Perennial contest
Findings & Analysis Report

2022-01-24

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 Perennial contest smart contract system written in Solidity. The code contest took place between December 9—December 15 2021.

Wardens

13 Wardens contributed reports to the Perennial contest:

1. leastwood
2. kenzo
3. WatchPug (jtp and ming)
4. 0x0x0x
5. 0x1f8b
6. cmichel
7. robee
8. hubble (ksk2345 and shri4net)
9. defsec
10. ye0lde
11. gzeon

12. pmerkleplant
13. broccolirob

This contest was judged by Alex the Entreprenerd.

Final report assembled by itsmetechjay and CloudEllie.

Summary

The C4 analysis yielded an aggregated total of 11 unique vulnerabilities and 38 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, 3 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 5 non-critical recommendations and 22 gas optimizations.

Scope

The code under review can be found within the C4 Perennial contest repository, and is composed of 38 smart contracts written in the Solidity programming language and includes 3531 lines of Solidity code and 7 lines of JavaScript.

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] Wrong shortfall calculation

Submitted by kenzo

Every time an account is settled, if shortfall is created, due to a wrong calculation shortfall will double in size and add the new shortfall.

Impact

Loss of funds: users won't be able to withdraw the correct amount of funds. Somebody would have to donate funds to resolve the wrong shortfall.

Proof of Concept

We can see in the `settleAccount` of `OptimisticLedger` that `self.shortfall` ends up being `self.shortfall+self.shortfall+newShortfall` : [(Code ref)](#)

```
    function settleAccount(OptimisticLedger storage self, address ac
    internal returns (UFixed18 shortfall) {
        Fixed18 newBalance = Fixed18Lib.from(self.balances[account])

        if (newBalance.sign() == -1) {
            shortfall = self.shortfall.add(newBalance.abs());
            newBalance = Fixed18Lib.ZERO;
        }

        self.balances[account] = newBalance.abs();
        self.shortfall = self.shortfall.add(shortfall);
    }
```

Additionally, you can add the following line to the “shortfall reverts if depleted” test in `Collateral.test.js`, line 190:

```
    await collateral.connect(productSigner).settleAccount(userB.addr
```

Previously the test product had 50 shortfall. Now we added 50 more, but the test will print that the actual shortfall is 150, and not 100 as it should be.

Recommended Mitigation Steps

Move the setting of `self.shortfall` to inside the if function and change the line to:

```
        self.shortfall = shortfall
```

kbrizzle (Perennial) confirmed:

> Excellent find 🙏

Alex the Entreprenerd (judge) commented:

> Agree with the finding `shortfall = self.shortfall.add(newBalance.abs());` is already shortfal + newBalance.abs() So performing line `73 self.shortfall = self.shortfall.add(shortfall);` is adding `shortfall` again

[H-02] `withdrawTo` Does Not Sync Before Checking A Position's Margin Requirements

Submitted by leastwood

Impact

The `maintenanceInvariant` modifier in `Collateral` aims to check if a user meets the margin requirements to withdraw collateral by checking its current and next maintenance. `maintenanceInvariant` inevitably calls

`AccountPosition.maintenance` which uses the oracle's price to calculate the margin requirements for a given position. Hence, if the oracle has not synced in a long time, `maintenanceInvariant` may end up utilising an outdated price for a withdrawal. This may allow a user to withdraw collateral on an undercollaterized position.

Proof of Concept

https://github.com/code-423n4/2021-12-perennial/blob/main/protocol/contracts/collateral/Collateral.sol#L67-L76

```
    function withdrawTo(address account, IProduct product, UFixed18
    notPaused
    collateralInvariant(msg.sender, product)
    maintenanceInvariant(msg.sender, product)
    external {
        _products[product].debitAccount(msg.sender, amount);
        token.push(account, amount);

        emit Withdrawal(msg.sender, product, amount);
    }
```

https://github.com/code-423n4/2021-12-perennial/blob/main/protocol/contracts/collateral/Collateral.sol#L233-L241

```
    modifier maintenanceInvariant(address account, IProduct product)
        _;

        UFixed18 maintenance = product.maintenance(account);
        UFixed18 maintenanceNext = product.maintenanceNext(account);

        if (UFixed18Lib.max(maintenance, maintenanceNext).gt(collate
            revert CollateralInsufficientCollateralError();
    }
```

https://github.com/code-423n4/2021-12-perennial/blob/main/protocol/contracts/product/types/position/AccountPosition.sol#L71-L75

```
    function maintenanceInternal(Position memory position, IProductI
        Fixed18 oraclePrice = provider.priceAtVersion(provider.curre
        UFixed18 notionalMax = Fixed18Lib.from(position.max()).mul(c
        return notionalMax.mul(provider.maintenance());
    }
```

Tools Used

Manual code review.

Recommended Mitigation Steps

Consider adding `settleForAccount(msg.sender)` to the `Collateral.withdrawTo` function to ensure the most up to date oracle price is used when assessing an account's margin requirements.

kbrizzle (Perennial) confirmed:

> Great catch 🙏

Alex the Entreprenerd (judge) commented:

> With most onChain protocols where there is potential for undercollateralized positions and liquidations, it is very important to accrue a user position before making any changes to their balance.
>
> The warden identified a potential way for a user to withdraw funds while their account is below margin requirements.
>
> Because this impacts the core functionality functionality of the protocol (accounting), I'm raising the severity to high
>
> Mitigation seems to be straightforward

Medium Risk Findings (3)

[M-01] No checks if given product is created by the factory

Submitted by 0x0x0x

An attacker can create a fake product. `Collateral` contract does not check whether the given product is created by the factory. A malicious product can return arbitrary maintenance amounts, therefore they can make any deposit to fake product stuck (simply return collateral - 1 as maintenance) and fake product owner can change the maintenance and liquidate whenever wanted and claim full collateral.

This is a serious attack vector, since by only interacting with `Collateral` contract the user can lose funds. Furthermore, this vulnerability poses big risks in combination web vulnerabilities. Users always have to check the product address to avoid getting scammed, but likely most users will only check the contract address.

Furthermore, if another protocol would want to add perennial protocol for its use case, the other protocol has to be careful with this behaviour and keep track of whitelisted products. This complicates adoption of perennial protocol, since whitelist has to be managed manually or else this vulnerability will likely be exploitable.

Mitigation step

Add a mapping inside `Collateral`, which verifies whether a product is created by factory or not. This mapping should get updated by the factory. This will add a little bit gas cost, but will eliminate a serious attack vector.

Or less gas efficient option is directly call a function from factory to verify.

kbrizzle (Perennial) disagreed with severity:

> Good find 🙏

> On severity: In general there's a whole plethora of ways a Product owner could create a product that harmful a user which choses to take part in it. The design model of Perennial is to give the Product owners the *freedom* to create risky products, but to segregate that risk to only those products, and this issue does not break this segregation of risk. This is especially important during our gated-beta phase while we learn which levers are worth it to lock down.

> That being said this is low hanging fruit to correct, so we think it should still be `2 (Medium)` severity.

Alex the Entreprenerd (judge) commented:

> Given the extensible nature of the `Collateral` contract, users can call the contract with any user input for `product` this could be used maliciously against the users as the warden highlighted.

> This can definitely happen, but is contingent on a set of external factors, so I think medium severity is more appropriate

[M-02] Multiple initialization of Collateral contract

Submitted by 0x1f8b

Impact

The attacker can initialize the contract, take malicious actions, and allow it to be re-initialized by the project without any error being noticed.

Proof of Concept

The `initialize` method of the `Collateral` contract does not contain the `initializer` modifier, it delegates this process to check if the factory is different from address(0) in the `UFactoryProvider__initialize` call, however the `factory_` is other than address(0), so an attacker could use front-running by listening to the memory pool, initialize the contract with a `factory=address(0)`, perform malicious actions, and still allow it to be started later by the I draft the contract without noticing any error.

Source reference:

- Collateral.initialize

Tools Used

Manual review

Recommended Mitigation Steps

Use `initializer` modifier

kbrizzle (Perennial) marked as duplicate:

> Duplicate of: https://github.com/code-423n4/2021-12-perennial-findings/issues/71

Alex the Entreprenerd (judge) commented:

> Disagree with sponsor that this is duplicate of #73

Alex the Entreprenerd (judge) commented:

> The finding mentions that `initialize` can be called multiple times, specifically by first having `factory` set to address(0) and then setting it to it's final value.
>
> This is true.
>
> Calling initialize multiple times could allow to set certain values up with a different token, then change to another token and set the factory. This could be used to trick the protocol accounting to later take advantage of other depositors.

Alex the Entreprenerd (judge) commented:

> The specific vulnerability is the lack of the `initializer` modifier on initialize, the consequences can be problematic only if a malicious actor were to set the contracts up.
>
> Because this can be done, and it's reliant on a specific setup, I'll mark the finding as medium severity, mitigation is straightforward

kbrizzle (Perennial) commented:

> My understanding of these is that:
>
> - 73 would allow an attacker to front-run the initialization of a contract to gain admin, but the

subsequent legitimate attempt to initialize would revert.

- 13 would allow an attacker to front-run the initialization of a contract to gain admin, and the subsequent legitimate attempt to initialize would still work (thereby possibly going unnoticed).

> If we want to tag these as separate issues, I think that makes sense as the effects are slightly different. Both are ultimately a documentation issue though solved by this remediation, so I'm wondering why one is tagged as `0 (Non-critical)` and the other `2 (Medium)`.

> Is that correct, or is there an issue here even *with* proper atomic deploy-and-initialize usage with OZ's upgrades plugin? Just want to make sure we're not missing something else here. Also - we do not have an `initializer` modifier available in the code base, so the remediation suggestion was a little confusing.

Alex the Entreprenerd (judge) commented:

> @kbrizzle Yes, this is something that can still be done, arguably only by your deployer

> You could deploy and `initialize` the contract with factory set to 0, use that to perform operations with a token you created (so it has no cost to you), then `initialize` again with factory set to the proper value

> Because certain functions do not check for the product being created by the factory, this can be done as a way to lay the ground for a rug against future depositors

> So to my eyes this finding highlights the consequences of a lack of the `initializer` modifier, while the other is the usual `initializer can be frontrun` non critical finding

Alex the Entreprenerd (judge) commented:

> @kbrizzle to clarify: you won't be frontrun with your deployment as the deploy + initialize is done in one tx. But the `Collateral` not having `initializer` can be used by the deployer for malicious purposes because what was perceived as an `unchangeable` token can actually be changed if you set up `initialize` with the factory address set to 0

kbrizzle (Perennial) commented:

> @Alex the Entreprenerd I see, that makes sense from the perspective of the token seeming like it should be immutable 👍

> FWIW in this scenario the deployer would have to run the risk of someone else initializing and taking admin for the duration that the malicious initial token was active - so I'm not sure how feasible that would be.

> That being said, I think a `UInitializable` would be a great addition to our lib anyways, so let's go this route since this does seem like a solution with better guarantees. Thanks for the explanation!

Alex the Entreprenerd (judge) commented:

> The attack is dependent upon external conditions (malicious deployer, nobody else calling initialize, setting up malicious accounting), for that reason (as per the docs) I believe medium severity to be proper.

[M-03] Chainlink's `latestRoundData` might return stale or incorrect results

Submitted by WatchPug, also found by cmichel, defsec, and yeOlde

https://github.com/code-423n4/2021-12-perennial/blob/fd7c38823833a51ae0c6ae3856a3d93a7309c0e4/protocol/contracts/oracle/ChainlinkOracle.sol#L50-L60

```
    function sync() public {
        (, int256 feedPrice, , uint256 timestamp, ) = feed.latestRou
        Fixed18 price = Fixed18Lib.ratio(feedPrice, SafeCast.toInt25
```

```
        if (priceAtVersion.length == 0 || timestamp > timestampAtVer
            priceAtVersion.push(price);
            timestampAtVersion.push(timestamp);

            emit Version(currentVersion(), timestamp, price);
        }
    }
```

On `ChainlinkOracle.sol`, we are using `latestRoundData`, but there is no check if the return value indicates stale data. This could lead to stale prices according to the Chainlink documentation:

- https://docs.chain.link/docs/historical-price-data/#historical-rounds
- https://docs.chain.link/docs/faq/#how-can-i-check-if-the-answer-to-a-round-is-being-carried-over-from-a-previous-round

Recommendation

Consider adding missing checks for stale data.

For example:

```
(uint80 roundID, int256 feedPrice, , uint256 timestamp, uint80 a
require(feedPrice > 0, "Chainlink price <= 0");
require(answeredInRound >= roundID, "Stale price");
require(timestamp != 0, "Round not complete");
```

kbrizzle (Perennial) confirmed

Alex the Entreprenerd (judge) commented:

> Agree with the finding, while you can get started with Chainlink's feed can be used with one line of code, in a production environment it is best to ensure that the data is fresh and within rational bounds

Low Risk Findings (6)

- [L-02] Not verified function inputs of public / external functions *Submitted by robee*
- [L-03] On updating the Incentive fee greater than UFixedLib18.ONE, new Programs can not be created *Submitted by hubble*
- [L-04] Missing fee parameter validation *Submitted by cmichel, also found by 0x1f8b and defsec*
- [L-05] Fixed18 conversions don't work for all values *Submitted by cmichel, also found by WatchPug*
- [L-06] `Factory.sol#updateController()` Lack of input validation *Submitted by WatchPug*
- [L-01] `Incentivizer.sol` Tokens with fee on transfer are not supported *Submitted by WatchPug*

Non-Critical Findings (5)

- [N-05] Unsecure Ownership Transfer *Submitted by 0x1f8b, also found by robee*
- [N-01] Unused imports *Submitted by robee*
- [N-02] Solidity compiler versions mismatch *Submitted by robee*
- [N-03] Initialization functions can be front-run *Submitted by broccolirob, also found by leastwood, cmichel, and WatchPug*
- [N-04] Removing redundant code can save gas (Collateral, Factory, Incentivizer, ChainlinkOracle) *Submitted by yeOlde*

Gas Optimizations (22)

- [G-01] claimFee loop does not check for zero transfer amount (Incentivizer.sol) *Submitted by yeOlde*
- [G-02] Cache storage read and call results in the stack can save gas *Submitted by WatchPug, also found by gzeon*
- [G-03] Cache array length in for loops can save gas *Submitted by WatchPug, also found by 0x0x0x, pmerkleplant, and robee*
- [G-04] Unnecessary checked arithmetic in for loops *Submitted by WatchPug, also found by robee*

- [G-05] Reuse operation results can save gas *Submitted by WatchPug, also found by robee*
- [G-06] Use immutable variables can save gas *Submitted by WatchPug, also found by robee*
- [G-07] Best Practice: public functions not used by current contract should be external *Submitted by WatchPug, also found by robee*
- [G-09] Avoid unnecessary arithmetic operations can save gas *Submitted by WatchPug, also found by cmichel*
- [G-10] Adding unchecked directive can save gas *Submitted by WatchPug*
- [G-11] Cache storage variables in the stack can save gas *Submitted by WatchPug*
- [G-12] Remove unnecessary variables can make the code simpler and save some gas *Submitted by WatchPug*
- [G-13] Inline unnecessary function can make the code simpler and save some gas *Submitted by WatchPug*
- [G-14] Avoid unnecessary `SafeCast.toInt256()` can save gas *Submitted by WatchPug*
- [G-15] Adding a new method `provider.currentPrice()` can save gas *Submitted by WatchPug*
- [G-16] Avoid unnecessary external calls can save gas *Submitted by WatchPug*
- [G-17] `Token18.sol#balanceOf()` When `isEther()`, `fromTokenAmount()` is unnecessary *Submitted by WatchPug*
- [G-18] `Token18.sol#push()` When `isEther()`, `toTokenAmount()` is unnecessary *Submitted by WatchPug*
- [G-19] `10 ** 18` can be changed to `1e18` and save some gas *Submitted by WatchPug*
- [G-20] `Collateral.sol#maintananceInvariant` can be combined with `collateralnvarant` to save gas *Submitted by 0x0x0x*
- [G-21] At `Product.sol#closeAll`, cache `_position[account]` *Submitted by 0x0x0x*
- [G-22] `NotControllerOwnerError` error not used *Submitted by cmichel*

- [G-08] At settleAccountInternal, check whether the position can be changeable to pre more efficiently *Submitted by 0x0x0x*

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