Kuiper contest
Findings & Analysis Report

2022-01-26

Table of contents

- Overview
  - About C4
  - Wardens
- Summary
- Scope
- Severity Criteria
- High Risk Findings (1)
  - [H-01] Bonding mechanism allows malicious user to DOS auctions
- Medium Risk Findings (8)
  - [M-01] `Basket.sol#mint()` Malfunction due to extra `nonReentrant` modifier
  - [M-02] Setting `Factory.auctionDecrement` to zero causes Denial of Service in `Auction.settleAuction()`
  - [M-03] Setting `Factory.bondPercentDiv` to zero cause Denial of Service in `Auction.bondForRebalance()`
  - [M-04] Fee on transfer tokens do not work within the protocol

    - [M-05] createBasket re-entrancy
    - [M-06] Validations
    - [M-07] Basket becomes unusable if everybody burns their shares
    - [M-08] Auction bonder can steal user funds if bond block is high enough
- Low Risk Findings (12)
- Non-Critical Findings (16)
- Gas Optimizations (34)
- 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 Kuiper contest smart contract system written in Solidity. The code contest took place between October 8—October 10 2021.

Note: this audit contest originally ran under the name `defiProtocol`*.*

Wardens

9 Wardens contributed reports to the Kuiper contest:

1. kenzo
2. pants
3. pauliax
4. WatchPug (jtp and ming)
5. loop

6. yeOlde
7. 0xngndev
8. defsec

This contest was judged by Alex the Entreprenerd.

Final report assembled by itsmetechjay and CloudEllie.

Summary

The C4 analysis yielded an aggregated total of 21 unique vulnerabilities and 71 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, 8 received a risk rating in the category of MEDIUM severity, and 12 received a risk rating in the category of LOW severity.

C4 analysis also identified 16 non-critical recommendations and 34 gas optimizations.

Scope

The code under review can be found within the C4 Kuiper contest repository, and is composed of 11 smart contracts written in the Solidity programming language and includes 552 lines of Solidity code and 460 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 (1)

[H-01] Bonding mechanism allows malicious user to DOS auctions

Submitted by kenzo.

A malicious user can listen to the mempool and immediately bond when an auction starts, without aim of settling the auction. As no one can cancel his bond in less than 24h, this will freeze user funds and auction settlement for 24h until his bond is burned and the new index is deleted. The malicious user can then repeat this when a new auction starts.

While the malicious user will have to pay by having his bond burned, it might not be enough of a detriment for the DOS of the basket.

Impact

Denial of service of the auction mechanism. The malicious user can hold the basket "hostage" and postpone or prevent implementing new index. The only way to mitigate it would be to try to front-run the malicious user, obviously not ideal.

Proof of Concept

publishAllIndex: https://github.com/code-423n4/2021-09-defiProtocol/blob/52b74824c42acbcd64248f68c40128fe3a82caf6/contracts/contracts/Basket.sol#L170

- The attacker would listen to this function / PublishedNewIndex event and upon catching it, immediately bond the auction.

- The publisher has no way to burn a bond before 24h has passed. But even if he could, it would not really help as the attacker could just bond again (though losing funds in the process).

`settleAuction` : https://github.com/code-423n4/2021-09-defiProtocol/blob/52b74824c42acbcd64248f68c40128fe3a82caf6/contracts/contracts/Auction.sol#L79

- Only the bonder can settle.

`bondBurn` : https://github.com/code-423n4/2021-09-defiProtocol/blob/52b74824c42acbcd64248f68c40128fe3a82caf6/contracts/contracts/Auction.sol#L111

- Can only burn 24h after bond.

Tools Used

Manual analysis, hardhat.

Recommended Mitigation Steps

If we only allow one user to bond, I see no real way to mitigate this attack, because the malicious user could always listen to the mempool and immediately bond when an auction starts and thus lock it.

So we can change to a mechanism that allows many people to bond and only one to settle; but at that point, I see no point to the bond mechanism any more. So we might as well remove it and let anybody settle the auction.

With the bond mechanism, a potential settler would have 2 options:

- Bond early: no one else will be able to bond and settle, but the user would need to leave more tokens in the basket (as newRatio starts large and decreases in time)
- Bond late: the settler might make more money as he will need to leave less tokens in the basket, but he risks that somebody else will bond and settle before him.

Without a bond mechanism, the potential settler would still have these equivalent 2 options:

- Settle early: take from basket less tokens, but make sure you win the auction
- Settle late: take from basket more tokens, but risk that somebody settles before you

So that's really equivalent to the bonding scenario.

I might be missing something but at the moment I see no detriment to removing the bonding mechanism.

frank-beard (Kuiper) acknowledged

itsmetechjay (organizer) commented:

> Warden apologizes for linking the code of the previous defiProtocol contest, however, these lines are not changed in the new contest.

Alex the Entreprenerd (judge) commented:

> I fully agree with this, anyone can grief the rest of the funds by bonding.

> Personally, this is so easy to execute that I have to raise the severity to High, as it means that every single time there's a benefit to performing a DOS, any malicious actor just has to bond to do it

Alex the Entreprenerd (judge) commented:

> The sponsor may want to consider de-prioritizing bonding to rebalance, by allowing multiple users to bond and rebalance at the same time (or by having bond and rebalance happen at the same time)

Alex the Entreprenerd (judge) commented:

> After thinking about it, I had put into question the high severity because of the "extractability of value". However because this finding allows to effectively DOS the auction, at any time, I still believe High Risk to be the correct severity

Medium Risk Findings (8)

[M-01] `Basket.sol#mint()` Malfunction due to extra `nonReentrant` modifier

Submitted by WatchPug, also found by kenzo, pants, and pauliax.

https://github.com/code-423n4/2021-10-defiprotocol/blob/7ca848f2779e2e64ed0b4756c02f0137ecd73e50/contracts/contracts/Basket.sol#L83-L88

```
    function mint(uint256 amount) public nonReentrant override {
        mintTo(amount, msg.sender);
    }

    function mintTo(uint256 amount, address to) public nonReentrant
        require(auction.auctionOngoing() == false);
```

The `mint()` method is malfunction because of the extra `nonReentrant` modifier, as `mintTo` already has a `nonReentrant` modifier.

Recommendation

Change to:

```
    function mint(uint256 amount) public override {
        mintTo(amount, msg.sender);
    }
```

frank-beard (Kuiper) confirmed

Alex the Entreprenerd (judge) commented:

> Mint is factually broken, definitely an oversight. I don't think high severity is correct here though as since no-one can mint, no funds are at risk. I'll go with medium severity as per the docs:

```
2 — Med: Assets not at direct risk, but the function of the **pr
```

[M-02] Setting `Factory.auctionDecrement` to zero causes Denial of Service in `Auction.settleAuction()`

Submitted by pants.

The function `Factory.setAuctionDecrement()` allows the owner to set the state variable `Factory.auctionDecrement` to zero.

Impact

If `Factory.auctionDecrement` equals zero then the function `Auction.settleAuction()` will always revert due to a division by zero:

```
uint256 b = (bondBlock - auctionStart) * BASE / factory.auctionD
```

Tool Used

Manual code review.

Recommended Mitigation Steps

Add an appropriate require statement to the function `Factory.setAuctionDecrement()` to disallow setting `Factory.auctionDecrement` to zero.

frank-beard (Kuiper) acknowledged

Alex the Entreprenerd (judge) commented:

> I agree with the finding, because this shows a way to DOS the protocol, given specific conditions, I will raise the severity to medium

[M-03] Setting `Factory.bondPercentDiv` to zero cause Denial of Service in `Auction.bondForRebalance()`

Submitted by pants.

The function `Factory.setBondPercentDiv()` allows the owner to set the state variable `Factory.bondPercentDiv` to zero.

Impact

If `Factory.bondPercentDiv` equals zero then the function `Auction.bondForRebalance()` will always revert due to a division by zero:

```
    bondAmount = basketToken.totalSupply() / factory.bondPercentDiv(
```

Tool Used

Manual code review.

Recommended Mitigation Steps

Add an appropriate require statement to the function `Factory.setBondPercentDiv()` to disallow setting `Factory.bondPercentDiv` to zero.

frank-beard (Kuiper) acknowledged

Alex the Entreprenerd (judge) commented:

> Similarly to #24 agree with finding and raising to medium

[M-04] Fee on transfer tokens do not work within the protocol

Submitted by anon.

Fee on transfer tokens transfer less tokens in than what would be expect. This means that the protocol request incorrect amounts when dealing with these tokens.

https://github.com/code-423n4/2021-10-defiprotocol/blob/7ca848f2779e2e64ed0b4756c02f0137ecd73e50/contracts/contracts/Basket.sol#L256

The protocol should use stored token balances instead of transfer for calculating amounts.

frank-beard (Kuiper) acknowledged

Alex the Entreprenerd (judge) commented:

> I agree with the finding and the severity. Given a `feeOnTransferToken`, the accounting of the protocol will break.

> because this is dependent on an external condition (using `feeOnTransferToken`) this is a medium severity finding

[M-05] createBasket re-entrancy

Submitted by pauliax.

Impact

function `createBasket` in Factory should also be `nonReentrant` as it interacts with various tokens inside the loop and these tokens may contain callback hooks.

Recommended Mitigation Steps

Add `nonReentrant` modifier to the declaration of createBasket.

frank-beard (Kuiper) confirmed

Alex the Entreprenerd (judge) commented:

> I agree that since the function can potentially interact with any ERC20like token, the function is vulnerable to re-entrancy, because we don't have any specific POC for an attack, this is a medium severity finding

[M-06] Validations

Submitted by pauliax.

Impact

function `setBondPercentDiv` should validate that `newBondPercentDiv` is not 0, or `bondForRebalance` will experience division by zero error otherwise. If you want to allow 0 values, then `bondForRebalance` should accommodate for such a possibility.

function `addBounty` should check that `amount > 0` to prevent empty bounties.

function `setMinLicenseFee` should validate that it is not over 100%: `newMinLicenseFee <= BASE`.

function `mintTo` should validate that 'to' is not an empty address (0x0) to prevent accidental loss of tokens.

function `validateWeights` should validate that token is not this `basket erc20: require(\_tokens\[i] != address(this));`

function `proposeBasketLicense` could validate that `tokenName` and `tokenSymbol` are not empty.

function `setBondPercentDiv` should validate that `newBondPercentDiv > 1`, otherwise it may become impossible to `bondBurn` because then `bondAmount = totalSupply` and calculation of newIbRatio will produce division by zero runtime error. Of course, this value is very unlikely but still would be nice to enforce this algorithmically.

Recommended Mitigation Steps

Consider applying suggested validations to make the protocol more robust.

frank-beard (Kuiper) acknowledged and disagreed with severity

Alex the Entreprenerd (judge) commented:

> I agree with the warden, adding these checks will provide additional safety guarantees to protocol users (by limiting owner privileges)

> Additionally, some of these setters can be used to DOS the protocol, as such this is a valid medium severity finding

[M-07] Basket becomes unusable if everybody burns their shares

Submitted by kenzo.

While handling the fees, the contract calculates the new `ibRatio` by dividing by `totalSupply`. This can be 0 leading to a division by 0.

Impact

If everybody burns their shares, in the next mint, `totalSupply` will be 0, `handleFees` will revert, and so nobody will be able to use the basket anymore.

Proof of Concept

Vulnerable line: https://github.com/code-423n4/2021-09-defiProtocol/blob/52b74824c42acbcd64248f68c40128fe3a82caf6/contracts/contracts/Basket.sol#L124 You can add the following test to Basket.test.js and see that it reverts (..after you remove “nonReentrant” from “mint”, see other issue):

```
it("should divide by 0", async () => {
await basket.connect(addr1).burn(await basket.balanceOf(addr1.ac
await basket.connect(addr2).burn(await basket.balanceOf(addr2.ac
await UNI.connect(addr1).approve(basket.address, ethers.BigNumbe
await COMP.connect(addr1).approve(basket.address, ethers.BigNumb
await AAVE.connect(addr1).approve(basket.address, ethers.BigNumb
await basket.connect(addr1).mint(ethers.BigNumber.from(1));
});
```

Tools Used

Manual analysis, hardhat.

Recommended Mitigation Steps

Add a check to `handleFees: if totalSupply= 0`, you can just return, no need to calculate new `ibRatio` / fees. You might want to reset `ibRatio` to BASE at this point.

frank-beard (Kuiper) confirmed

itsmetechjay (organizer) commented:

> Warden apologizes for linking the code of the previous Kuiper contest, however these lines are not changed in the new contest.

Alex the Entreprenerd (judge) commented:

> Burning all shares will bring `totalSupply` to 0, which will cause `handleFees` to revert. The finding is valid and I agree with the severity as this can happen if every share holder decides to burn

> Personally I think moving `handleFees` to a separate external call would be a simple mitigation (which also reduces gas cost for all users) Alternatively, the sponsor could always mint a few shares for each basket, to ensure `totalSupply` never reaches 0

[M-08] Auction bonder can steal user funds if bond block is high enough

Submitted by kenzo.

After an auction has started, as time passes and according to the `bondBlock`, `newRatio` (which starts at 2*ibRatio) gets smaller and smaller and therefore less and less tokens need to remain in the basket. This is not capped, and after a while, `newRatio` can become smaller than current `ibRatio`.

Impact

If for some reason nobody has bonded and settled an auction and the publisher didn't stop it, a malicious user can wait until `newRatio < ibRatio`, or even until

`newRatio` ~= 0 (for an initial `ibRatio` of ~1e18 this happens after less than 3.5 days after auction started), and then bond and settle and steal user funds.

Proof of Concept

These are the vulnerable lines: https://github.com/code-423n4/2021-10-defiprotocol/blob/main/contracts/contracts/Auction.sol#L95:#L105

```
    uint256 a = factory.auctionMultiplier() * basket.ibRatio();
    uint256 b = (bondBlock - auctionStart) * BASE / factory.auctionD
    uint256 newRatio = a - b;

    (address[] memory pendingTokens, uint256[] memory pendingWeights
    IERC20 basketAsERC20 = IERC20(address(basket));

    for (uint256 i = 0; i < pendingWeights.length; i++) {
        uint256 tokensNeeded = basketAsERC20.totalSupply() * pending
        require(IERC20(pendingTokens[i]).balanceOf(address(basket))
    }
```

The function verifies that `pendingTokens[i].balanceOf(basket) >= basketAsERC20.totalSupply() * pendingWeights[i] * newRatio / BASE / BASE`. This is the formula that will be used later to mint/burn/withdraw user funds. As bondBlock increases, newRatio will get smaller, and there is no check on this. After a while we'll arrive at a point where `newRatio ~= 0`, so `tokensNeeded = newRatio*(...) ~= 0`, so the attacker could withdraw nearly all the tokens using outputTokens and outputWeights, and leave just scraps in the basket.

Tools Used

Manual analysis, hardhat.

Recommended Mitigation Steps

Your needed condition/math might be different, and you might also choose to burn the bond while you're at it, but I think at the minimum you should add a sanity check in `settleAuction`:

```
require (newRatio > basket.ibRatio());
```

Maybe you would require `newRatio` to be > BASE but not sure.

frank-beard (Kuiper) confirmed

Alex the Entreprenerd (judge) commented:

> Would need to confirm with sponsor: Isn't the point of `settleAuction` to be incentivized by offering a discount over time? If you're offering a discount, then by definition `newRatio` will be less than `basket.ibRatio`

Alex the Entreprenerd (judge) commented:

> Have yet to hear back from the sponsor.

> The more I think about it, the more this is a property of the discounted auction, the basket token can be bought for less, and that creates MEV opportunities. This is the economic incentive for bonding and settling the auction (else why do it?)

> On the other hand there may be situation where the price decay is so aggressive, and the bonding gives a 24hrs privilege for settling which could create a situation where the bonder is incentivized to wait.

> Locking the discount at the time of bonding, or forcing to bond and settle at the same time may mitigate this (creating effectively a dutch auction for the discounted basket tokens).

> Given what I understand about the system, I would argue that:

- Given a specific `auctionDecrement` and a basket big enough, the bonder has a 24 hour window to maximize the value they can extract, which can end up being too much from what the developer / users may expect.

> What do you think @frank-beard ?

frank-beard (Kuiper) commented:

> Agree with @Alex the Entreprenerd. The purpose of the auction is to create an opportunity for participants to rebalance a basket if it is in their interest. However the warden is correct that there can be issues if the `ibRatio` drops too low, or even to 0, which would effectively allow someone to steal funds. We plan to mitigate this by having the auction have a minimum `ibRatio` at which it can be settled.

Alex the Entreprenerd (judge) commented:

> The vulnerability is reliant on `auctionDecrement` being impactful enough on a 24 hr window (time in which bonder has privileged ability to settle or just stall)

> If `auctionDecrement` gives a steep enough discount then it can creates scenarios where the bonder can get access to the underlying at no price. This can be taken to the extreme of rebalancing the entire basket (taking all funds)

> Because of the openness of the protocol am inclined to rate this a medium severity, however, the sponsor needs to be aware that every single time this scenario shows itself it will be abused against the protocol users.

> Mitigation can happen by either setting a minimum `ibRatio` or by allowing multiple entities to bond and settle at the same, creating a “prisoners dilemma” dutch auction that effectively motivates actors to rebalance as early as economically feasible

Low Risk Findings (12)

- [L-01] Sensitive variables should not be able to be changed easily *Submitted by anon.*
- [L-02] Input Validation on Factory.sol *Submitted by anon.*
- [L-03] How much to approve before calling mintTo *Submitted by pauliax.*
- [L-04] Array out-of-bounds error in `Auction` *Submitted by pants.*
- [L-05] Array out-of-bounds errors in `Factory` *Submitted by pants.*
- [L-06] Tests are broken *Submitted by kenzo.*
- [L-07] Inaccurate log emitted at deleteNewIndex *Submitted by kenzo.*

- [L-08] `Basket.sol` should use the Upgradeable variant of OpenZeppelin Contracts *Submitted by WatchPug.*
- [L-09] `Basket.sol#changePublisher()` Insufficient input validation *Submitted by WatchPug.*
- [L-10] `Factory.proposeBasketLicense()` and `IFactory.proposeBasketLicense()` accept arguments with different data locations *Submitted by pants.*
- [L-11] `Auction.settleAuction()` and `IAuction.settleAuction()` accept arguments with different data locations *Submitted by pants.*
- [L-12] `Basket.publishNewIndex()` and `IBasket.publishNewIndex()` accept arguments with different data locations *Submitted by pants.*

Non-Critical Findings (16)

- [N-01] Remove hardhat import *Submitted by anon, also found by WatchPug.*
- [N-02] `nonReentrant` modifier should be used before any other modifier *Submitted by pants.*
- [N-03] Events in `IAuction` don't use the `indexed` keyword *Submitted by pants.*
- [N-04] Inconsistent naming of a function's argument in `Factory` *Submitted by pants.*
- [N-05] Lack of Documentation on key functions *Submitted by anon.*
- [N-06] Open TODOs in `Basket` *Submitted by pants.*
- [N-07] Open TODOs in `Auction` *Submitted by pants.*
- [N-08] Open TODOs in `IFactory` *Submitted by pants.*
- [N-09] Open TODOs in `IBasket` *Submitted by pants.*
- [N-10] Require statements without messages in `Factory` *Submitted by pants.*
- [N-11] Require statements without messages in `Basket` *Submitted by pants.*
- [N-12] Require statements without messages in `Auction` *Submitted by pants.*
- [N-13] Open TODOs in `Factory` *Submitted by pants.*

- [N-14] Missing events for owner only functions that change critical parameters *Submitted by defsec.*
- [N-15] Missing events for basket only functions that change critical parameters *Submitted by defsec.*
- [N-16] `Basket.sol` should have methods to cancel pending changes *Submitted by WatchPug.*

Gas Optimizations (34)

- [G-01] Minimize Storage Slots (Auction.sol) *Submitted by yeOlde, also found by 0xngndev and kenzo.*
- [G-02] Unused Named Returns Can Be Removed *Submitted by yeOlde, also found by pants.*
- [G-03] Increase optimizer runs *Submitted by anon.*
- [G-04] uint256 can be lowered to unitX with X < 256 in some cases *Submitted by anon.*
- [G-05] Unchecked modifiers should be used when over/under-flow isnt an issue to save gas *Submitted by anon.*
- [G-06] Uninitialized variables are automatically set to 0 *Submitted by anon, also found by kenzo, pauliax, and WatchPug.*
- [G-07] Set initial value for lastFee *Submitted by pauliax.*
- [G-08] Cache factory.ownerSplit() *Submitted by pauliax.*
- [G-09] Cache basketAsERC20.totalSupply() *Submitted by pauliax, also found by kenzo.*
- [G-10] There may be no bounties or user is not interested in any of them *Submitted by pauliax.*
- [G-11] Empty `else if` block in `Basket.publishNewIndex()` *Submitted by pants.*
- [G-12] Unnecessary `SLOAD`s and `MLOAD`s in for-each loops *Submitted by pants.*
- [G-13] Unnecessary `SLOAD`s in `Factory` *Submitted by pants.*
- [G-14] Unnecessary `SLOAD`s in `Basket` *Submitted by pants.*
- [G-15] Unnecessary `SLOAD`s in `Auction` *Submitted by pants.*

- [G-16] Unnecessary require statement in `Auction.initialize()` and `Basket.initialize()` *Submitted by pants, also found by WatchPug.*
- [G-17] Unnecessary checked arithmetic in for loops *Submitted by pants.*
- [G-18] Unnecessary checked arithmetic in `Basket.handleFees()` *Submitted by pants.*
- [G-19] Unnecessary checked arithmetic in `Auction.addBounty()` and `Factory.proposeBasketLicense()` *Submitted by pants.*
- [G-20] Unnecessary checked arithmetic in `Auction.settleAuction()`, `Auction.bondBurn()`, `Basket.changePublisher()`, `Basket.changeLicenseFee()` and `Basket.publishNewIndex()` *Submitted by pants.*
- [G-21] Prefix increament is cheaper than postfix increament *Submitted by pants.*
- [G-22] Unnecessary cast in `Basket.onlyPublisher()` *Submitted by pants.*
- [G-23] Unnecessary cast in `Factory.proposeBasketLicense()` *Submitted by pants.*
- [G-24] `internal` function in `Auction` can be `private` *Submitted by pants.*
- [G-25] `public` functions in `Factory` can be `external` *Submitted by pants.*
- [G-26] `public` functions in `Basket` can be `external` *Submitted by pants.*
- [G-27] `public` functions in `Auction` can be `external` *Submitted by pants.*
- [G-28] State variables in `Factory` can be `immutable` *Submitted by pants.*
- [G-29] Comparisons to boolean constant *Submitted by loop.*
- [G-30] Basket: No need for initialized variable *Submitted by kenzo.*
- [G-31] Unnecessary new list in Basket's validateWeights() *Submitted by kenzo.*
- [G-32] Restore state to 0 if not needed anymore *Submitted by kenzo.*
- [G-33] `Basket.sol#changePublisher()` Remove redundant assertion can save gas *Submitted by WatchPug.*
- [G-34] `Basket.sol#changeLicenseFee()` Remove redundant check can save gas *Submitted by WatchPug.*

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