Duality Focus contest
Findings & Analysis Report

2022-05-26

Table of contents

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 audit 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 audit contest outlined in this document, C4 conducted an analysis of the Duality Focus smart contract system written in Solidity. The audit contest took place between April 6—April 8 2022.

Wardens

20 Wardens contributed reports to the Duality Focus contest:

1. rayn
2. cmichel
3. IIIIIII
4. cccz
5. hyh

6. 0x1f8b
7. Sleepy
8. defsec
9. sorrynotsorry
10. Dravee
11. robee
12. 0xDjango
13. 0v3rf10w
14. kebabsec (okkothejawa and FlameHorizon)
15. bugwriter001
16. CertoraInc
17. samruna
18. reassor
19. Chom

This contest was judged by Jack the Pug.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 5 unique vulnerabilities. Of these vulnerabilities, 0 received a risk rating in the category of HIGH severity and 5 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 17 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Duality Focus contest repository, and is composed of 8 smart contracts written in the Solidity programming language and includes 1,172 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/non-critical.

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.

Medium Risk Findings (5)

[M-01] Dysfunctional `CToken._acceptAdmin` due to lack of function to assign `pendingAdmin`

Submitted by rayn, also found by 0x1f8b, cmichel, and Sleepy

CToken.sol#L1379

The implementation of `CToken` in Duality introduced an `_acceptAdmin` function, which presumably should allow changing the `admin`. However, there does not exist a pairing `proposePendingAdmin` function that can propose a new `pendingAdmin`, thus `pendingAdmin` will never be set. This renders the `_acceptAdmin` function useless.

Proof of Concept

`_acceptAdmin` requires `msg.sender` to equal `pendingAdmin`, however, since `pendingAdmin` can never be set, it will always be `address(0)`, making this function unusable.

```
    function _acceptAdmin() external returns (uint256) {
        // Check caller is pendingAdmin and pendingAdmin ≠ address(0
        if (msg.sender != pendingAdmin || msg.sender == address(0))
            return fail(Error.UNAUTHORIZED, FailureInfo.ACCEPT_ADMIN
        }
        // Save current values for inclusion in log
        address oldAdmin = admin;
        address oldPendingAdmin = pendingAdmin;
        // Store admin with value pendingAdmin
        admin = pendingAdmin;
        // Clear the pending value
        pendingAdmin = address(0);
        emit NewAdmin(oldAdmin, admin);
        emit NewPendingAdmin(oldPendingAdmin, pendingAdmin);
        return uint256(Error.NO_ERROR);
    }
```

Tools Used

vim, ganache-cli

Recommended Mitigation Steps

Add a `proposePendingAdmin` function where the current admin can propose successors.

```
    function _proposePendingAdmin(address newPendingAdmin) external
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.PROPOSE_PENI
        }
        address oldPendingAdmin = pendingAdmin;
        pendingAdmin = newPendingAdmin;
        emit NewPendingAdmin(oldPendingAdmin, newPendingAdmin);
        return uint256(Error.NO_ERROR)
    }
```

0xdramaone (Duality Focus) confirmed and commented:

> Acknowledged, we were aware this matches the behavior of actual compound when deployed with CErc20Immutable. We originally thought we would stick with this functionality, but have now opted for enabling admin transitions.

0xdramaone (Duality Focus) resolved

[M-02] `Comptroller._setUniV3LpVault` will always cause in-use uniswapV3 positions to become stuck in `UniV3LpVault`

Submitted by rayn

Comptroller.sol#L1105

`Comptroller._setUniV3LpVault` allows the admin of `Comptroller` to change the accompanying `UniV3LpVault`. However since actions including collateral calculation, uniswapV3 position withdrawal, uniswapV3 collateral liquidation all require `Comptroller` and `UniV3LpVault` to cooperate seamlessly, a change in `Comptroller.uniV3LpVault` would mean all the above actions are no longer performable on existing NFTs.

Proof of Concept

`_setUniV3LpVault` allows changing of `uniV3LpVault`.

```
    function _setUniV3LpVault(IUniV3LpVault newVault) public returns
        ...
        uniV3LpVault = newVault;
        ...
    }
```

However, functions such as `UniV3LpVault.withdrawToken` require `Comptroller` to estimate NFT collateral value. This estimation can only be done when the address

of `UniV3LpVault` matches `Comptroller.uniV3LpVault` as shown in `addNFTCollateral` below.

```
contract UniV3LpVault is IUniV3LpVault {
    ...
    function withdrawToken(...) external override nonReentrant(f
        ...
    }

    modifier avoidsShortfall() {
        _;
        (, , uint256 shortfall) = comptroller.getAccountLiquidit
        require(shortfall == 0, "insufficient liquidity");
    }
    ...
}

contract Comptroller is ComptrollerV3Storage, ComptrollerInterfa
    ...
    function getAccountLiquidity(address account) public view re
        (Error err, uint256 liquidity, uint256 shortfall) = getH
        ...
    }

    function getHypotheticalAccountLiquidityInternal(...) interr
        ...
        addNFTCollateral(account, vars);
        ...
    }

    function addNFTCollateral(address account, AccountLiquidityI
        uint256 userTokensLength = uniV3LpVault.getUserTokensLer
        for (uint256 i = 0; i < userTokensLength; i++) {
            ...
            {
                ...
                address poolAddress = uniV3LpVault.getPoolAddres
                ...
            }
            ...
        }
        ...
    }
}
```

The mutual reliance causes NFT tokens to become stuck. In some cases users can solve this issue by depositing more collateral to cover the shortcoming caused by “disappearing NFTs”. In other cases such as liquidation, the functionality becomes downright broken and unuseable..

Tools Used

vim, ganache-cli

Recommended Mitigation Steps

Remove the option to change `Comptroller.uniV3LpVault` altogether, as this functionality is not really helpful for the overall protocol.
Another way to handle this is to forcefully evict all NFTs before changing the vault. However, this is extremely complex as it would potentially cause users to become severely under-collateralized, and would require more care in tracking and maintaining states.

Oxdramaone (Duality Focus) confirmed and commented:

> We agree, should only allow one setting of the UniV3LpVault (no overwrites). Maintaining as med risk given that this would block user withdrawals.

Jack the Pug (judge) commented:

> Good catch! Some sanity check is needed.

Oxdramaone (Duality Focus) resolved

[M-03] Not calling `approve(0)` before setting a new approval causes the call to revert when used with Tether (USDT)

Submitted by lllllll, also found by cccz and hyh

Some tokens do not implement the ERC20 standard properly but are still accepted by most code that accepts ERC20 tokens. For example Tether (USDT)‘s `approve()`

function will revert if the current approval is not zero, to protect against front-running changes of approvals.

Impact

The code as currently implemented does not handle these sorts of tokens properly when they're a Uniswap pool asset, which would prevent USDT, the sixth largest pool, from being used by this project. This project relies heavily on Uniswap, so this would hamper future growth and availability of the protocol.

Proof of Concept

1. File: contracts/vault*and*oracles/FlashLoan.sol (line 48)

```
IERC20(assets[0]).approve(address(LP_VAULT), amounts[0]);
```

2. File: contracts/vault*and*oracles/FlashLoan.sol (line 58)

```
IERC20(assets[0]).approve(address(LENDING_POOL), amountOwing);
```

3. File: contracts/vault*and*oracles/UniV3LpVault.sol (line 418)

```
IERC20Detailed(params.asset).approve(msg.sender, owedBack);
```

There are other calls to `approve()`, but they correctly set the approval to zero after the transfer is done, so that the next approval can go through.

Recommended Mitigation Steps

Use OpenZeppelin's `SafeERC20`'s `safeTransfer()` instead.

Oxdramaone (Duality Focus) confirmed and commented:

> Agree that we should set approve 0 before all approvals to avoid approval protection reverting and to support assets such as USDT.

Oxdramaone (Duality Focus) resolved

[M-04] Undercollateralized loans possible

Submitted by cmichel

Comptroller.sol#L1491

The `_setPoolCollateralFactors` function does not check that the collateral factor is < 100%.
It's possible that it's set to 200% and then borrows more than the collateral is worth, stealing from the pool.

Recommended Mitigation Steps

Disable the possibility of ever having a collateral factor > 100% by checking:

```
    for (uint256 i = 0; i < pools.length; i++) {
+       require(collateralFactorsMantissa[i] <= 1e18, "CF > 100%");
        poolCollateralFactors[pools[i]] = collateralFactorsMantissa[
    }
```

Oxdramaone (Duality Focus) confirmed, but disagreed with Medium severity and commented:

> We agree, we should have a max setting for collateral factor of pools to provide confidence to users. That said, this would only be abusable by admins, and so we consider low risk (since controlled by multisig).

Jack the Pug (judge) commented:

> Good catch!

> Unbounded `collateralFactor` configuration made it possible for malicious/compromised privileged roles to rug the users, which I consider a real and very practical threat that should be addressed from the smart contract level.

Oxdramaone (Duality Focus) resolved

[M-05] Arbitrary contract call within `UniV3LpVault._swap` with controllable `swapPath`

Submitted by rayn

UniV3LpVault.sol#L621
UniV3LpVault.sol#L379
UniV3LpVault.sol#L520
UniV3LpVault.sol#L521

`UniV3LpVault._swap` utilizes `swapRouter.exactInput` to perform swaps between two tokens. During swaps, `transfer` function of each token along the path will be called to propagate the assets.

Since anyone can create a uniswap pair of arbitrary assets, it is possible to include intermediate hop with malicious tokens within the path. Thus `UniV3LpVault._swap` effectively grants users the ability to perform arbitrary contract calls during the swap process if `swapPath` is not validated properly.

Usage of invalidated `swapPath` can be found in `UniV3LpVault.flashFocusCall` and `UniV3LpVault.repayDebt`.

Proof of Concept

The security of `Comptroller` and `UniV3LpVault` relies on validating all used tokens thoroughly. This is done by a whitelist mechanism where admin decides a predefined set of usable tokens, and users can only perform actions within the allowed range. This whitelist approach eliminates most of the attack surface regarding directly passing in malicious tokens as arguments.

Apart from passing malicious tokens directly, there are a few other potential weaknesses, the most obvious one is leveraging flash loans for collaterals. However, due to the adoption of AAVE LendingPool, the external validation within flash loan pool blocks this approach.

Unfortunately, a more obscure path exists. Looking at the swapping mechanism, it is not hard to realize it is backed by uniswapV3. An interesting characteristic of uniswap pools is that anyone can create pools for any token pairs, thus if we don't fully validate each and every pool we are using, chances are there will be malicious entries hidden within them.

This is partially the case which we see here, the user gets to supply a path, where the source and target are validated against benign tokens, the intermediate ones are not. An example of utilizing path for arbitrary function call is illustrated below

1. Create malicious token tokenM
2. Create pools tokenS<->tokenM and tokenM<->tokenT where tokenS and tokenT are benign tokens
3. Supply path (tokenS, tokenM, tokenT) for swapping

In the above case, when transferring tokenM while doing swap, we have full control over code executed and can insert arbitrary contract calls within.

Noticeably, while gaining arbitrary contract calls sounds dangerous, it does not necessarily mean the contract is exploitable. It still depends on the scenario in which an arbitrary call happens.

In the case of duality, the two locations where arbitrary `swapPath` can be provided is in `flashFocusCall` and `repayDebt`, both in which holds a local lock over `UniV3LpVault`. No global are applied to `Comptroller` or `Ctokens` while performing swaps.

```
    function flashFocusCall(FlashFocusParams calldata params) exterr
        ...
        {
            ...
            if (!tokenOfPool && params.swapPath.length > 0) amountIr
            ...
        }
        ...
    }

    function flashFocus(FlashFocusParams calldata params)
        external
```

```
        override
        nonReentrant(true)
        isAuthorizedForToken(params.tokenId)
        avoidsShortfall
    {
        ...
        flashLoan.LENDING_POOL().flashLoan(
            receiverAddress,
            assets,
            amounts,
            modes,
            onBehalfOf,
            newParams,
            referralCode
        );
    }

    function repayDebt(RepayDebtParams calldata params)
        external
        override
        nonReentrant(true)
        isAuthorizedForToken(params.tokenId)
        avoidsShortfall
        returns (uint256 amountReturned)
    {
        ...
        {
            ...
            if (amountOutFrom0 == 0 && params.swapPath0.length > 0)
            if (amountOutFrom1 == 0 && params.swapPath1.length > 0)
            ...
        }
        ...
    }
```

The lack of global locks here had us doubting whether an attack is possible. While we spent a considerable amount of time and failed to come up with any possible attack vectors, the complexity of the system held us back from concluding that an attack is impossible.

Thus we report this finding here in hope of inspiring developers either to prove the attack impossible or mitigate the attack surface.

Tools Used

vim, ganache-cli

Recommended Mitigation Steps

The easiest way to mitigate this is to validate the entire path against a predefined whitelist while in `_checkSwapPath`. This approach is far from optimal and also limits the flexibility of swapping between tokens. However, before security is proved, this is the best approach we can come up with.

Oxdramaone (Duality Focus) confirmed and commented:

> Great issue raised!

> Agreed that it is safest to validate all tokens to avoid arbitrary transfer functionality, and we will also enforce global reentrancy lock wherever possible. Agree with severity.

Jack the Pug (judge) commented:

> After some quick research on this issue, I also failed to find a valid attack vector. I agree that this is a noteworthy issue. A `med` severity seems fair given the detailed and honest issue description by the warden.

Oxdramaone (Duality Focus) resolved

Low Risk and Non-Critical Issues

For this contest, 17 reports were submitted by wardens detailing low risk and non-critical issues. The **report highlighted below** by **defsec** received the top score from the judge.

The following wardens also submitted reports: **rayn**, **lllllll**, **sorrynotsorry**, **cmichel**, **Dravee**, **robee**, **0xDjango**, **0x1f8b**, **0v3rf10w**, **cccz**, **kebabsec**, **bugwriter001**, **CertoraInc**, **hyh**, **samruna**, *and* **reassor**.

[L-01] Use safeTransfer/safeTransferFrom consistently instead of transfer/transferFrom

It is good to add a require() statement that checks the return value of token transfers or to use something like OpenZeppelin's safeTransfer/safeTransferFrom unless one is sure the given token reverts in case of a failure. Failure to do so will cause silent failures of transfers and affect token accounting in contract.

Reference: This similar medium-severity finding from Consensys Diligence Audit of Fei Protocol.

Proof of Concept

1. Navigate to the following contracts.
2. transfer/transferFrom functions are used instead of safe transfer/transferFrom on the following contracts.

FlashLoan.sol#L57
UniV3LpVault.sol#L366

Recommended Mitigation Steps

Consider using safeTransfer/safeTransferFrom or require() consistently.

[L-02] Missing Re-entrancy Guard

The re-entrancy guard is missing on the Eth anchor interaction. The external router interaction can cause to the re-entrancy vulnerability.

Proof of Concept

Navigate to the following contract.

CToken.sol#L781
CToken.sol#L905

Recommended Mitigation Steps

Follow the check effect interaction pattern or put re-entrancy guard.

[N-01] Incompatibility With Rebasing/Deflationary/Inflationary tokens

Does not appear to support rebasing/deflationary/inflationary tokens whose balance changes during transfers or over time. The necessary checks include at least verifying the amount of tokens transferred to contracts before and after the actual transfer to infer any fees/interest.

Proof of Concept

Navigate to the following contracts.

FlashLoan.sol#L57
UniV3LpVault.sol#L366

Recommended Mitigation Steps

- Ensure that to check previous balance/after balance equals to amount for any rebasing/inflation/deflation
- Add support in contracts for such tokens before accepting user-supplied tokens
- Consider supporting deflationary / rebasing / etc tokens by extra checking the balances before/after or strictly inform your users not to use such tokens if they don't want to lose them.

[N-02] Missing zero-address check in constructors and the setter functions

Missing checks for zero-addresses may lead to infunctional protocol, if the variable addresses are updated incorrectly.

Proof of Concept

Navigate to the following all contract functions.

UniV3LpVault.sol#L59
MasterPriceOracle.sol#L37
FlashLoan.sol#L24
UniV3LpVault.sol#L59

Recommended Mitigation Steps

Consider adding zero-address checks in the discussed constructors: require(newAddr != address(0));.

[N-03] Critical changes should use two-step procedure

The critical procedures should be two step process.

Proof of Concept

Navigate to the following contract.

UniV3LpVault.sol#L865
UniV3LpVault.sol#L877
UniV3LpVault.sol#L849

Recommended Mitigation Steps

Lack of two-step procedure for critical operations leaves them error-prone. Consider adding two step procedure on the critical functions.

[N-04] Missing events for only functions that change critical parameters

The afunctions that change critical parameters should emit events. Events allow capturing the changed parameters so that off-chain tools/interfaces can register such changes with timelocks that allow users to evaluate them and consider if they would like to engage/exit based on how they perceive the changes as affecting the trustworthiness of the protocol or profitability of the implemented financial services. The alternative of directly querying on-chain contract state for such changes is not considered practical for most users/usages.

Missing events and timelocks do not promote transparency and if such changes immediately affect users' perception of fairness or trustworthiness, they could exit the protocol causing a reduction in liquidity which could negatively impact protocol TVL and reputation.

Proof of Concept

Navigate to the following contract.

UniV3LpVault.sol#L765
UniV3LpVault.sol#L906

See similar High-severity H03 finding in OpenZeppelin's Audit of Audius and Medium-severity M01 finding in OpenZeppelin's Audit of UMA Phase 4.

Recommended Mitigation Steps

Add events to all functions that change critical parameters.

[N-05] Use of Block.timestamp

Block timestamps have historically been used for a variety of applications, such as entropy for random numbers (see the Entropy Illusion for further details), locking funds for periods of time, and various state-changing conditional statements that are time-dependent. Miners have the ability to adjust timestamps slightly, which can prove to be dangerous if block timestamps are used incorrectly in smart contracts.

Proof of Concept

Navigate to the following contract.

UniV3LpVault.sol#L676

Recommended Mitigation Steps

Block timestamps should not be used for entropy or generating random numbers—i.e., they should not be the deciding factor (either directly or through some derivation) for winning a game or changing an important state.

Time-sensitive logic is sometimes required; e.g., for unlocking contracts (time-locking), completing an ICO after a few weeks, or enforcing expiry dates. It is sometimes recommended to use block.number and an average block time to estimate times; with a 10 second block time, 1 week equates to approximately, 60480 blocks. Thus, specifying a block number at which to change a contract state can be more secure, as miners are unable to easily manipulate the block number.

0xdramaone (Duality Focus) commented:

> Great report.

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