PUBLIC

Code Assessment

of the yCRV and ZapYCRV Smart Contracts

6 September, 2022

Produced for

yearn

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Yearn,

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 yCRV and ZapYCRV according to Scope to support you in forming an opinion on their security risks.

For this assessment Yearn redesigned the Yearn Vault system for voting escrow locked CRV tokens. This new `yCRV` Vault allows unidirectional conversion of `CRV` and old `yveCRV` tokens into new `yCRV` Vault tokens. Another contract is `ZapYCRV` - a helper converter that allows conversions between different `CRV` and `yCRV` related tokens. Using it, users can convert allowed tokens into `lp-yCRV` and `st-yCRV` - Curve StableSwap CRV/yCRV LP token and staked autocompounded yCRV token versions.

The most critical subjects covered in our audit are solvency, functional correctness and compatibility with external systems. Security regarding system solvency is high after the fix of a critical bug that caused users not to receive their tokens, see LPYCRV Outputs Not Transferred to User. Functional correctness is high. Compatibility with external systems is satisfactory, due to a justified potential delay of CRV tokens being locked, see CRV Not Locked When Used to Mint YCRV.

The general subjects covered are specification and error handling. Documentation and Specification are outdated and require significant extension, since system intentions and features are not fully describe. Error handling is extensive.

In summary, we find that the codebase provides a satisfactory level of security. Discovered findings have been fixed or their risks were accepted by the Yearn. We advice revisiting and addressing the issues for witch the risks were accepted. In addition, prior the deployment, we suggest using extensive testing techniques like property based testing and forked mainnet testing to avoid potential problems with the upgrade of the `yveCRV` system.

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

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

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

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

Critical-Severity Findings	1
• Code Corrected	1
High-Severity Findings	1
• Code Corrected	1
Medium-Severity Findings	2
• Code Corrected	1
• Risk Accepted	1
Low-Severity Findings	3
• Code Corrected	2
• Risk Accepted	1

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 source code files inside the yCRV and ZapYCRV repository based on the documentation provided and refined in written communications. The following files were part of the assessment:

1. `contracts/yCRV.vy`
2. `contracts/ZapYCRV-addresses.vy`

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

V	Date	Commit Hash	Note
1	03 August 2022	904d33c7371782898b060509ee7fd065b16acfe4	Initial Version
2	19 August 2022	ce303aee5c6393926138ad1e0bd457109af9f853	Version 2
3	30 August 2022	0552d1a9366084c316a2b8f883fa36334a77a032	Version 3

For the vyper smart contracts, the compiler version `0.3.3` was initially used. The version was upgraded to `0.3.6` in Version 3 .

2.1.1 Excluded from scope

Any contract inside the repository that are not mentioned in `Scope` are not part of this assessment. All external libraries and imports are assumed to behave correctly according to their high-level specification, without unexpected side effects.

2.1.2 Assumptions

Assessment was performed when the development process was not yet concluded. Thus we relied on certain assumptions.

- In `ZapYCRV` smart contract, STYCRV and LPYCRV are assumed to be Yearn Vaults v2.
- In `ZapYCRV` smart contract, POOL is assumed to be Curve Finance StableSwap plain pool contract, created through the Curve Factory contract, with user interface at curve.fi/factory/create.

For the update of the `yveCRV` to `yCRV`, we assume following steps are done in order:

1. Final claim of fees is made from `yveCRV`.
2. All strategies updated with `.setProxy(new strategy proxy address)`.
3. New `st-yCRV` Strategy is assigned as `feeRecipient` via call to `StrategyProxy.setFeeRecipient`.

This list of steps is not complete, there can be more steps done, but those 3 steps are relevant to the contracts from Scope.

All Strategy Managers are assumed to be trusted and well behaving. During the update process, we assume that `st-yCRV` Strategy Manager won't trigger the weekly fee claim function in unfair way. First fee harvest after update from yveCRV to st-yCRV is assumed to happen after users were notified in advance and given time for migration.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

At the end of this report section we have added subsections for each of the changes accordingly to the versions.

Assessment was performed on two smart contracts:

1. yCRV
2. ZapYCRV

These contracts are planned to be part of Yearn strategies that are integrated with Curve Finance protocol CRV reward system.

Curve CRV tokens can be locked for up to 4 years in Curve VotingEscrow contract. Such locking mints veCRV tokens. These non-transferrable tokens receive a share of trading fees from Curve protocol. In addition, veCRV tokens can boost rewards on liquidity provided to Curve protocol when token holders participate in governance, which allows to direct the CRV rewards towards selected pools.

Currently, Yearn allows Curve CRV tokens to be locked as veCRV using the yveCRV yVault contract. This mints yveCRV tokens that represent the locked tokens, but which are also transferrable.

veCRV tokens are held by Yearn CurveYCRVVoter contract. Fees received by it are forwarded to the feeRecipient with the help of StrategyProxy contract. Contract yveCRV is currently the feeRecipient inside the StrategyProxy.

2.2.1 Contract yCRV

New yCRV contract is a replacement for yveCRV yVault. The yCRV.burn_to_mint function allows users to "burn" (lock forever) yveCRV tokens, and mint new yCRV tokens instead at a rate 1:1. In addition, yCRV tokens can be minted by locking CRV tokens with the yCRV.mint function.

This contract has a privileged role address - admin. The admin can call the yCRV.sweep and yCRV.sweep_yvecrv functions, that transfer accidentally sent tokens to admin. The sweep function can also be used by admin during the migration period from yveCRV to yCRV to redeem the 3CRV rewards that are forwarded to the yveCRV locked in the yCRV contract.

2.2.2 Contract ZapYCRV

Contract ZapYCRV can be seen as a universal converter between different tokens. There are 3 main functions:

- zap This function converts an amount of _input_token into _output_token. Output tokens are sent to _recipient. This functions by design does not emit events, since data can be derived from Transfer events.
- relative_price This function returns an estimation of the conversion of an amount of _input_token into _output_token. This functions assumes that all AMM pools that are used during the conversion are balanced. In addition this function does not account neither for slippage nor for fees during the conversions. In combination with calc_expected_out this function can be used to estimate the output amount of tokens resulting from the conversion.
- calc_expected_out This function returns an estimation of the conversion of an amount of _input_token into _output_token. Compared to the relative_price function, this function accounts for slippage and for the liquidity of the Curve StableSwap pools, but not for fees.

The following tokens can be inputs for these 3 functions:

- yveCRV
- yvBOOST - Compounded version of yveCRV, where the interest is automatically reinvested.
- CRV
- CVXCRV - Convex protocol tokenized version of veCRV.
- yCRV
- lp-yCRV Vault - Assumed to be a yVault for Curve yCRV/CRV StableSwap pool LP tokens.
- st-yCRV Vault - Assumed to be a yVault for autocompounded Curve Admin fees.

The following tokens can be outputs:

- yCRV
- lp-yCRV Vault
- st-yCRV Vault

In the new version of Yearn veCRV integration, StrategyProxy will be able to change the feeRecipient. New feeRecipient is assumed to be the Yearn Strategy Contract for the st-yCRV Vault.

Similar to yCRV, there is a privileged admin role that can call the ZapYCRV.sweep function. ZapYCRV is stateless and should not hold balance of any token outside of transaction execution.

2.2.3 Differences in Version 3

The contracts logic stay the same, but the admin role is renamed to sweep_recipient, as the only privileged action that can be performed on these contracts is the sweeping of excess token balance.

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

- CRV Not Locked When Used to Mint YCRV Risk Accepted

Low-Severity Findings	1

- Trades During ZapYCRV.zap Conversions Risk Accepted

5.1 CRV Not Locked When Used to Mint YCRV

Correctness Medium Version 1 Risk Accepted

When YCRV is minted with the `mint()` function, CRV is not locked.

In yveCRV, CRV is locked upon minting. In `YCRV.mint(...)` it is not locked immediately, but a separate call to `StrategyProxy.lock()` is needed.

```
assert ERC20(CRV).transferFrom(msg.sender, VOTER, amount)  # dev: no allowance
self._mint(_recipient, amount)
log Mint(msg.sender, _recipient, False, amount)
return amount
```

Not locking the CRV immediately in the CRV voting escrow implies a mismatch between the total supply of YCRV and the effective voting power and total rewards of VOTER. It also imposes increased trust requirements towards governance, which might sweep the not yet locked CRV from the VOTER.

Risk accepted

Yearn states:

```
Locking CRV is gas intensive. Decision was made to have locking occur at
some periodic interval via external process rather than burden each user
with gas costs.
```

5.2 Trades During `ZapYCRV.zap` Conversions

Security | Low | Version 1 | Risk Accepted

The `ZapYCRV.zap` function can involve multiple Curve pools during the conversion.

First, `CRV` -> `LPYCRV` conversions will involve up to 2 trades in LPYCRV pool:

1. Trade of all CRV to yCRV
2. Trade of some yCRV to CRV, during the unbalanced deposit into the pool

Compared to trade of some CRV to yCRV and a balanced deposit, the 2 trades double pay the fees.

Second, in the case when CVXCRV is an input, these 2 trades are preceded by a trade on CVXCRVPOOL.

Please note, that due to number of pools and exchanges during the conversion process the `min_out` argument can be hard to specify precisely. In addition, imprecise `min_out` specified would allow 3rd parties to front run the `zap`.

Risk accepted

Yearn states:

```
Realize that for some specific paths, this can be inefficient. However, hardcoding
paths will lead to more contract complexity and overall gas consumption (including
for users who's zap path touches neither of these tokens) which we view as
undesirable. We agree that users can potentially lose more due to swap fees, but
ultimately most of those same fees get realized to the pool LPs, helping to repay
them over time.
```

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	1

- LPYCRV Outputs Not Transferred to User Code Corrected

High-Severity Findings	1

- Incorrect relative_price When Input Is Not Legacy and Output Is LPYCRV Code Corrected

Medium-Severity Findings	1

- ZapYCRV _min_out LPYCRV Limit Code Corrected

Low-Severity Findings	2

- ERC20 Return Values Not Checked Code Corrected
- ZapYCRV.zap Natspec Code Corrected

6.1 LPYCRV Outputs Not Transferred to User

Correctness Critical Version 1 Code Corrected

In the `zap` function of `ZapYCRV`, converting to `LPYCRV` as `_output_token` will not transfer `LPYCRV` to the user but leave it in the ZapYCRV contract instead.

When `LPYCRV` is the output token, `ZapYCRV` should first deposit `YCRV` as liquidity in the `POOL` StableSwap pool, receiving the `POOL` liquidity token. The `POOL` liquidity token should then be deposited in the `LPYCRV` vault and the issued shares transferred to the user.

The following line of code in `_convert_to_output` is responsible for the specified logic.

```
amount_out: uint256 = Vault(LPYCRV).deposit(self._lp([0, amount], _min_out,
    _recipient))
```

which calls the `self._lp(...)` function, defined as

```
@internal def _lp(_amounts: uint256[2], _min_out: uint256, _recipient:
address) -> uint256:
    return Curve(POOL).add_liquidity(_amounts, _min_out)
```

The `_recipient` argument is passed to the `_lp` function, but never used. The `_lp` function doesn't actually need the `_recipient` argument, because `ZapYCRV` will still need to deposit the liquidity token into the `LPYCRV` vault.

The `Vault(LPYCRV).deposit` function is called without specifying the `recipient` argument, which therefore defaults to `msg.sender`, which is the `ZapYCRV` contract in the context of the `deposit` call. Finally the `zap` function returns and the issued shares of `LPYCRV` are never transferred to the user, but left to `ZapYCRV` instead.

Code corrected

The `recipient` argument of the `_lp` function has been removed.

```
@internal def _lp(_amounts: uint256[2]) -> uint256:
        return Curve(POOL).add_liquidity(_amounts, 0)
```

A `recipient` value is now specified in the `deposit` call to the `LPYCRV` vault in the `_convert_to_output` function.

```
amount_out: uint256 = Vault(LPYCRV).deposit(self._lp([0, amount]), _recipient)
```

6.2 Incorrect `relative_price` When Input Is Not Legacy and Output Is LPYCRV

Correctness High Version 1 Code Corrected

In `relative_price` the relative price for the `POOL` liquidity token is returned instead of the relative price of `LPYCRV` when `_input_token` is not a legacy token and `_output_token` is `LPYCRV`.

The relative price for `_input_token` not in `legacy_tokens` and `_output_token` equal to `LPYCRV` is computed as follow:

```
return amount * 10 ** 18 / Curve(POOL).get_virtual_price()
```

This doesn't take into account that the output token is `LPYCRV` and not `POOL`, so the `POOL` tokens need to be used to purchase `LPYCRV` shares at price `Vault(LPYCRV).pricePerShare()`.

When `_input_token` is a legacy token, it is computed correctly as follows:

```
lp_amount: uint256 = amount * 10 ** 18 / Curve(POOL).get_virtual_price()
return lp_amount * 10 ** 18 / Vault(LPYCRV).pricePerShare()
```

Code corrected

```
return amount * 10 ** 18 / Curve(POOL).get_virtual_price()
```

is replaced with

```
lp_amount: uint256 = amount * 10 ** 18 / Curve(POOL).get_virtual_price()
return lp_amount * 10 ** 18 / Vault(LPYCRV).pricePerShare()
```

6.3 ZapYCRV `_min_out` LPYCRV Limit

Design Medium Version 1 Code Corrected

In `ZapYCRV.zap`, the `_min_out` argument of the `zap` function asserts a lower bound on the amount of output token received by the user. When `_output_token` is `LPYCRV` it incorrectly asserts the amount of liquidity tokens issued as an intermediate conversion step by `Curve(POOL).add_liquidity`.

In the LPYCRV branch of `_convert_to_output`, `_min_out` gets first passed to `_lp()`, which uses it as a lower bound to the amount of liquidity tokens issued by `Curve(POOL).add_liquidity()`

```
@internal
def _lp(_amounts: uint256[2], _min_out: uint256, _recipient: address) -> uint256:
    return Curve(POOL).add_liquidity(_amounts, _min_out)
```

It is then used again as a lower bound for `amount_out` issued by `Vault(LPYCRV).deposit()`.

```
amount_out: uint256 = Vault(LPYCRV).deposit(self._lp([0, amount], _min_out, _recipient))
assert amount_out >= _min_out # dev: min out
```

This basically makes `_min_out` used for limit of LPYCRV vault shares and POOL LP shares. Due to how the share values are computed, in the general case they will be not worth 1:1. Thus, `_min_out` as a limit is not practical.

Code corrected

The `_min_out` argument of the `_lp` function has been removed. Thus it is not used as a lower bound to the amount of liquidity tokens issued by `Curve(POOL).add_liquidity()` anymore.

```
@internal
def _lp(_amounts: uint256[2]) -> uint256:
    return Curve(POOL).add_liquidity(_amounts, 0)
```

Yearn notes:

```
Hardcode the minimum to 0 in add_liquidity, as we will rely on subsequent check to
compare user inputted min_out.
```

6.4 ERC20 Return Values Not Checked

Correctness Low Version 1 Code Corrected

According to EIP-20, Callers MUST NOT assume that false is never returned. However, not all calls to ERC20 assert that true is returned. `ZapYCRV` and `yCRV` do not check bool `success` values for calls to `ERC20.approve` and `ERC20.transfer`. Even though in most cases the contracts are known in advance and it is safe not to check this value, new features and codebase reuse can lead to potential problems.

In function `sweep` in both contracts the return value of `ERC20.transfer` can be missing, if for example USDT is used. In that case the call will fail.

Code corrected

Asserts have been added to the `approve` and `transfer` calls to make sure that true is returned.

Compiler version has been increased to `vyper 0.3.6` in order to use the external call keyword argument `default_return_value=True`, which ensures that `transfer` calls do not revert when calling non EIP-20 compliant tokens such as `USDT` which do not return a boolean value.

6.5 `ZapYCRV.zap` Natspec

Correctness **Low** **Version 1** **Code Corrected**

The `@param _input_token` for `zap` function does not describe that `cvxCRV` can be used as input token.

Code corrected

The `cvxCRV` has been added to the `@param _input_token` in the `zap` function's natspec.

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 Some but Not All Fees Are Accounted for in `calc_expected_out`

Note **Version 1**

The natspec of `calc_expected_out` says that fees are not accounted for when computing the result. But actually, almost in all cases, when the Curve pools are used, they are accounted. The only case when the fees are not taken into account is when the output token is `LPYCRV`. Then the deposit of `YCRV` in `POOL` is simulated with `Curve(POOL).calc_token_amount(...)`, which does not account for the fees.

7.2 `ZapYCRV` Curve StableSwap Token Indices Sanity Check

Note **Version 1**

`ZapYCRV` contract uses `POOL` contract, that is assumed to be Curve Finance StableSwap contract. In StableSwap the tokens can be in any order. The `yCRV/CRV` pool is not yet deployed. Thus, the assumption that CRV will be index 0 and yCRV will have index 1 might be violated. A sanity check in the constructor of `ZapYCRV` contract can prevent human and misconfiguration errors and lower the costs associated with redeployment.

7.3 `ZapYCRV` Return Values Ignored

Note **Version 1**

In the `ZapYCRV._zap_from_legacy` function the return value of calls to `IYCRV(YCRV).burn_to_mint` are ignored. Return value is assumed to be same as the `amount` argument that the function takes. However, in case when the `amount` is equal to MAX_UINT256, the `burn_to_mint` might return other value. In the current version such situation should never happen, because this case is handled by the `zap` function itself. In `ZapYCRV._zap_from_legacy`, `amount` should never be equal to MAX_UINT256. However use of return value will prevent potential bugs in case of code reuse or if new features are added.

7.4 yCRV as an ERC20 Implementation

Note **Version 1**

There are 2 things we would like to note regarding the `yCRV` token.

1. The `approve` function has a known race condition attack vector described here

2. The `transferFrom` function does not emit `Approval` event. While this is compliant with specification, one cannot reconstruct the state of user allowances based only on events, since `transferFrom` does not emit any special events that show that approval was used.