OpenLeverage contest Findings & Analysis Report

2022-03-09

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 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 the OpenLeverage smart contract system written in Solidity. The code contest took place between January 27—February 2 2022.

Wardens

31 Wardens contributed reports to the OpenLeverage contest:

1. hyh
2. defsec
3. gzeon
4. WatchPug (jtp and ming)
5. robee
6. mics
7. csanuragjain
8. Dravee
9. pauliax
10. jayjonah8

11. cmichel
12. jonah1005
13. rfa
14. samruna
15. sirhashalot
16. cccz
17. 0x1f8b
18. 0x0x0x
19. tqts
20. Tomio
21. throttle
22. 0v3rf10w
23. Ruhum
24. p4st13r4 (0x69e8 and 0xb4bb4)
25. Fitraldys
26. m_smirnova2020
27. lllllll
28. GeekyLumberjack
29. wuwe1

This contest was judged by 0xleastwood.

Final report assembled by liveactionllama and CloudEllie.

Summary

The C4 analysis yielded an aggregated total of 21 unique vulnerabilities and 83 total findings. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 1 received a risk rating in the category of HIGH severity, 5 received a risk rating in the category of MEDIUM severity, and 15 received a risk rating in the category of LOW severity.

C4 analysis also identified 22 non-critical recommendations and 40 gas optimizations.

Scope

The code under review can be found within the C4 OpenLeverage contest repository, and is composed of 33 smart contracts written in the Solidity programming language and includes 4251 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 (1)

[H-01] OpenLevV1Lib's and LPool's `doTransferOut` functions call native `payable.transfer`, which can be unusable for smart contract calls

Submitted by hyh

When OpenLev operations use a wrapped native token, the whole user withdraw is being handled with a `payable.transfer()` call.

This is unsafe as `transfer` has hard coded gas budget and can fail when the user is a smart contract. This way any programmatical usage of OpenLevV1 and LPool is at risk.

Whenever the user either fails to implement the payable fallback function or cumulative gas cost of the function sequence invoked on a native token transfer exceeds 2300 gas consumption limit the native tokens sent end up undelivered and the corresponding user funds return functionality will fail each time.

As OpenLevV1 `closeTrade` is affected this includes user's principal funds freeze scenario, so marking the issue as a high severity one.

Proof of Concept

OpenLevV1Lib and LPool have `doTransferOut` function that calls native token payable.transfer:

OpenLevV1Lib.doTransferOut

https://github.com/code-423n4/2022-01-openleverage/blob/main/openleverage-contracts/contracts/OpenLevV1Lib.sol#L253

LPool.doTransferOut

https://github.com/code-423n4/2022-01-openleverage/blob/main/openleverage-contracts/contracts/liquidity/LPool.sol#L297

LPool.doTransferOut is used in LPool redeem and borrow, while OpenLevV1Lib.doTransferOut is used in OpenLevV1 trade manipulation logic:

closeTrade

https://github.com/code-423n4/2022-01-openleverage/blob/main/openleverage-contracts/contracts/OpenLevV1.sol#L204

https://github.com/code-423n4/2022-01-openleverage/blob/main/openleverage-contracts/contracts/OpenLevV1.sol#L215

liquidate

https://github.com/code-423n4/2022-01-openleverage/blob/main/openleverage-contracts/contracts/OpenLevV1.sol#L263

https://github.com/code-423n4/2022-01-openleverage/blob/main/openleverage-contracts/contracts/OpenLevV1.sol#L295

https://github.com/code-423n4/2022-01-openleverage/blob/main/openleverage-contracts/contracts/OpenLevV1.sol#L304

References

The issues with `transfer()` are outlined here:

https://consensys.net/diligence/blog/2019/09/stop-using-soliditys-transfer-now/

Recommended Mitigation Steps

OpenLevV1's `closeTrade` and `liquidate` as well as LPool's `redeem`, `redeemUnderlying`, `borrowBehalf`, `repayBorrowBehalf`, `repayBorrowEndByOpenLev` are all `nonReentrant`, so reentrancy isn't an issue and `transfer()` can be just replaced.

Using low-level `call.value(amount)` with the corresponding result check or using the OpenZeppelin `Address.sendValue` is advised:

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/Address.sol#L60

ColaM12 (OpenLeverage) confirmed and resolved

0xleastwood (judge) commented:

> Awesome find! Completely agree with the warden here. This would prevent users from calling sensitive functions which withdraw their funds in some way.

Medium Risk Findings (5)

[M-01] `UniV2ClassDex.sol#uniClassSell()` Tokens with fee on transfer are not fully supported

Submitted by WatchPug

https://github.com/code-423n4/2022-01-openleverage/blob/501e8f5c7ebaf1242572712626a77a3d65bdd3ad/openleverage-contracts/contracts/dex/bsc/UniV2ClassDex.sol#L31-L56

```
    function uniClassSell(DexInfo memory dexInfo,
        address buyToken,
        address sellToken,
        uint sellAmount,
        uint minBuyAmount,
        address payer,
        address payee
    ) internal returns (uint buyAmount){
        address pair = getUniClassPair(buyToken, sellToken, dexInfo.
        IUniswapV2Pair(pair).sync();
        (uint256 token0Reserves, uint256 token1Reserves,) = IUniswap
        sellAmount = transferOut(IERC20(sellToken), payer, pair, sel
        uint balanceBefore = IERC20(buyToken).balanceOf(payee);
        dexInfo.fees = getPairFees(dexInfo, pair);
        if (buyToken < sellToken) {
            buyAmount = getAmountOut(sellAmount, token1Reserves, tok
            IUniswapV2Pair(pair).swap(buyAmount, 0, payee, "");
        } else {
            buyAmount = getAmountOut(sellAmount, token0Reserves, tok
            IUniswapV2Pair(pair).swap(0, buyAmount, payee, "");
        }

        require(buyAmount >= minBuyAmount, 'buy amount less than mir
        uint bought = IERC20(buyToken).balanceOf(payee).sub(balanceB
        return bought;
    }
```

While `uniClassBuy()` correctly checks the actually received amount by comparing the before and after the balance of the receiver, `uniClassSell()` trusted the result given by `getAmountOut()`. This makes `uniClassSell()` can result in an output amount fewer than `minBuyAmount`.

https://github.com/code-423n4/2022-01-openleverage/blob/501e8f5c7ebaf1242572712626a77a3d65bdd3ad/openleverage-contracts/contracts/dex/bsc/UniV2ClassDex.sol#L101-L102

Recommendation

Change to:

```
    function uniClassSell(DexInfo memory dexInfo,
        address buyToken,
        address sellToken,
        uint sellAmount,
        uint minBuyAmount,
        address payer,
        address payee
    ) internal returns (uint bought){
        address pair = getUniClassPair(buyToken, sellToken, dexInfo.
        IUniswapV2Pair(pair).sync();
        (uint256 token0Reserves, uint256 token1Reserves,) = IUniswap
        sellAmount = transferOut(IERC20(sellToken), payer, pair, sel
        uint balanceBefore = IERC20(buyToken).balanceOf(payee);
        dexInfo.fees = getPairFees(dexInfo, pair);
        if (buyToken < sellToken) {
            buyAmount = getAmountOut(sellAmount, token1Reserves, tok
            IUniswapV2Pair(pair).swap(buyAmount, 0, payee, "");
        } else {
            buyAmount = getAmountOut(sellAmount, token0Reserves, tok
            IUniswapV2Pair(pair).swap(0, buyAmount, payee, "");
        }
        uint bought = IERC20(buyToken).balanceOf(payee).sub(balanceB
        require(bought >= minBuyAmount, 'buy amount less than min');
    }
```

ColaM12 (OpenLeverage) confirmed and resolved

Oxleastwood (judge) commented:

> Agree with this finding! Uniswap token swaps are meant to support all types of tokens. It does seem possible for there to be `payer` to experience increased slippage because the check operates on `getAmountOut()` and not the `bought` output.

> It's fair to say that this will lead to value leakage, so I think `medium` severity is justified.

[M-02] Missing payable

Submitted by robee

The following functions are not payable but uses msg.value - therefore the function must be payable. This can lead to undesired behavior.

```
LPool.sol, addReserves should be payable since using msg.val
```

ColaM12 (OpenLeverage) confirmed and resolved

0xleastwood (judge) commented:

> Nice find! The warden has identified a function which is missing the `payable` keyword. Preventing any users from adding reserves using native ether.

[M-03] Eth sent to Timelock will be locked in current implementation

Submitted by defsec

Eth sent to Timelock will be locked in current implementation. I came across this problem while playing around with the governance contract.

Proof of Concept

- Setup the governance contracts (GovernanceAlpha, Timelock)

- Send eth to timelock contract
- Setup a proposal to send 0.1 eth out. Code snippet in ether.js below. proxy refers to GovernorAlpha.

```
await proxy.propose(
    [signers[3].address],
    [ethers.utils.parseEther("0.1")],
    [""],
    [ethers.BigNumber.from(0)],
    "Send funds to 3rd signer"
);
```

- Vote and have the proposal succeed.
- Execute the proposal, the proposal number here is arbitrary.

```
await proxy.execute(2);  // this fails
    await proxy.execute(2, {value: ethers.utils.parseEther("0.1'
    0.1 eth will be sent out, but it is sent from the msg.sender
```

Recommended Mitigation Steps

Consider implementing the following code.

```
function execute(uint proposalId) external {
    require(state(proposalId) == ProposalState.Queued, "Governor
    Proposal storage proposal = proposals[proposalId];
    proposal.executed = true;
    for (uint i = 0; i < proposal.targets.length; i++) {
        timelock.executeTransaction(proposal.targets[i], proposa
    }
    emit ProposalExecuted(proposalId);
}
```

Reference

https://github.com/compound-finance/compound-protocol/pull/177/files

ColaM12 (OpenLeverage) acknowledged

0xleastwood (judge) commented:

> I agree with this finding!

[M-04] OpenLevV1.closeTrade with V3 DEX doesn't correctly accounts fee on transfer tokens for repayments

Submitted by hyh

The amount that OpenLevV1 will receive can be less than V3 DEX indicated as a swap result, while it is used as given for position debt repayment accounting.

This way actual funds received can be less than accounted, leaving to system funds deficit, which can be exploited by a malicious user, draining contract funds with multiple open/close with a taxed token.

In the `trade.depositToken != longToken` case when `flashSell` is used this can imply inability to send remainder funds to a user and the failure of the whole closeTrade function, the end result is a freezing of user's funds within the system.

Proof of Concept

`trade.depositToken != longToken` case, can be wrong repayment accounting, which will lead to a deficit if the received funds are less than DEX returned `closeTradeVars.receiveAmount`.

As a side effect, `doTransferOut` is done without balance check, so the whole position close can revert, leading to inability to close the position and freeze of user's funds this way:

https://github.com/code-423n4/2022-01-openleverage/blob/main/openleverage-contracts/contracts/OpenLevV1.sol#L197-204

I.e. if there is enough funds in the system they will be drained, if there is not enough funds, user's position close will fail.

V3 sell function doesn't check for balance change, using DEX returned amount as is:

https://github.com/code-423n4/2022-01-openleverage/blob/main/openleverage-contracts/contracts/dex/eth/UniV3Dex.sol#L61-70

Recommended Mitigation Steps

If fee on tranfer tokens are fully in scope, do control all the accounting and amounts to be returned to a user via balance before/after calculations for DEX V3 logic as well.

ColaM12 (OpenLeverage) confirmed and resolved

0xleastwood (judge) commented:

> Awesome find. I was able to confirm that `UniV3Dex.uniV3Sell()` does not properly handle fee-on-transfer tokens by treating the amount received as the difference between before balance and after balance.

[M-05] anti-flashloan mechanism may lead to protocol default

Submitted by gzeon

There is a price check to avoid flash loan attacks which significantly moved the price. If current price is 5% lower than the stored twap price, the liquidation will fail. This design can be dangerous as it is to openleverage's benefit to close under-collateralized position ASAP when there is a huge market drawdown. When the market keep trading downward, it is possible that the spot price keep trading 5% lower than the twap, which prevent any liquidation from happening and causing the protocol to be under-collateralized.

Proof of Concept

https://github.com/code-423n4/2022-01-openleverage/blob/501e8f5c7ebaf1242572712626a77a3d65bdd3ad/openleverage-contracts/contracts/OpenLevV1Lib.sol#L191

```
    // Avoid flash loan
    if (prices.price < prices.cAvgPrice) {
```

```
        uint differencePriceRatio = prices.cAvgPrice.mul(100).div(pr
        require(differencePriceRatio - 100 < maxLiquidationPriceDiff
    }
```

Recommended Mitigation Steps

Instead of revert with `maxLiquidationPriceDiffientRatio`, use the twap price to determine if the position is healthy.

ColaM12 (OpenLeverage) disputed

0xleastwood (judge) commented:

> From first impression, this finding seems legitimate. Can I get some more details on why it was disputed? @ColaM12

ColaM12 (OpenLeverage) commented:

> There is always a chance to front run a flash loan transaction before trading in OpenLev. Also, see in line 196, position is considered not healthy only if all three price check failed including the twap price.

0xleastwood (judge) commented:

> It looks like only one condition would need to be satisfied for `isPositionHealthy` to return false as it uses `||` and not `&&`.

ColaM12 (OpenLeverage) commented:

> Do you mean return true? All 3 price checks should fail when liquidating. But the position may still hold funds to pay off debt. by using maxLiquidationPriceDiffientRatio, under-priced-swaps can be limited . Otherwise, all remaining funds in the position could be drained from a flash loan attack which directly leads to a bad debt to lender.

0xleastwood (judge) commented:

> Ahh sorry my mistake. I misinterpreted that.

> I agree with the sponsor here. The issue outlined by the warden seems to be safeguarded by the two other checks in `isPositionHealthy()`

Oxleastwood (judge) commented:

> Actually thinking about this more, I think the warden raised an issue related to the liquidations continuing to fail if the price keeps trending downward at an accelerated pace. I don't think the protocol would be able to respond to such events if **this** reverts.

Oxleastwood (judge) commented:

> After discussion with the sponsor, we have agreed that this issue is valid. It is expected that the TWAP is only valid for 1 min. By removing this condition, there is potential for even larger security issues. So the sponsor has decided to make this a wont-fix but I'll keep the issue open as it is valid.

> This was an awesome find!

Low Risk Findings (15)

- **[L-01] Funds can be lost** *Submitted by csanuragjain*
- **[L-02] endTime can be before startTime** *Submitted by samruna, also found by defsec and WatchPug*
- **[L-03] transfer() may break in future ETH upgrade** *Submitted by gzeon, also found by pauliax*
- **[L-04] The check for `max rate 1000 ole` should be inclusive** *Submitted by Dravee*
- **[L-05] User reward can get stuck** *Submitted by csanuragjain*
- **[L-06] Anyone can call release() in OLETokenLock.sol** *Submitted by jayjonah8*
- **[L-07] Race condition in approve()** *Submitted by cccz, also found by defsec, mics, and sirhashalot*
- **[L-08] Anyone can claim airdrop amounts on behalf of anyone** *Submitted by cmichel*
- **[L-09] Assert instead require to validate user inputs** *Submitted by mics, also found by defsec, Dravee, and hyh*

- [L-10] Anyone can crash transferTo *Submitted by pauliax*
- [L-11] UniV2Dex and UniV2ClassDex use hard coded factory addresses for Pair and PairFees getters *Submitted by hyh*
- [L-12] Mult instead div in compares *Submitted by mics*
- [L-13] In the following public update functions no value is returned *Submitted by mics*
- [L-14] UniV3Dex uniV3Buy slippage check error message is misleading *Submitted by hyh*
- [L-15] Bad actor may steal deposit return when liquidating a trade *Submitted by jonah1005*

Non-Critical Findings (22)

- [N-01] Require with empty message *Submitted by mics*
- [N-02] Require with not comprehensive message *Submitted by mics*
- [N-03] transferAllowed does not fail *Submitted by GeekyLumberjack*
- [N-04] The initialize function can be called multiple times *Submitted by cccz, also found by 0x1f8b, Dravee, Fitraldys, hyh, p4st13r4, rfa, sirhashalot, and wuwe1*
- [N-05] no validation checks in ControllerV1.sol initialize function() *Submitted by jayjonah8, also found by 0v3rf10w, cccz, Dravee, and hyh*
- [N-06] `ControllerStorage` : related market data should be grouped in a struct *Submitted by Dravee*
- [N-07] No validation for constructor arguments in OLEToken.sol *Submitted by jayjonah8*
- [N-08] Multiple potential reentrancies *Submitted by 0v3rf10w*
- [N-09] Use of tx.origin in ControllerV1.sol *Submitted by jayjonah8*
- [N-10] Two arrays length mismatch *Submitted by mics*
- [N-11] No Transfer Ownership Pattern *Submitted by cccz, also found by Dravee*
- [N-12] mint() function doesn't require 0 to be larger than 0 *Submitted by jayjonah8*
- [N-13] FarmingPools' notifyRewardAmounts and initDistributions do not check the lengths of input arrays *Submitted by hyh*

- [N-14] Unused parameters in OpenLevV1 and ControllerV1 functions *Submitted by hyh, also found by samruna*
- [N-15] Named return issue *Submitted by mics*
- [N-16] Misc *Submitted by 0x1f8b*
- [N-17] Last reward is discarded when reward added twice *Submitted by csanuragjain*
- [N-18] Not verified input *Submitted by mics*
- [N-19] Does not validate the input fee parameter *Submitted by mics*
- [N-20] Never used parameters *Submitted by mics*
- [N-21] Unused imports *Submitted by mics*
- [N-22] Timelock.sol modification removes logic checks *Submitted by sirhashalot*

Gas Optimizations (40)

- [G-01] Gas: Tautology on “variable >= 0” which is always true as variable is uint *Submitted by Dravee, also found by 0v3rf10w, gzeon, and pauliax*
- [G-02] Optimize `OpenLevV1.sol#addMarket` *Submitted by 0x0x0x*
- [G-03] Prefix increments are cheaper than postfix increments *Submitted by mics, also found by 0x1f8b, defsec, Dravee, lllllll, m*smirnova2020, p4st13r4, and throttle_
- [G-04] State variables that could be set immutable *Submitted by mics, also found by 0x1f8b, gzeon, pauliax, Ruhum, and throttle*
- [G-05] Gas saving optimizing setImplementation *Submitted by 0x1f8b*
- [G-06] Gas saving optimizing storage *Submitted by 0x1f8b, also found by Dravee*
- [G-07] Gas Optimization: Tight variable packing in `LPoolStorage.sol` *Submitted by Dravee*
- [G-08] Gas: “constants” expressions are expressions, not constants. Use “immutable” instead. *Submitted by Dravee, also found by pauliax*
- [G-09] Use bytes32 instead of string to save gas whenever possible *Submitted by mics, also found by Dravee*

- [G-10] Caching array length can save gas *Submitted by mics, also found by defsec, Dravee, pauliax, throttle, and WatchPug*
- [G-11] Short the following require messages *Submitted by robee, also found by Dravee, mics, and sirhashalot*
- [G-12] Upgrade pragma to at least 0.8.4 *Submitted by mics, also found by defsec, Dravee, and gzeon*
- [G-13] Gas: Shift Right instead of Dividing by 2 *Submitted by Dravee*
- [G-14] Use != 0 instead of > 0 *Submitted by mics, also found by Dravee and gzeon*
- [G-15] Use calldata instead of memory *Submitted by mics, also found by defsec, Dravee, rfa, Ruhum, Tomio, and WatchPug*
- [G-16] Gas in `Adminable.sol:acceptAdmin()` : SLOADs minimization *Submitted by Dravee, also found by Fitraldys, mics, p4st13r4, pauliax, Tomio, and WatchPug*
- [G-17] Gas: `// Shh - currently unused` *Submitted by Dravee, also found by sirhashalot*
- [G-18] Gas in `LPool.sol:availableForBorrow()` : Avoid expensive calculation with an inclusive inequality *Submitted by Dravee*
- [G-19] Inline one time use functions *Submitted by mics, also found by Dravee and robee*
- [G-20] Unnecessary equals boolean *Submitted by mics, also found by Dravee*
- [G-21] Using `require` instead of `&&` can save gas *Submitted by Tomio, also found by rfa*
- [G-22] Unused library `ReentrancyGuard` *Submitted by WatchPug, also found by defsec*
- [G-23] Gas savings and corrections *Submitted by csanuragjain*
- [G-24] Gas Optimization: No need to use SafeMath everywhere *Submitted by gzeon*
- [G-25] Gas Optimization: Redundant check *Submitted by gzeon*
- [G-26] OpenLevV1.closeTrade can save trade.deposited to memory *Submitted by hyh*

- [G-27] uniV2Buy calls buyAmount.toAmountBeforeTax twice, while it's constant *Submitted by hyh*
- [G-28] Unnecessary array boundaries check when loading an array element twice *Submitted by mics*
- [G-29] Check if amount is not zero to save gas *Submitted by mics*
- [G-30] Unused inheritance *Submitted by mics*
- [G-31] using > instead of >= *Submitted by rfa*
- [G-32] unnecessary uint declaration *Submitted by rfa*
- [G-33] use require instead if/else *Submitted by rfa*
- [G-34] set pancakeFactory to constant *Submitted by rfa*
- [G-35] unnecessary _unsedFactory call *Submitted by rfa*
- [[G-36] pass the `dexInfo[dexName[i]` value without caching `DexInfo struct`](https://github.com/code-423n4/2022-01-openleverage-findings/issues/71) *Submitted by rfa*
- [G-37] caching struct data type in memory cost more gas *Submitted by rfa*
- [G-38] declaring that contract is using `Utils` lib can use more gas *Submitted by rfa*
- [G-39] unnecessary msg.sender cache *Submitted by rfa*
- [G-40] Gas saving by caching state variables *Submitted by tqts*

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.