YIELD

Yield-Convex contest
Findings & Analysis Report

2022-03-04

Table of contents

- Gas Optimizations (18)
- Disclosures

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 code contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the code contest outlined in this document, C4 conducted an analysis of Yield-Convex contest smart contract system written in Solidity. The code contest took place between January 28—January 30 2022.

Wardens

23 Wardens contributed reports to the Yield-Convex contest:

1. leastwood
2. kenzo
3. WatchPug (jtp and ming)
4. Dravee
5. sirhashalot
6. pants
7. GeekyLumberjack
8. robee
9. TomFrenchBlockchain
10. throttle
11. Fitraldys
12. Funen

13. defsec
14. rfa
15. 0x1f8b
16. cccz
17. hack3r-0m
18. hyh
19. ye0lde
20. IIIIIII
21. bitbopper
22. Tomio

This contest was judged by Alex the Entreprenerd.

Final report assembled by liveactionllama and CloudEllie.

Summary

The C4 analysis yielded an aggregated total of 10 unique vulnerabilities and 37 total findings. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity, 2 received a risk rating in the category of MEDIUM severity, and 6 received a risk rating in the category of LOW severity.

C4 analysis also identified 9 non-critical recommendations and 18 gas optimizations.

Scope

The code under review can be found within the C4 Yield-Convex contest repository, and is composed of 4 smart contracts and 1 library written in the Solidity programming language and includes 741 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on [the C4 website](#).

High Risk Findings (2)

[H-01] Malicious Users Can Duplicate Protocol Earned Yield By Transferring `wCVX` Tokens To Another Account

Submitted by leastwood, also found by kenzo

`ConvexYieldWrapper.sol` is a wrapper contract for staking convex tokens on the user's behalf, allowing them to earn rewards on their deposit. Users will interact with the `Ladle.sol` contract's `batch()` function which:

- Approves Ladle to move the tokens.
- Transfers the tokens to `ConvexYieldWrapper.sol`.
- Wraps/stakes these tokens.
- Updates accounting and produces debt tokens within `Ladle.sol`.

During `wrap()` and `unwrap()` actions, `_checkpoint()` is used to update the rewards for the `from_` and `to_` accounts. However, the [reference](#) contract implements a `_beforeTokenTransfer()` function which has been removed from Yield Protocol's custom implementation.

As a result, it is possible to transfer `wCVX` tokens to another account after an initial checkpoint has been made. By manually calling `user_checkpoint()` on the new account, this user is able to update its deposited balance of the new account while the sender's balance is not updated. This can be repeated to effectively replicate a user's deposited balance over any number of accounts. To claim yield generated by the protocol, the user must only make sure that the account calling `getReward()` holds the tokens for the duration of the call.

Proof of Concept

The exploit can be outlined through the following steps:

- Alice receives 100 `wCVX` tokens from the protocol after wrapping their convex tokens.
- At that point in time, `_getDepositedBalance()` returns 100 as its result. A checkpoint has also been made on this balance, giving Alice claim to her fair share of the rewards.
- Alice transfers her tokens to her friend Bob who then manually calls `user_checkpoint()` to update his balance.
- Now from the perspective of the protocol, both Alice and Bob have 100 `wCVX` tokens as calculated by the `_getDepositedBalance()` function.
- If either Alice or Bob wants to claim rewards, all they need to do is make sure the 100 `wCVX` tokens are in their account upon calling `getReward()`. Afterwards, the tokens can be transferred out.

Tools Used

Manual code review. Discussion/confirmation with the Yield Protocol team.

Recommended Mitigation Steps

Consider implementing the `_beforeTokenTransfer()` function as shown in the reference contract. However, it is important to ensure the wrapper contract and collateral vaults are excluded from the checkpointing so they are not considered in the rewards calculations.

alcueca (Yield) confirmed and commented:

> Confirmed. The fact that rewards can be drained also means that users lose on their expected rewards, so I think that Sev 3 is right.

iamsahu (Yield) resolved

Alex the Entreprenerd (judge) commented:

> In systems that track growing rewards, anytime a user balances changes, it's important to recalculate their balances as to properly distribute pending rewards and to influence the future-rate at which rewards will be distributed (process generally called `accruing`)

> In the case of the ConvexYieldWrapper, the warden has shown that because the `wCVX` token doesn't perform a `_checkpoint` on each transfer, a malicious attacker could repeatedly transfer their tokens in order to reuse the same balance in multiple accounts, effectively sybil attacking the protocol.

> The fix seems to be straightforward, however the impact of the finding breaks the accounting of the protocol, as such I believe High Severity to be appropraite

Alex the Entreprenerd (judge) commented:

> The sponsor has mitigated in a subsequent PR by overriding the `_transfer` function

[H-02] Malicious Users Can Transfer Vault Collateral To Other Accounts To Extract Additional Yield From The Protocol

Submitted by leastwood

`ConvexYieldWrapper.sol` is a wrapper contract for staking convex tokens on the user's behalf, allowing them to earn rewards on their deposit. Users will interact with the `Ladle.sol` contract's `batch()` function which:

- Approves Ladle to move the tokens.
- Transfers the tokens to `ConvexYieldWrapper.sol`.

- Wraps/stakes these tokens.
- Updates accounting and produces debt tokens within `Ladle.sol`.

`_getDepositedBalance()` takes into consideration the user's total collateral stored in all of their owned vaults. However, as a vault owner, you are allowed to give the vault to another user, move collateral between vaults and add/remove collateral. Therefore, it is possible to manipulate the result of this function by checkpointing one user's balance at a given time, transferring ownership to another user and then create a new checkpoint with this user.

As a result, a user is able to generate protocol yield multiple times over on a single collateral amount. This can be abused to effectively extract all protocol yield.

Proof of Concept

Consider the following exploit scenario:

- Alice owns a vault which has 100 tokens worth of collateral.
- At that point in time, `_getDepositedBalance()` returns 100 as its result. A checkpoint has also been made on this balance, giving Alice claim to her fair share of the rewards.
- Alice then calls `Ladle.give()`, transferring the ownership of the vault to Bob and calls `ConvexYieldWrapper.addVault()`.
- Bob is able to call `user_checkpoint()` and effectively update their checkpointed balance.
- At this point in time, both Alice and Bob have claim to any yield generated by the protocol, however, there is only one vault instance that holds the underlying collateral.

https://github.com/code-423n4/2022-01-yield/blob/main/contracts/ConvexYieldWrapper.sol#L100-L120

```
    function _getDepositedBalance(address account_) internal view ov
        if (account_ == address(0) || account_ == collateralVault) {
            return 0;
        }

        bytes12[] memory userVault = vaults[account_];
```

```
        //add up all balances of all vaults registered in the wrappe
        uint256 collateral;
        DataTypes.Balances memory balance;
        uint256 userVaultLength = userVault.length;
        for (uint256 i = 0; i < userVaultLength; i++) {
            if (cauldron.vaults(userVault[i]).owner == account_) {
                balance = cauldron.balances(userVault[i]);
                collateral = collateral + balance.ink;
            }
        }

        //add to balance of this token
        return _balanceOf[account_] + collateral;
    }
```

Tools Used

Manual code review. Discussion/confirmation with the Yield Protocol team.

Recommended Mitigation Steps

Ensure that any change to a vault will correctly checkpoint the previous and new vault owner. The affected actions include but are not limited to; transferring ownership of a vault to a new account, transferring collateral to another vault and adding/removing collateral to/from a vault.

iamsahu (Yield) confirmed

Alex the Entreprenerd (judge) commented:

> The warden identified a way to sidestep the accounting in the `ConvexYieldWrapper`.
>
> Because `ConvexYieldWrapper` takes lazy accounting, transferring vaults at the `Ladle` level allows to effectively register the same vault under multiple accounts, which ultimately allow to steal more yield than expected.
>
> While the loss of yield can be classified as a medium severity, the fact that the warden was able to break the accounting invariants of the `ConvexYieldWrapper` leads me to raise the severity to high

> Ultimately mitigation will require to `_checkpoint` also when vault operations happen (especially transfer), this may require a rethinking at the Ladle level as the reason why the warden was able to sidestep the checkpoint is because the `Ladle` doesn't notify the `Wrapper` of any vault transfers

alcueca (Yield) commented:

> Yes, that's right. To fix this issue we will deploy a separate Ladle to deal specifically with convex tokens. The fix will probably involve removing `stir` and `give` instead of notifying the wrapper, but we'll see.

Medium Risk Findings (2)

[M-01] Oracle data feed is insufficiently validated

Submitted by throttle, also found by 0x1f8b, cccz, defsec, hack3r-0m, hyh, kenzo, leastwood, sirhashalot, TomFrenchBlockchain, WatchPug, and ye0lde

Price can be stale and can lead to wrong `quoteAmount` return value

Proof of Concept

Oracle data feed is insufficiently validated. There is no check for stale price and round completeness. Price can be stale and can lead to wrong `quoteAmount` return value

```
    function _peek(
        bytes6 base,
        bytes6 quote,
        uint256 baseAmount
    ) private view returns (uint256 quoteAmount, uint256 updateTime)
        ...

        (, int256 daiPrice, , , ) = DAI.latestRoundData();
        (, int256 usdcPrice, , , ) = USDC.latestRoundData();
        (, int256 usdtPrice, , , ) = USDT.latestRoundData();

        require(
            daiPrice > 0 && usdcPrice > 0 && usdtPrice > 0,
```

```
            "Chainlink pricefeed reporting 0"
        );

        ...
    }
```

Recommended Mitigation Steps

Validate data feed

```
    function _peek(
        bytes6 base,
        bytes6 quote,
        uint256 baseAmount
    ) private view returns (uint256 quoteAmount, uint256 updateTime)
        ...
        (uint80 roundID, int256 daiPrice, , uint256 timestamp, uint8
        require(daiPrice > 0, "ChainLink: DAI price <= 0");
        require(answeredInRound >= roundID, "ChainLink: Stale price'
        require(timestamp > 0, "ChainLink: Round not complete");

        (roundID, int256 usdcPrice, , timestamp, answeredInRound) =
        require(usdcPrice > 0, "ChainLink: USDC price <= 0");
        require(answeredInRound >= roundID, "ChainLink: Stale USDC p
        require(timestamp > 0, "ChainLink: USDC round not complete")

        (roundID, int256 usdtPrice, , timestamp, answeredInRound) =
        require(usdtPrice > 0, "ChainLink: USDT price <= 0");
        require(answeredInRound >= roundID, "ChainLink: Stale USDT p
        require(timestamp > 0, "ChainLink: USDT round not complete")

        ...
    }
```

iamsahu (Yield) confirmed and resolved

Alex the Entreprenerd (judge) commented:

> When using Chainlink Price feeds it is important to ensure the price feed data was updated recently. While getting started with chainlink requires just one line of code, it is best to add additional checks for in production environments.

> I believe the finding to be valid and Medium severity to be appropriate.

> The sponsor has mitigated in a subsequent PR.

[M-02] Rewards distribution can be disrupted by a early user

Submitted by WatchPug

https://github.com/code-423n4/2022-01-yield/blob/e946f40239b33812e54fafc700eb2298df1a2579/contracts/ConvexStakingWrapper.sol#L206-L224

```
    function _calcRewardIntegral(
        uint256 _index,
        address[2] memory _accounts,
        uint256[2] memory _balances,
        uint256 _supply,
        bool _isClaim
    ) internal {
        RewardType storage reward = rewards[_index];

        uint256 rewardIntegral = reward.reward_integral;
        uint256 rewardRemaining = reward.reward_remaining;

        //get difference in balance and remaining rewards
        //getReward is unguarded so we use reward_remaining to keep
        uint256 bal = IERC20(reward.reward_token).balanceOf(address(
        if (_supply > 0 && (bal - rewardRemaining) > 0) {
            rewardIntegral = uint128(rewardIntegral) + uint128(((bal
            reward.reward_integral = uint128(rewardIntegral);
        }
```

`reward.reward_integral` is `uint128`, if a early user mint (wrap) just `1` Wei of `convexToken`, and make `_supply == 1`, and then tranferring `5e18` of `reward_token` to the contract.

As a result, `reward.reward_integral` can exceed `type(uint128).max` and overflow, causing the rewards distribution to be disrupted.

Recommendation

Consider `wrap` a certain amount of initial totalSupply, e.g. `1e8`, and never burn it. And consider using uint256 instead of uint128 for `reward.reward_integral`. Also, consdier lower `1e20` down to `1e12`.

alcueca (Yield) confirmed but disagreed with severity and commented:

> Assets are not a direct risk if it is the first user disrupting the contract. We would need to redeploy better code, but that's it. I suggest this is reported as medium, as merits for a DoS attack.

> As suggested elsewhere, the right solution if uint128 is to be used in storage, is to cast up to uint256 for calculations, and then back to uint128 to store again.

iamsahu (Yield) resolved

Alex the Entreprenerd (judge) changed severity from High to Medium and commented:

> The warden identified a way an early depositor can grief the system, I believe the finding to be valid, and because:
>
> - It would conditionally disrupt the system
> - It would "brick the contract" at the loss of the griefer
> - No additional funds would be at risk
>
> I believe medium severity to be appropriate

Low Risk Findings (6)

- [L-01] Unsafe uint128 casting may overflow *Submitted by sirhashalot*
- [L-02] Cvx3CrvOracle earned function calculates cvx wrongly if pool claimed indirectly *Submitted by kenzo*
- [L-03] `ConvexStakingWrapper.sol` : unused `nonReentrant` modifier *Submitted by Dravee*
- [L-04] Cvx3CrvOracle returns 0 for small baseAmount *Submitted by kenzo*

- [L-05] Unbounded loop on array can lead to DoS *Submitted by pants*
- [L-06] Only passing in one depositedBalance in _checkpointAndClaim() *Submitted by GeekyLumberjack*

Non-Critical Findings (9)

- [N-01] Gas: Unused Named Returns *Submitted by Dravee, also found by throttle and WatchPug*
- [N-02] Unused imports *Submitted by robee*
- [N-03] Typos *Submitted by yeOlde*
- [N-04] Comment missing function parameter *Submitted by sirhashalot, also found by kenzo*
- [N-05] Missing commenting *Submitted by robee*
- [N-06] `ConvexStakingWrapper.sol` : related data should be grouped in a struct *Submitted by Dravee*
- [N-07] `ConvexStakingWrapper.sol` : `AccessControl` capabilities aren't used *Submitted by Dravee*
- [N-08] Race condition in approve() *Submitted by cccz*
- [N-09] Lack of important event *Submitted by 0x1f8b*

Gas Optimizations (18)

- [G-01] `ConvexYieldWrapper#removeVault()` `found` is redundant *Submitted by WatchPug, also found by throttle*
- [G-02] `ConvexYieldWrapper.sol` Redundant code *Submitted by WatchPug, also found by Fitraldys*
- [G-03] Adding unchecked directive can save gas *Submitted by WatchPug*
- [G-04] Gas saving using immutable *Submitted by 0x1f8b, also found by defsec, Funen, IIIIIII, throttle, TomFrenchBlockchain, Tomio, and WatchPug*
- [G-05] `ConvexStakingWrapper.sol#` Switching between 1, 2 instead of 0, 1 is more gas efficient *Submitted by WatchPug, also found by bitbopper, Funen, and throttle*

- [G-06] Avoid unnecessary arithmetic operations and storage reads can save gas *Submitted by WatchPug*
- [G-07] Gas in `Cvx3CrvOracle.sol: _peek()` : `ethId` and `cvx3CrvId` should get cached *Submitted by Dravee, also found by robee and TomFrenchBlockchain*
- [G-08] Unnecessary check on quote in Cvx3CrvOracle *Submitted by TomFrenchBlockchain*
- [G-09] Gas: No need to initialize variables with default values *Submitted by Dravee, also found by robee, and throttle*
- [G-10] Prefix increments are cheaper than postfix increments *Submitted by robee, also found by 0x1f8b, defsec, Dravee, Funen, IIIIIII, and throttle*
- [G-11] Perform math inside code branch *Submitted by sirhashalot*
- [G-12] Gas: Tight variable packing in `ConvexStakingWrapper.sol` *Submitted by Dravee, also found by robee*
- [G-13] Gas: `> 0` is less efficient than `!= 0` for unsigned integers *Submitted by Dravee, also found by defsec, IIIIIII, and robee*
- [G-14] Unnecessary array boundaries check when loading an array element twice *Submitted by robee*
- [G-15] less gas usage by calling the `TransferHelper` lib directly *Submitted by rfa*
- [G-16] caching curveToken in memory can cost less gas *Submitted by Funen*
- [G-17] calldata is cheaper than memory *Submitted by Fitraldys*
- [G-18] Gas in `ConvexStakingWrapper.sol: _calcRewardIntegral()` : `bal - rewardRemaining` can't underflow *Submitted by Dravee*

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct

formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top