ramped (split into small changes over time) to prevent successful sandwich attacks. With each operation or by explicitly calling update functions, these changes can be applied gradually.
- `update_weights`: Can be called by anyone to update the weights and amplification factor when a weight ramping is in progress.
- `set_ramp_step`: Sets the minimum time between two ramp steps in seconds.
- `stop_ramp`: Stops the ramping process at current values.
- `add_asset`: Allows to add a new asset to the pool. The defined weight for the asset will be taken equally distributed from all existing asset weights. The caller must assure that effective amplification before and after the call is the same.
- `rescue`: Allows the management to transfer tokens from the pool that are not the asset or the pool token.
- `skim`: In case the pool has more tokens of an asset than the pool should have and has accounted for, management can transfer out these tokens.
- `set_swap_fee_rate`: The management can set a fee for swapping. Between 0 and 100 percent.
- `set_weight_bands`: The management can set the maximum and minimum bounds for a token weight in the pool. If a token value is outside of the bounds, operations will revert to prevent deviations too far away from the target weights.
- `set_rate_provider`: Management function to set the rate provider contract for a token.
- `set_staking`: Management function to set the staking contract address.
- `set_management`: Management function to change the management address.
- `set_guardian`: Management or the guardian can call the function to set the guardian address.
- `pause` and `unpause`: Guardian and management role can pause or unpause the pool. In a paused pool no swaps, liquidity additions, single-sided removals, ramping or rate provider updates are possible.
- `kill`: Will pause a pool forever by removing the possibility to unpause the pool.

Staking:

- `transfer`, `transferFrom`: Transfer shares to another address. In the case of `transferFrom` an approval is needed to transfer on behalf of the user.
- `vote_weight`: Returns the voting weight of an account at the end of the last week. Any change on the amount of staked shares during the ongoing week is not considered by this function. Therefore, an account withdrawing (or transferring) all of its staked shares, still has a non-zero voting weight during that week.
- `approve`: Allows users to approve an account to use `transferFrom` to move their assets.
- `deposit` and `mint`: Users can deposit yETH into the vault to be eligible for rewards. In the case of a deposit, the user specifies the amount of yETH they want to deposit. Calling `mint` allows one to specify the number of vault shares to be received.
- `withdraw`, `redeem`: Users can withdraw their yETH from the vault. In the case of a withdraw, the user specifies the amount of yETH they want to withdraw. Calling `redeem` allows one to specify the number of vault shares to be redeemed.
- `update_amounts`: Updates the amounts in each of the three buckets (pending, steaming and locked) by moving funds into pending and from pending to streaming or locked.
- `rescue`: Allows the management to transfer tokens from the vault that are not yETH (vault assets) tokens.
- `set_performance_fee_rate`: The management can set a fee for the vault. The fee is taken if the vault's total assets increased.
- `set_half_time`: This function can be called by management to set the time after which a user will have half of the maximum voting weight. This only applies to cases when `t` has not been adjusted yet through transfers.
- `set_management`: This function allows the management to change the management address.
- `set_treasury`: This function allows the management or treasury to change the treasury address that will collect the fees.

2.2.1 Roles and trust model:

Each contract has a management role that needs to be fully trusted as it performs operations that could stop the contracts from working correctly. The pool additionally has a guardian role. This role can pause and unpause the contract. Hence, this role is also assumed to be fully trusted. The rate providers used in the pool contract are also fully trusted, we assume they return the correct rates at all times (resistant to DoS attacks) and they are resistant to attacks which alter the rate by changing token amounts in a 3rd party system. We assume the rate providers publish rates frequently and they represent the correct backing ratio of the asset in ETH. We assume the ratio is always in 18 decimals.

We assume the underlying assets of a pool are ERC20-compliant, use 18 decimals, and are non-malicious. Additionally, we assume they do not have special behaviors such as rebasing, charging fees on transfers, or implementing transfer callback (like ERC777 or ERC677). Finally, the rate of pool assets is roughly equal to 1 ETH.

The oracles are assumed to be safe and price fluctuations are small enough to not become an issue as described in Decreasing Pool Value through rate updates.

All parameter changes need to be simulated rigorously and carefully evaluated before being applied.

In this review, we assume that the `LP token` in `Pool` and the `asset` in `Staking` is `yETH` (as implemented in `Token.vy`). Furthermore, we assume that `yETH` reverts on `burn` function if there is not enough balance, and that only the pool contract has the minter role in `yETH`.

The initialization process is assumed to be done correctly, such that the pool and staking contract integrate and work as intended. Finally, we assume the contracts in scope of this review are not upgradable.

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:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- 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	1

- Circumvention of Ramping Code Partially Corrected

Low-Severity Findings	9

- Decreasing Pool Value Through Rate Updates Acknowledged
- Guardian Can Front-Run Kill Command Acknowledged
- Implementation Mismatch With ERC-4626 Code Partially Corrected
- Inefficient Initial Approximation Value for Pi in Supply Calculation Acknowledged
- Missing Sanity Checks Code Partially Corrected
- Possible to Frontrun the First Deposit in Pool Risk Accepted
- Possible to Update Ramp Step While Ramping Risk Accepted
- Violation of Sum of Weights Acknowledged
- Voting Weight Increase Differs for New and Existing Positions Acknowledged

5.1 Circumvention of Ramping

Design Medium Version 1 Code Partially Corrected

The `management` account of `Pool` can initiate a change of asset weights or the amplification factor for a pool. The change should be applied slowly to minimize profits from sandwiching attacks, see Sandwiching Curve changes. However, the function `add_asset` allows the `management` to modify (reduce) the weights of assets and the amplification factor by avoiding the ramping limitations entirely. The natspec description of the function notes:

```
@dev Every other asset will have their weight reduced pro rata
@dev Caller should assure that effective amplification before and after call are the same
```

Code partially corrected:

The function `add_asset` now sets an upper limit of 1% on the initial weight of the new asset being added to the pool. Although this reduces the likelihood of accidentally changing the weights of assets significantly, it does not enforce any restriction on the amplification factor (`_amplification` represents the term `A * f^n` in the whitepaper). Therefore, `management` should consider sandwiching attacks when calling this function and carefully choose input parameters. The response of Yearn is:

```
Added a limit to the new asset weight, it is not allowed to exceed 1%. Since
the `amplification` in the pool represents 'A * f^n', we cannot easily put
bounds on the amplification factor. It is up to the management role to make sure
the call cannot be sandwiched, either by picking a new amplification factor and
initial weight (even lower than 1%) that minimises the effect or by first pausing
the pool and in a separate call add the asset before unpausing.
```

5.2 Decreasing Pool Value Through Rate Updates

Design | Low | Version 1 | Acknowledged

Big rate updates might drain value from the pool. Assuming a swap_fee of `0.3`%, then a rate update of `1`% (e.g. from `1.00 -> 1.01`) seems already *too big*. Generally, it seems that any rate change twice as big as the `swap_fee` (so any rate change `>= 0.6`% for a `swap_fee` of `0.3`%) leads to this issue.

We provide an example with three assets. In the beginning, everything is balanced and the rates are all 1.00. Now, the pool can lose value in the following way:

- The price of asset 2 in the market rises from `1.00 -> 1.01`
- Assuming an efficient market, trades happen inside the pool which imbalance the pool so that `get_dy(2, 0, 10**18) == get_dy(2, 1, 10**18) == 1.01 * 10**18`. (If such trades do not happen an attacker can front-run the rate update with such trades.) Here the pool is selling asset 2 too cheaply.
- Now the rate update is performed, setting the rate of asset 2 from `1.00 -> 1.01`.
- As a consequence, the pool's exchange rate from `asset 2 => asset 1` (and `asset 2 => asset 0`), will now go to roughly `1.02`. The pool is paying too much to obtain asset 2.
- Therefore an attacker trail-runs the rate update and sells asset 2 to the pool.
- Eventually, the price of asset 2 goes back down from `1.01 -> 1.00`.
- Again trades happen which change the balance of the pool so that it has a 1:1 exchange ratio between the assets. Here the attacker or others sell asset 2 to the pool.
- Now the rate update is performed, setting the rate of asset 2 from `1.01 -> 1.00`.

As a consequence, the pool's exchange rate is so that asset 2 can be bought too cheaply. Hence, the attacker buys asset 2 cheaply. After all, trades are settled, the pool has fewer funds than at the beginning. In this example, they might have lost 0.3% of value. As a consequence, the balance in the Staking contract is now smaller, even though all prices are the same as in the beginning.

Generally, the significance of this issue depends on different parameters, like number of assets, weights and amplification factor. In some configurations, it will be more severe than in others. A combination of smaller, parallel rate updates for different assets might also be problematic.

Acknowledged

Yearn is aware of the issue and acknowledges it. They will take care to mitigate it by only using high-quality rate providers that return the backing rate on the beacon chain which they assume will not fluctuate much to be an issue. Additionally, these oracles are assumed to be not influenced by the market. Yearn emphasized that every oracle will be rigorously tested and simulated before being used.

5.3 Guardian Can Front-Run Kill Command

Security Low Version 1 Acknowledged

The `guardian` role in contract `Pool` can set or unset the `paused` flag, while only `management` can set the flag `killed` to `true`. For the function `kill` to execute successfully, the flag `paused` should be `true`. Therefore, `guardian` can prevent the execution of function `kill` by frontrunning the transaction with a call to function `unpause`.

Acknowledged:

Yearn acknowledges the risk of frontrunning and accepts the consequences of the attack with the following reasoning: "In the unlikely case the guardian decides to grief by front-running such a call, management has the option to replace the guardian".

We want to note that the replacement of the `guardian` can also be front-run by the `guardian` as `set_guardian` can be called by both roles.

5.4 Implementation Mismatch With ERC-4626

Correctness Low Version 1 Code Partially Corrected

The contract `Staking` implements the external functions specified in the standard `ERC4626`. The implementation of functions `maxWithdraw` and `maxRedeem` is not in line with the standard. Both functions return `max_value(uint256)`, but the standard for `maxWithdraw` (similarly for `maxRedeem`) states:

```
MUST return the maximum amount of assets that could be transferred from ``owner``
 through ``withdraw`` and not cause a revert, which MUST NOT be higher than the
 actual maximum that would be accepted (it should underestimate if necessary).

MUST factor in both global and user-specific limits, like if withdrawals are
 entirely disabled (even temporarily) it MUST return 0.
```

Code partially corrected:

Both functions have been updated in Version 3 to return the maximum amount of assets or shares that can be withdrawn or redeemed by address `_owner`.

However, the special case when `totalSupply` cannot be less than `MINIMUM_INITIAL_DEPOSIT` is not handled correctly. Therefore, it is possible that both functions `maxWithdraw` and `maxRedeem` return non-zero values, while the respective functions `withdraw` and `redeem` could revert, which violates the standard.

5.5 Inefficient Initial Approximation Value for Pi in Supply Calculation

Design Low Version 1 Acknowledged

When a rate change or balance change occurs, the starting value for `pi` (vb product), to later approximate the supply, is:

```
vb_prod * self._pow_up(prev_rate * PRECISION / rate, wn) / PRECISION
```

or

```
vb_prod_final * self._pow_up(prev_vb * PRECISION / vb, wn) / PRECISION
```

The result of this calculation is then passed into `_calc_supply`. In `_calc_supply` an iterative method is used to approximate the correct supply. Starting with the result as the first guess for `r` in:

```
r = unsafe_div(unsafe_mul(r, sp), s)
```

The initial guess seems inefficient and almost always dominated by the old value for pi as starting value.

Acknowledged:

Yearn replied:

```
The value for pi needs to be updated somewhere before or during the iteration process,
as otherwise the supply will not converge to the correct value. It might be possible
to save on iterations by updating pi to the correct value after the first iteration,
but such a change would significantly complicate the function and as such is deemed
not worth it.
```

5.6 Missing Sanity Checks

Design Low Version 1 Code Partially Corrected

The following functions update important state variables but do not perform any sanity check on inputs.

`Staking` contract:

1. `_asset` in function `__init__`.
2. `_fee_rate` in function `set_performance_fee_rate`.
3. `_management` in function `set_management`.
4. `_treasury` in function `set_treasury`.
5. Version 2: If `_value` is larger than current allowance, an underflow happens.
6. Version 2: `_spender` in functions that modify allowance.

`Pool` contract:

7. `_assets` in function `__init__` can include duplicate.
8. `_duration` in function `set_ramp` can be `0`.
9. `_rate_provider` in function `set_rate_provider`.
10. `_staking` in function `set_staking`.
11. `_guardian` in function `set_guardian`.
12. `_management` in function `set_management`.

Code partially corrected:

Missing sanity checks reported in the Staking contract have been added. Additionally, the same checks were applied in the Token contract. In the Pool contract sanity checks for points 10 and 12 were added. The sanity check for `_guardian` (point 11) is intentionally left out to allow for flexibility of burning the role in the future.

5.7 Possible to Frontrun the First Deposit in Pool

Design | Low | Version 1 | Risk Accepted

The first liquidity provider in a pool does not pay any fee. Other liquidity providers do not pay a fee only if they deposit tokens in the same ratio as the current state of the pool. If a user adds liquidity into a pool in an unbalanced manner (e.g., single token or with different ratios from the current state), a fee is payed.

An attacker can frontrun the first deposit to add tokens in a pool in a wrong ratio such that the victim user pays high fees. The fees are sent to the Staking contract and can be claimed after a delay by users that have staked their `yETH`. The profits of the attacker depend on the amount of tokens deposited by the victim and the share of `yETH` staked by the attacker at the time rewards (includes fee) move to `unlocked` bucket in Staking contract.

Risk accepted:

Yearn replied:

```
This is acceptable behaviour, and can be mitigated by setting a very tight value for the
minimum amount of tokens received for the initial deposit.
```

5.8 Possible to Update Ramp Step While Ramping

Correctness | Low | Version 1 | Risk Accepted

The function `set_ramp_step` sets a new `ramp_step` without checking if there is currently an active ramp. Raising `ramp_step` while there is an active ramp increases the risks of sandwiching attacks (see Sandwiching Curve changes) as the `_duration` of ramping remains the same.

Risk accepted:

Yearn replied:

```
We’d like to retain the option to increase the step size, even during a ramp. However,
management should take care not to increase it to such a degree that it affects the
sandwich risk in a significant way. This is in line with the responsibilities the
management account already has. It has the ability to set the duration of a ramp,
which suffers from the same consequences if not set properly
```

5.9 Violation of Sum of Weights

Correctness Low Version 1 Acknowledged

The trading curve is defined by a function that assumes that the sum of all weights in a pool equals `PRECISION` (100%). However, this invariant does not apply always as the weights of assets change when: i) adding a new asset, ii) updating weights in a ramp. Therefore, it is possible that these dynamic changes of weights break the invariant due to rounding errors. For instance, `current` is rounded down in the following code:

```
if current > target:
    current = current - (current - target) * span / duration
else:
    current = current + (target - current) * span / duration
```

Acknowledged:

Yearn acknowledges the issue.

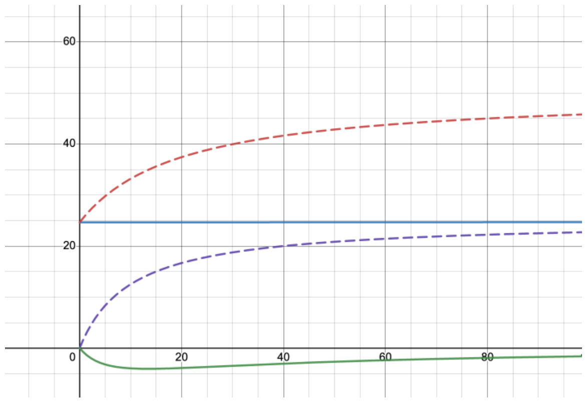
5.10 Voting Weight Increase Differs for New and Existing Positions

Correctness Low Version 1 Acknowledged

Transferring shares to a position increases the voting weight of the receiving position. For the same amount of shares transferred, the new voting weight depends on the existing state of the receiver, namely variable `Weight.t`.

The picture below plots voting power where x-axis is the time and y-axis is the voting weight. The blue line illustrates a position that has staked 25 shares for a long time. If this position receives 25 more shares, its voting power will change over time as shown by the red dashed line.

However, if 25 shares are sent to a new position, its voting weight increases according to the violet dashed line. The green line shows the difference on the voting weight after new tokens are received between an existing position (red dashed line) versus existing position and new position that receives tokens (blue line and violet dashed line). The plot suggests that receiving shares in new positions instead of existing ones maximizes the voting weight of a party.



Acknowledged:

Yearn replied:

```
This is an acceptable side effect of our choice to have a asymptotically increasing
weight function.
```

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	0

High-Severity Findings	0

Medium-Severity Findings	5

- Incorrect Computation of Product Term Code Corrected
- Missing Transfer of Tokens When Adding New Assets Code Corrected
- Pool Might End up With Less Shares Than MINIMUM_INITIAL_DEPOSIT Code Corrected
- Share Distribution Depends on First Deposit Code Corrected
- Wrong Calculation of Voting Weight for Withdrawn Shares Code Corrected

Low-Severity Findings	10

- Approve Can Be Frontrun Code Corrected
- Default Target Weight Code Corrected
- Incomplete Specifications for Paused Pool Specification Changed
- Inconsistent Behavior of Conversion Function Code Corrected
- Mismatch of Code With the Specification for Pending Rewards Specification Changed
- Missing Slippage Protection When Adding New Asset Code Corrected
- No Meaningful Revert Messages Code Corrected
- Possible to Lock Management Role Code Corrected
- Types of Variables in Weight Code Corrected
- Unused Event in Staking Code Corrected

6.1 Incorrect Computation of Product Term

Correctness Medium Version 1 Code Corrected

The product term pi in the whitepaper depends on `D`, `w_i` and `x_i`. The function `_calc_vb_prod` takes as input _s which is the sum of tokens in the `Pool` and is different from `D` if the pool is not at equilibrium point.

This issue was found by Yearn also while the review was ongoing.

Code corrected:

The function `_update_weights` has been updated to pass `supply` when calling function `_calc_vb_prod` as follows:

```
supply: uint256 = self.supply
if supply > 0:
    vb_prod = self._calc_vb_prod(supply)
```

Furthermore, the function `_calc_vb_prod_sum`, which is called on first deposit or when adding a new asset, is revised to not take _s (sum term) as an input parameter.

6.2 Missing Transfer of Tokens When Adding New Assets

Correctness Medium Version 1 Code Corrected

Function `add_asset` can be called only by `management` which is trusted to behave correctly in the contract `Pool`. The parameter `_amount` in function `add_asset` is the amount of tokens that are deposited into the `Pool` when the new asset is added. However, the code does not pull the funds from an external account (if approved) or check that the `Pool` has already the required balance (if already transferred).

Code corrected:

The issue has been resolved by adding the code in function `add_asset` that pulls the respective tokens from `msg.sender`:

```
assert ERC20(_asset).transferFrom(msg.sender, self, _amount, default_return_value=True)
```

6.3 Pool Might End up With Less Shares Than `MINIMUM_INITIAL_DEPOSIT`

Design Medium Version 1 Code Corrected

The new `MINIMUM_INITIAL_DEPOSIT` amount does not mitigate that a user manipulates the share amount to be very low before another user deposits or rewards are accounted.

A malicious user might deposit `MINIMUM_INITIAL_DEPOSIT` tokens but could immediately call redeem in such a way that they end up with one remaining share. This breaks the assumption that the pool always has a minimum amount of assets, which could have unintended side effects.

Code corrected:

The internal function `_withdraw` has been updated in Version 3 to enforce that `totalSupply` in the contract Staking is either `0` or larger than `MINIMUM_INITIAL_DEPOSIT`. This restriction is implemented in the following code:

```
if total_shares < MINIMUM_INITIAL_DEPOSIT:
    assert total_shares == 0
```

6.4 Share Distribution Depends on First Deposit

Correctness **Medium** **Version 1** **Code Corrected**

The user's shares when depositing an amount of yETH are calculated as:

```
_assets * _total_shares / _total_assets
```

However, in the case of the first deposit, the number of assets deposited is the number of shares the user receives. In case a user deposits a very small amount (at best 1 WEI), they would receive 1 share. When the total assets increase because profits are made, the fraction `_total_shares / _total_assets` will become `0` for amounts smaller than `_total_assets`. Additionally, when adding assets, they need to be multiples of `_total_assets`. Hence, the first deposit determines the minimum step size or rounding error for the following deposits.

The was independently reported by Yearn while the review was ongoing.

Code corrected:

Yearn implemented a practical solution by specifying a minimum deposit amount `MINIMUM_INITIAL_DEPOSIT` of `1e15` which makes the attack unlikely in practice. However, we would like to highlight that the core issue is still present even with this practical mitigation. The issue arises only in case of a high discrepancy between the first deposit and the potential rewards which now should be higher by a factor of `1e15`.

6.5 Wrong Calculation of Voting Weight for Withdrawn Shares

Correctness **Medium** **Version 1** **Code Corrected**

By depositing `yETH` into the `Staking` contract, users gain voting power that continuously increases over time. The voting power depends on the amount of shares a user has and the time they have been deposited in the contract. The contract stores two checkpoints for a user: `weights` and `previous_weights`. `weights` track the latest state of a position, while `previous_weight` stores the state of the position in the week before latest changes.

The function `vote_weight` should consider `previous_weights` when the position is updated on the ongoing week. However, the code checks for two conditions as follows:

```
if weight.week > week or weight.week == 0:
    weight = self.previous_weights[_account]
```

The second condition is `true` for users that have withdrawn or transferred out all their shares. In this case, the code still considers their `previous_weights` and incorrectly computes a voting power based on the state of the position before its shares were removed. This can be exploited by attackers to create positions that gain voting power, and then move shares to a new position.

This issue was uncovered by Yearn also while the review was ongoing.

Code partially corrected:

Version 2: The internal function `_update_account_shares` has been revised to reset only field `t` when an account withdraws all of its shares:

```
if shares == 0:
    t = 0
    last_shares = 0
```

The function `vote_weight` checks if the position of an account has been updated on the ongoing week as follows:

```
if week > current_week or week == 0:
    packed_weight = self.previous_packed_weights[_account]
```

The second condition `week == 0` is `true` only for empty accounts, which should have a voting weight of `0` and there is no need to consider their previous state.

Code corrected:

Version 3: The unnecessary check `week == 0` has been removed from function `vote_weight`.

6.6 Approve Can Be Frontrun

Security | Low | Version 1 | Code Corrected

The function `approve` in `Staking` contract is vulnerable to frontrunning attacks. The function `approve` always overwrites the current value without checking if the allowance has been consumed or not.

Assume a scenario where Alice provides an allowance of value `X` to a spender. Then, she decides to change the allowance to a value `Y`. The spender can front-run the second transaction, spend `X`, and then spend the new allowance `Y` also. This attack vector and possible mitigations are discussed in EIP20.

Code corrected:

`increaseAllowance` and `decreaseAllowance` functions were added. These functions are similar to `approve` function, but they do not overwrite the current value. Instead, they increase or decrease the current value with a given delta.

6.7 Default Target Weight

Correctness | Low | Version 1 | Code Corrected

The function `weight` in contract `Pool` returns `0` as the default target weight when no ramp is active:

```
if self.ramp_last_time == 0:
    target = 0
```

Furthermore, the function `add_asset` does not set `0` as `target` weight although no ramp is active.

Code corrected:

The external function `weight` has been updated to return the current `weight` of an asset in the pool when there is no active ramp.

6.8 Incomplete Specifications for Paused Pool

Correctness Low Version 1 Specification Changed

When a pool is paused no swaps can be executed. Furthermore, rate providers cannot be updated while the pool is in this state as `_update_rates` reverts. Specifications do not describe these behaviors.

The following functions cannot be executed when a pool is paused:

- update_rates
- update_weights
- set_ramp
- swap
- swap_exact_out
- add_liquidity
- remove_liquidity_single
- set_rate_provider

Specifications changed:

The specifications regarding `pause` mode have been extended in file `specification.md`.

6.9 Inconsistent Behavior of Conversion Function

Correctness Low Version 1 Code Corrected

External view functions `convertToShares` and `convertToAssets` return the input value, `_assets` and `_shares` respectively, when `total_assets` is `0`. However, on the same conditions (`` _total_assets == 0``) both internal functions `_preview_deposit` and `_preview_mint` return `0`. Therefore, depositing and minting in this scenario reverts.

Code corrected:

Yearn corrected the code and both external functions are now in line with the internal ones.

6.10 Mismatch of Code With the Specification for Pending Rewards

Correctness Low Version 1 Specification Changed

The specifications of the contract `Staking` state:

```
If the balance has increased, it is added to the pending bucket. If one or more week has been missed,
the increase is distributed instead over the three buckets fairly.
```

However, the function `_get_amounts` adds rewards to the `streaming` bucket if it is called on the first day of a new week:

```
if weeks == 1 and block.timestamp % WEEK_LENGTH <= DAY_LENGTH:
    streaming += rewards
```

Specification changed:

Yearn added a more concise specification for this scenario:

```
If the first update of the week is in the first day, it is added to the streaming bucket directly instead.
```

6.11 Missing Slippage Protection When Adding New Asset

Security Low Version 1 Code Corrected

The management can add a new asset into a Pool by calling the function add_asset. The caller should send _amount tokens of the new asset to the pool, which increases the overall value of the Pool. The function mints the difference in the total supply (supply - prev_supply) as LP tokens to the address _receiver, however no slippage protection is implemented.

Code corrected:

The function add_asset has been revised to take an additional argument _min_lp_amount as input and now explicitly asserts that supply has strictly increased and the caller receives more LP shares than _min_lp_amount:

```
...
assert supply > prev_supply
lp_amount: uint256 = unsafe_sub(supply, prev_supply)
assert lp_amount >= _min_lp_amount
PoolToken(token).mint(_receiver, lp_amount)
...
```

6.12 No Meaningful Revert Messages

Design Low Version 1 Code Corrected

Reverts could emit meaningful messages to provide the reason for failed calls. The downside of informing users accordingly is the slightly increased gas costs. Hence, Yearn needs to evaluate if a meaningful revert message should be returned.

Code corrected:

Yearn added selected revert messages.

6.13 Possible to Lock Management Role

Design | Low | Version 1 | Code Corrected

Both contracts `Staking` and `Pool` implement the function `set_management` that allows the existing `management` account to set a new `management` address. As `management` is responsible for setting multiple parameters of contracts, measures should be taken to avoid mistakes when updating it. Besides sanity checks, the update of critical roles that cannot be recovered should follow the `set/accept` approach.

Code corrected:

Both functions were changed to a commit/accept scheme with two functions `set_management` and `accept_management`.

6.14 Types of Variables in Weight

Design | Low | Version 1 | Code Corrected

The types `uint16`, `uint56` and `uint128` are used for variables of `struct Weight`. Together these values fit in a storage slot (256 bits). However, Vyper does not optimize storage used by packing together variables that fit in 32 bytes. As each value is stored in a separate storage slot, EVM uses additional operations to convert the value from 32 bytes to the correct type.

Code corrected:

Yearn added a custom way to pack the variables in a single storage slot for the variables: `previous_packed_weights` and `packed_weights`. This optimization on the storage comes with slighly added gas costs on execution due to packing and unpacking of variables in a single storage slot.

6.15 Unused Event in Staking

Design | Low | Version 1 | Code Corrected

The event `SetMinter` in the contract `Staking` is not used.

Code corrected:

The unused event has been removed.

6.16 Functions Return True Always

Informational | Version 1 | Code Corrected

The natspec description for the return value of functions `transfer` and `transferFrom` states:

```
@return Flag indicating whether the transfer was successful
```

Both functions return only `True`, otherwise they revert.

Code corrected:

The natspec description for the return value has been updated: `@return True`.

6.17 Missing Natespec

Informational | Version 1 | Code Corrected

A majority of the critical logic is implemented in internal functions. In-line documentation and proper natspec for all functions can significantly improve code readability to understand correctly the intended behavior of the code.

Code corrected:

Natspec was added to all functions.

6.18 Possible to Index Event Parameters

Informational | Version 1 | Code Corrected

It is recommended to index the relevant event parameters to allow integrators and dApps to quickly search for these and simplify UIs. We would like to highlight that `asset` could be indexed in the respective events.

Code corrected:

The parameter `asset` is now indexed in events `Swap`, `RemoveLiquiditySingle` and `SetWeightBand`.

6.19 Possible to Mark Functions as View

Informational | Version 1 | Code Corrected

Functions `virtual_balance` and `rate` in `Pool` do not modify the state and can be marked as `view`.

Code corrected:

Both functions have been marked as view functions.

6.20 Redundant Code in _Calc_Supply

Informational | Version 1 | Code Corrected

The code inside `if/else` branches in the function `_calc_supply` is redundant and could be removed if the delta between values `s` and `sp` is computed first.

Code corrected:

The function `_calc_supply` has been revised to avoid the redundant code.

6.21 Return Values When Removing Liquidity

Informational Version 1 Specification Changed

The function `remove_liquidity_single` returns `dx` which is the amount of tokens being withdrawn for the target asset. However, the function `remove_liquidity` does not return any value.

Specification provided:

Yearn informed ChainSecurity that this was done intentionally as a gas saving measure, as otherwise it would need to construct and return an array of up to 32 values of type `uint256`.

6.22 Transfers of 0 Values Revert in Staking

Informational Version 1 Code Corrected

Both functions `transfer` and `transferFrom` check that value being transferred is non-zero (`assert _value > 0`). This behavior is not in line with EIP20 which has the following note:

```
Transfers of 0 values MUST be treated as normal transfers and fire the Transfer event.
```

Code corrected:

Yearn changed the code to allow zero value transfers.

7 Open Questions

Here, we list open questions that came up during the assessment and that we would like to clarify to ensure that no important information is missing.

7.1 Adding New Assets When Paused

Open Question **Version 1**

The function `add_asset` does not check if the pool has been paused when adding a new asset. Is this behavior intentional?

7.2 Approval Events on transferFrom

Open Question **Version 1**

The `Staking.transferFrom` function does not emit any event regarding the approval change. Thus, it is not possible to recover state based on Approval+Transfer events. While this is compliant with ERC4626/ERC20 specification, some libraries like OpenZeppelin, emit explicit `Approval` event during the `transferFrom`. On the other hand, `DAI` token does not emit such event. We would like to bring this detail to your attention and know if it is as expected in your case.

7.3 Total Assets Can Be 0

Open Question **Version 1**

Both functions `_preview_deposit` and `_preview_mint` check if `_total_assets == 0` although `_total_shares` are non-zero. Can you please describe the scenarios when this happens?

8 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.

8.1 Extracting Value From First Deposit in Pool

Informational Version 2

The attacks based on strategies that artificially inflate the value of LP shares in Pool are unlikely to succeed due to the way how rewards (donations) are tracked in different buckets. Nevertheless, it is theoretically possible for an attacker to extract value from the first depositor if certain conditions hold before first deposit, e.g., significant rewards are ready to be moved to `unlocked` bucket.

8.2 Incomplete Natspec

Informational Version 1

The natspec description for the return value of function `update_weights` is incomplete.

8.3 Missing Events in Staking Contract

Informational Version 1 Code Partially Corrected

Functions `rescue`, `set_half_time`, `set_management` and `set_treasury` in contract `Staking` update the state, but no event is emitted.

Code partially corrected:

The respective events in the functions listed above were added except function `rescue`, which still does not emit an event.

8.4 Preview Functions Round in Favor of Users

Informational Version 2

The functions `_preview_withdraw` is used to calculate the number of shares a user needs to pay for withdrawing a given amount of assets. The calculation rounds in favor of the user. This means the user needs to pay slightly less shares for the respective assets. Hence, reducing the value of all shares. The same issue is also present in `_preview_mint`. The magnitude of the rounding error depends on the share-to-assets ratio.

This violates the invariant that the share value can only go down by incurred losses. Still, the impact should be limited and the issue is mainly theoretical.

8.5 Theoretical Underflow in _Get_Amounts

Informational **Version 1**

The function `Staking._get_amounts` can theoretically underflow when the `shortage` is higher than the sum of all tokens accounted in the buckets. The underflow happens in the statement `unlocked -= shortage`. However, practically this should not happen as the loss cannot be larger than the balance of Staking contract in `yETH`.

9 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.

9.1 Assets Cannot Be Removed From Pool

Note **Version 1**

Contract Pool implements the function `add_asset` that allows the management account to add new assets into the Pool, up to a total of 32 assets. We highlight that the contract does not implement a functionality to remove an asset from a Pool. Therefore, the asset removal requires a redeployment of the contract, which forces all LPs to withdraw their liquidity from the old Pool and deposit into the new one.

9.2 Assumption on Balance of Staking

Note **Version 1**

The internal function `_update_supply` is called when key functionalities of the contract `Pool` are performed. Based on the activity of the `Pool` LP tokens are minted to the `staking` (if `supply` increases), or LP tokens are burned from the `staking` (if `supply` decreases). The implementation of the function assumes that the `staking` contract has always enough balance in yETH to cover the losses of the pool such that the burning of LP tokens will always succeed.

Yearn is aware that after deployment or in certain conditions (e.g., no staked tokens) this assumption might not hold, and extra measures need to be taken for the function to work as intended.

9.3 Buckets Can Be Updated at Most Once per Block

Note **Version 1**

The function `_get_amounts` returns the current state if the buckets have already been updated in the same block:

```
if updated == block.timestamp:
    return self.pending, self.streaming, self.unlocked, 0, 0
```

If the `Staking` contract receives rewards in a block, after function `_update_totals` has already been called, the `pending` and `streaming` buckets do not get updated. Note that, the `unlocked` bucket is always updated when users stake or unstake their yETH tokens.

9.4 Charged Fees Are Unclear

Note **Version 1**

The function `add_liquidity` charges fees depending on the differences between deposited amounts and the current state of the pool. The larger the delta, the higher the fees. However, it is not easy for a liquidity provider to know the actual fees payed. Similarly, the function `remove_liquidity_single` charges fees but it is not explicit to the caller.

9.5 Decay of Voting Weight on Transfers

Note **Version 1**

Voting weight is computed by a asymptotic function that depends on the amount of shares and the time they have been staked. The variable `Weight.t` is adjusted (lowered) when a position receives new shares such that the voting weight before and after the transfer remains the same. However, when transferring out tokens the variable `Weight.t` is not modified.

The side effect of this behavior is that if a position with a voting weight `v1` receives `x` shares and then transfers out the same amount `x` shares, ends up with a lower voting weight `v2` (`v2 < v1`) although the number of staked shares has not changed. Furthermore, transfers also affect the global voting weight as transfers decrease voting weight of individual positions.

9.6 Large Ratio Drops for an Asset Break Pool Composition

Note **Version 1**

Pool implements a safety mechanism to ensure that the portfolio composition of underlying assets is according to specified parameters. Each asset in the Pool has a target weight (a range) associated with it. User operations, like swap, deposit, or withdraw, that change the asset balances in the Pool are permitted only if they do not move the actual weights of the involved assets outside the specified ranges. However, the Pool composition changes also when updating rates as asset balances change. The safety mechanism is not enforced in such changes of Pool composition.

While the mechanism of safety bands helps to maintain the desired composition of the Pool when all assets have a backing ratio as expected (around 1 ETH), they do no limit the value loss of the Pool when the ratio of one asset drops significantly (e.g., goes towards 0). A lower rate for an asset results in a lower virtual balance for the asset, which lowers its weight in the Pool, therefore enabling trades that transfer the cheaper asset into the Pool and transfer out other assets. If the rate of one asset drops significantly (e.g., due to a hack), the Pool should be paused immediately, before the new ratio is published by the respective provider, to prevent traders from selling the worthless asset to the Pool.

9.7 Precision of Packed Weights

Note **Version 1**

The precision of weight variables passed as arguments in function is 18 decimals. This precision is lost when the variables are stored as packed in storage due to space limitations. Variables `weight`, `target`, `lower` and `upper` are limited to `20` bits, therefore they can store values with a precision of 6 digits only. The `management` account should take into consideration this behavior when setting the respective parameters of `Pool`.

For instance, the function `set_ramp` does not enforce a lower bound on the values passed in the array `_weights`. If a value smaller than `10**12` is passed as target weight for an asset, the function `_pack_weight` will store `0`. In this case, the ramp cannot complete and main functionalities of the Pool stop working as the function `_update_weights` calls function `_calc_vb_prod` which requires the weight of each asset to be non-zero: `assert weight > 0`.

9.8 Sandwiching Curve Changes

Note Version 1

There are many ways a Curve can significantly change its shape. A prominent attack example is the sandwich attack on Curve when the amplification factor is changed. Therefore, these important changes are ramped (split in smaller changes over a defined time) to minimize the revenue of sandwiching these changes. Yearn also implements ramping for weight changes and amplification factor. However, ramping does not guarantee that an attack is not profitable. As Yearn does have more potential ways to change their Curve than e.g., the original Curve by Curve finance, this risk is increased.

Ramping can be initiated only by the trusted account `management` which should carefully select the parameters `_amplification`, `_weights`, `_duration` and `ramp_step`. First, as `_amplification` represents the factor `A * f^n` and `f` depends on weights, both target `_amplification` and target `_weights` should be chosen such that they stay in line with each-other in intermediary steps of ramping. Otherwise, `management` should execute each step as a separate ramp. For instance, transitioning from a pool with weights (10%, 20%, 70%) to a pool with weights (60%, 30%, 10%) introduces an error in the amplification factor of up to 49% in the intermediary steps of the ramping. The error gets higher for more excessive changes. For example, transitioning from a pool with weights (1%,99%) to weights (99%,1%) introduces an error up to 72% in the intermediary steps of the ramping.

Finally, `_duration` and `ramp_step` should be carefully chosen such that each step of ramping does not change the curve significantly. Any update of the `ramp_step` should take into consideration the ongoing ramp.

9.9 Staking Does Not Lock Tokens

Note Version 1

Users holding `yETH` can stake their tokens into the contract `Staking`. The contract does not lock staked tokens and there is no time restriction to withdraw them. The only incentive to keep tokens staked is the increasing voting weight. Hence, `yETH` holders might have a stronger incentive to stake their tokens if the `Pool` generates rewards and withdraw if there are losses.

9.10 Supply Updates in Pool

Note Version 1

The rate update of underlying assets can be triggered explicitly by calling the function `update_rates` or it gets triggered when sensitive operations are executed, e.g., adding/removing liquidity or swaps. The rate update changes the composition of the virtual balances of assets in the pool. The new `supply` is then computed, and `Pool` mints or burns tokens to/from the `staking` based on the positive or negative delta.

Updates of the `supply` provide different incentives to users. For instance, if new rates are published and they lower `supply`, an existing LP can profit by sandwiching the transaction that triggers the `supply` update by withdrawing their tokens first and then deposing again after the `supply` gets updated.

On the other case, if `supply` is going to be increased, LPs have an additional incentive to stake their `yETH` to claim the respective rewards (subject to delays).