Introducing Code4rena Blue: Dedicated defense. Competitive bounties. Independent judging.
Learn more →

Kuiper contest
Findings & Analysis Report

2022-05-02

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 Kuiper smart contract system written in Solidity. The audit contest took place between December 8—December 10 2021.

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

Wardens

27 Wardens contributed reports to the Kuiper contest:

1. kenzo
2. 0x0x0x
3. WatchPug (jtp and ming)
4. GiveMeTestEther
5. gzeon
6. TomFrenchBlockchain
7. broccolirob
8. Ruhum
9. 0v3rf10w
10. gpersoon
11. robee
12. Jujic
13. yeOlde
14. rishabh
15. Meta0xNull
16. cmichel
17. saian
18. neslinesli93
19. danb
20. pmerkleplant
21. pedroais
22. bw
23. sirhashalot
24. hagrid
25. 0x1f8b
26. BouSalman

This contest was judged by 0xleastwood.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 20 unique vulnerabilities and 89 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, 11 received a risk rating in the category of MEDIUM severity, and 8 received a risk rating in the category of LOW severity.

C4 analysis also identified 29 non-critical recommendations and 40 gas optimizations.

Scope

The code under review can be found within the C4 Kuiper contest repository, and is composed of 3 smart contracts written in the Solidity programming language and includes 588 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] Wrong fee calculation after `totalSupply` was 0

Submitted by kenzo

`handleFees` does not update `lastFee` if `startSupply == 0`. This means that wrongly, extra fee tokens would be minted once the basket is resupplied and `handleFees` is called again.

Impact

Loss of user funds. The extra minting of fee tokens comes on the expense of the regular basket token owners, which upon withdrawal would get less underlying than their true share, due to the dilution of their tokens' value.

Proof of Concept

Scenario:

- All basket token holders are burning their tokens. The last burn would set totalSupply to 0.
- After 1 day, somebody mints basket tokens.

`handleFees` would be called upon mint, and would just return since totalSupply == 0. Note: It does not update `lastFee`.

```
    } else if (startSupply == 0) {
                return;
```

[Basket.sol#L136:#L137]()

- The next block, somebody else mints a token. Now `handleFees` will be called and will calculate the fees according to the current supply and the time diff between now and `lastFee`:

```
    uint256 timeDiff = (block.timestamp - lastFee);
```

Basket.sol#L139

But as we saw, `lastFee` wasn't updated in the previous step. `lastFee` is still the time of 1 day before - when the last person burned his tokens and the basket supply was 0. So now the basket will mint fees as if a whole day has passed since the last calculation, but actually it only needs to calculate the fees for the last block, since only then we had tokens in the basket.

Recommended Mitigation Steps

Set `lastFee = block.timestamp` if `startSupply == 0`.

frank-beard (Kuiper) confirmed

0xleastwood (judge) commented:

> The issue can be outlined as follows:

- A user interacts with the basket and mints some amount of tokens, which sets `lastFee = block.timestamp`.
- The same user decides to exit the basket and burn their tokens.
- Some amount of time passes and another user enters the basket, but `handleFees()` did not set `lastFee = block.timestamp`. As a result, fees are charged on the user's deposit for the entire time that the basket was inactive for.

> It seems that the basket is severely flawed in calculating fees on partially inactive baskets. This puts users' funds at direct risk of being lost. Malicious publishers can setup baskets as a sort of honeypot to abuse this behaviour.

> This was an interesting find! Kudos to the warden.

Medium Risk Findings (11)

[M-01] Missing cap on `LicenseFee`

Submitted by gzeon, also found by 0x0x0x and TomFrenchBlockchain

There is no cap on `LicenseFee`. While change of `LicenseFee` is under 1 day timelock, introducing a `maxLicenseFee` can improve credibility by removing the "rug" vector. There is a `minLicenseFee` in the contracts, while imo make little sense to have `minLicenseFee` but not `maxLicenseFee`.

An incorrectly set `LicenseFee` can potentially lead to over/underflow in Basket.sol#L140-141 which is used in most of the function.

Proof of Concept

Basket.sol#L177
Factory.sol#L77
Basket.sol#L49

Recommended Mitigation Steps

Define a `maxLicenseFee`

frank-beard (Kuiper) acknowledged, but disagreed with High severity and commented:

> Generally it is intended for the publishers to act correctly and the timelock is intended to prevent incorrect values from making it all the way through, however there is validity in reducing how the fee can be modified, such as reducing how much any one fee change can change the fee. I would consider this a low risk issue.

0xleastwood (judge) decreased severity to Medium and commented:

> It seems like changes to `licenseFee` could potentially brick the contract as `handleFees()` underflows, preventing users from minting/burning tokens. I'd deem this as `medium` severity due to compromised protocol availability.

[M-02] Publisher can lock all user funds in the `Basket` in order to force a user to have their bond burned

Submitted by TomFrenchBlockchain, also found by WatchPug

All user funds in a basket being held hostage by the publisher

Proof of Concept

The `Basket` publisher can propose an auction in order to set new tokens and weights with a 1 day timelock.

[Basket.sol#L216-L244]

As part of this call they can set the `minIbRatio` variable which determines what the maximum slippage on the auction is allowed to be. If it's set to the current `IbRatio` then the Basket accepts no slippage.

The publisher can choose to set `minIbRatio = type(uint256).max` which will prevent any auction bids from being successful, locking the basket in the auction state.

It's not possible to enter or exit the basket while an auction is going on, so any users who hold any funds in the basket are forced to take the only option to kill the auction available to them.

[Basket.sol#L91-L119]

If a user makes a bond and then waits a day to then call `Auction.bondBurn`, it will reset the auction and allow users to withdraw but it requires 0.25% of the supply of the basket token to be burned.

[Auction.sol#L121-L134]

One of the basket's users is then forced to give up some of their assets to secure the release of the remaining assets in the basket (for a 24hr period until the publisher starts a new auction).

This attack can be launched at any time with only 24 hours warning. This is a very short amount of time which near guarantees that if other users hold funds in the basket that not all of them will successfully withdraw in that time and so will have funds locked.

Recommended Mitigation Steps

Again this is tricky to mitigate as there are legitimate scenarios where we would expect the `ibRatio` to increase. e.g. a basket containing WBTC being changed to contain USDC as each basket token should be worth much more USDC than it was in terms of WBTC.

To be frank the entire auction mechanism is a bit shaky as it doesn't account for changes in the values of the tokens over time.

frank-beard (Kuiper) acknowledged, but disagreed with High severity and commented:

> This is where community action and the timelock should mitigate attacks of these types. Users should be able to hold publishers accountable for their rebalances, whether that is through a dao or other means. We acknowledge there is some level of trust required between the user and the publisher however this is also intentional, for the types of products this protocol is for.

0xleastwood (judge) decreased severity to Medium and commented:

> Because a timelock is used in this instance, exploiting this issue proves more difficult and requires that the `publisher` is malicious. As we are dealing with an abuse of privileges, I think this fits the criteria of a `medium` severity issue as the issue can only be exploited by a trusted account.

0xleastwood (judge) commented:

> Actually I'm not sure if I understand the exploit:

- The publisher must wait a day to call `Basket.publishNewIndex()`, setting new values and starting an auction. In that time, users are free to exit the protocol as they wish.

- However, users who were not able to exit in time are obligated to call `Auction.bondForRebalance()` in order to unlock the basket's underlying assets. But because `Auction.settleAuction()` will always revert, this user forfeits their bond to unlock the rest of their tokens.

> In this case, I see this as an abuse of the publisher's privileges and lack of oversight by the users. `medium` severity seems correct in this situation.

[M-03] `Basket.sol#auctionBurn` calculates `ibRatio` wrong

Submitted by 0x0x0x

The function is implemented as follows:

```
    function auctionBurn(uint256 amount) onlyAuction nonReentrant ex
            uint256 startSupply = totalSupply();
            handleFees(startSupply);
            _burn(msg.sender, amount);

            uint256 newIbRatio = ibRatio * startSupply / (startSuppl
            ibRatio = newIbRatio;

            emit NewIBRatio(newIbRatio);
            emit Burned(msg.sender, amount);
        }
```

When `handleFees` is called, `totalSupply` and `ibRatio` changes accordingly, but for `newIbRatio` calculation tokens minted in `handleFees` is not included. Therefore, `ibRatio` is calculated higher than it should be. This is dangerous, since last withdrawing user(s) lose their funds with this operation. In case this miscalculation happens more than once, `newIbRatio` will increase the miscalculation even faster and can result in serious amount of funds missing. At each time `auctionBurn` is called, at least 1 day (auction duration) of fees result in this miscalculation. Furthermore, all critical logic of this contract is based on `ibRatio`, this behaviour can create serious miscalculations.

Mitigation step

Rather than

```
uint256 newIbRatio = ibRatio * startSupply / (startSupply - amount);
```

A practical solution to this problem is calculating `newIbRatio` as follows:

```
    uint256 supply = totalSupply();
    uint256 newIbRatio = ibRatio * (supply + amount) / supply;
```

frank-beard (Kuiper) confirmed

0xleastwood (judge) decreased severity to Medium and commented:

> The warden has identified an issue whereby `newIbRatio` uses an incorrect `startSupply` variable which is under-represented. As new tokens may be minted in `handleFees()`, this will lead to an incorrect `ibRatio` which is used in all other areas of the protocol. A lower `ibRatio` causes `pushUnderlying()` and `pullUnderlying()` to be improperly accounted for. As a result, burning basket tokens will redeem a smaller amount of underlying tokens and minting basket tokens will require a smaller amount of underlying tokens.

> This causes the protocol to leak value from all basket token holders but it does not allow assets to be stolen. As such, I think this is better put as a `medium` severity issue.

[M-04] Reentrancy vulnerability in `Basket` contract's `initialize()` method.

Submitted by broccolirob

A malicious "publisher" can create a basket proposal that mixes real ERC20 tokens with a malicious ERC20 token containing a reentrancy callback in it's `approve()` method. When the `initialize()` method is called on the newly cloned `Basket` contract, a method called `approveUnderlying(address(auction))` is called, which would trigger the reentrancy, call `initialize()` again, passing in altered critical values such as `auction` and `factory`, and then removes its self from `proposal.tokens` and `proposal.weights` so it doesn't appear in the token list to basket users.

[Basket.sol#L44-L61]

Impact

`Auction` and `Factory` can be set to custom implementations that do malicious things. Since all baskets and auctions are clones with their own addresses, this fact would be difficult for users to detect. `Auction` controls ibRatio, which a malicious version could send back a manipulated value to `Basket`, allowing the malicious "publisher" to burn basket tokens till all users underlying tokens are drained.

Tools Used

Manual review and Hardhat.

Recommended Mitigation Steps

Since `Basket` inherits from `ERC20Upgradeable` the `initializer` modifier should be available and therefore used here. It has an `inititializing` variable that would prevent this kind of reentrancy attack.

[frank-beard (Kuiper) confirmed]

[0xleastwood (judge) decreased severity to Medium and commented]:

> While the warden is correct, a malicious publisher could re-enter the `Basket.initialize()` function and overwrite `factory` and `auction` with their own addresses, this does not lead to a direct loss of funds for users. It would require that users interact with their malicious contracts which is entirely possible if baskets created via the factory are deemed as trusted. I think this fits the criteria of a `medium` severity issue.

[M-05] Change in `auctionMultiplier/auctionDecrement`

change profitability of auctions and factory can steal all tokens from a basket abusing it

Submitted by 0x0x0x

When factory changes `auctionMultiplier` or `auctionDecrement` profitability of bonded auctions change. There is no protection against this behaviour. Furthermore, factory owners can decide to get all tokens from baskets where they are bonded for the auction.

Proof of concept

1- Factory owners call `bondForRebalance` for an auction.

2- Factory owners sets `auctionMultiplier` as 0 and `auctionDecrement` as maximum value

3- `settleAuction` is called. `newRatio = 0`, since `a = b = 0`. All tokens can be withdrawn with this call, since `tokensNeeded = 0`.

Extra notes

Furthermore, even the factory owners does not try to scam users. In case `auctionMultiplier` or `auctionDecrement` is changed, all current `auctionBonder` from `Auctions` can only call `settleAuction` with different constraints. Because of different constraints, users/bonder will lose/gain funds.

Mitigation step

Save `auctionDecrement` and `auctionMultiplier` to global variables in `Auction.sol`, when `startAuction` is called.

frank-beard (Kuiper) confirmed and commented:

> Adding in protection from the global governance is definitely important.

0xleastwood (judge) decreased severity to Medium and commented:

> The warden has identified an issue whereby the factory owner can rug-pull baskets through the re-balancing mechanism. Because `newRatio = 0`, the basket improperly checks the tokens needed in the contract. However, this assumes that the factory owner is malicious which satisfies `medium` severity due to assets not being at direct risk.

> The sponsor has decided to add additional protections (potentially via timelock) to mitigate this issue.

> This rug-pull is made even more difficult by the fact that `newRatio` must be `>= minIbRatio`. Because `minIbRatio` is behind timelock, I think this rug vector is unlikely or at least can only be used to steal a fixed amount of funds.

[M-06] Basket can be fully drained if the auction is settled within a specific block

Submitted by Ruhum

The `settleAuction()` function allows someone to settle the auction by transferring funds in a way that the new pending index is fulfilled. As a reward, they are able to take out as many tokens as they want as long as the pending index is fulfilled after that. The function verifies that the basket has received everything it wanted using the following logic:

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

The attack vector here is to manipulate `tokensNeeded` to be 0. That way we can drain the basket completely without the function reverting.

For that, we manipulate `newRatio` to be 0 then the whole thing will be 0. `newRatio` is defined as:

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

There's 1 value the attacker controls, `bondBlock`. That value is the block in which the `bondForRebalance()` function was triggered. So the goal is to get `newRatio` to be 0. With the base settings of the contract:

- auctionMultiplier == 2
- ibRatio == 1e18
- BASE == 1e18
- auctionDecrement == 10000

`bondBlock` has to be `auctionStart + 20000`. Meaning, the `bondForRebalance()` function has to be triggered exactly 20000 blocks after the action was started. That would be around 3 1/2 days after auction start.

At that point, `newRatio` is 0, and thus `tokensNeeded` is 0. The only thing left to do is to call `settleAuction()` and pass the basket's tokens and balance as the output tokens and weight.

Proof of Concept

Here's a test implementing the above scenario as a test. You can add it to `Auction.test.js`.:

```
        it.only("should allow me to steal funds", async() => {
          // start an auction
          let NEW_UNI_WEIGHT = "2400000000000000000";
          let NEW_COMP_WEIGHT = "2000000000000000000";
          let NEW_AAVE_WEIGHT = "400000000000000000";

          await expect(basket.publishNewIndex([UNI.address, COMP.a
              [NEW_UNI_WEIGHT, NEW_COMP_WEIGHT, NEW_AAVE_WEIGHT],
          await increaseTime(60 * 60 * 24)
          await increaseTime(60 * 60 * 24)
          await expect(basket.publishNewIndex([UNI.address, COMP.a
            [NEW_UNI_WEIGHT, NEW_COMP_WEIGHT, NEW_AAVE_WEIGHT], 1)
          let auctionAddr = await basket.auction();
          let auction = AuctionImpl.attach(auctionAddr);

          ethers.provider.getBlockNumber();
          // increase the block number for `bondBlock - auctionSta
          // When that's the case, the result of `newRatio` in `se
          // is `0`. And that means `tokensNeeded` is 0. Which mea
          // we can take out all the tokens we want using the `out
          // without having to worry about basket's balance at the
          // The math changes depending on the settings of the fac
          // Basket contract. But, the gist is that you try to get
          // The only values you can control as a attacker is the
          // was started.
          for (let i = 0; i < 20000; i++) {
            await hre.network.provider.send("evm_mine")
          }
          await basket.approve(auction.address, '5000000000000000'
          await expect(auction.bondForRebalance()).to.be.ok;
          await expect(auction.settleAuction([], [], [], [UNI.addr
        });
```

Again, this test uses the base values. The math changes when the settings change. But, it should always be possible to trigger this attack. The gap between auction start and bonding just changes.

Recommended Mitigation Steps

- Verify that `newRatio != 0`

frank-beard (Kuiper) confirmed and commented:

> This is the reasoning for the minIbRatio value that the publisher sets when rebalancing weights. However we do need a check to make sure that `minIbRatio` is above 0.

0xleastwood (judge) decreased severity to Medium and commented:

> I don't think this deserves a `high` severity rating. There are a number of assumptions made:

- `minIbRatio` has not been set to an expected value.

- The bonded user must be able to wait a certain number of blocks, likely exceeding the maximum amount of time allowed to settle the auction. This is currently set to one day. However, I understand that there might be some time that passes before a user bonds tokens and when the auction started.

> Because this issue is not directly exploitable, I think this behaviour fits the criteria of a `medium` severity issue.

[M-07] `Auction.sol#settleAuction()` Bonder may not be able to settle a bonded auction, leading to loss of funds

Submitted by WatchPug

Auction.sol#L97-L102

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

        (address[] memory pendingTokens, uint256[] memory pendingWei
        require(newRatio >= minIbRatio);
```

In the current implementation, `newRatio` is calculated and compared with `minIbRatio` in `settleAuction()` .

However, if `newRatio` is less than `minIbRatio` , `settleAuction()` will always fail and there is no way for the bonder to cancel and get a refund.

Proof of Concept

Given:

- `bondPercentDiv` = 400
- `basketToken.totalSupply` = 40,000
- `factory.auctionMultiplier` = 2
- `factory.auctionDecrement` = 10,000

- `basket.ibRatio` = 1e18
- p `endingWeights.minIbRatio` = 1.9 * 1e18
- Alice called `bondForRebalance()` `2,000` blocks after the auction started, paid `100` basketToken for the bond;
- Alice tries to `settleAuction()` , it will always fail because `newRatio < minIbRatio`;
- a = 2 * 1e18
- b = 0.2 * 1e18
- newRatio = 1.8 * 1e18;
- Bob calls `bondBurn()` one day after, `100` basketToken from Alice will been burned.

Recommended Mitigation Steps

Move the `minIbRatio` check to `bondForRebalance()` :

```
function bondForRebalance() public override {
    require(auctionOngoing);
    require(!hasBonded);

    bondTimestamp = block.timestamp;
    bondBlock = block.number;

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

    (address[] memory pendingTokens, uint256[] memory pendingWei
    require(newRatio >= minIbRatio);

    IERC20 basketToken = IERC20(address(basket));
    bondAmount = basketToken.totalSupply() / factory.bondPercent
    basketToken.safeTransferFrom(msg.sender, address(this), bonc
    hasBonded = true;
    auctionBonder = msg.sender;

    emit Bonded(msg.sender, bondAmount);
}
```

frank-beard (Kuiper) confirmed

Oxleastwood (judge) decreased severity to Medium and commented:

> While this issue is correct and I think this is a safer way to handle the `require(newRatio >= minIbRatio)` check, there are a few assumptions that are made. For example, it is assumed that the user bonds their tokens without checking `minIbRatio` and a publisher is able to maliciously update `minIbRatio` which must first go through timelock. Based on this, I'm more inclined to downgrade this to `medium` severity as I think this more accurately reflects the threat model.

[M-08] Lost fees due to precision loss in fees calculation

Submitted by kenzo, also found by 0v3rf10w

In fees calculation, division is being used in the midst of the calculation, not at the end of it. This leads to lost precision in fee amount (as solidity doesn't save remainder of division). Division should happen at the end to maintain precision.

Impact

Lost fees. The exact amount depends on the parameters set and being tested. According to a few tests I ran, it seems that in normal usage, 1% of fees are lost. In some cases even 7.5% of fees.

Proof of Concept

Division in the midst of a calculation:

```
    uint256 feePct = timeDiff * licenseFee / ONE_YEAR;
    uint256 fee = startSupply * feePct / (BASE - feePct);

    _mint(publisher, fee * (BASE - factory.ownerSplit()) / BASE);
    _mint(Ownable(address(factory)).owner(), fee * factory.ownerSpli
```

Basket.sol#L140:#L145

It's a little hard to share a POC script as it involves changing the .sol file so I tested it manually. But after moving the division to the end using the mitigation below, I saw 1%-7% increases in fees minted. Usually 1%.

Recommended Mitigation Steps

We want to firstly do all multiplication and lastly do all the division. So remove the usage of feePct and instead set fee to be:

```
uint256 fee = startSupply * licenseFee * timeDiff / ONE_YEAR /
```

frank-beard (Kuiper) confirmed

0xleastwood (judge) commented:

> Nice find! I think this qualifies as `medium` risk due to the protocol regularly leaking value. This can be mitigated by performing division at the very end of the fee calculation.

[M-09] `Basket:handleFees` fee calculation is wrong

Submitted by GiveMeTestEther

The fee calculation on L141 is wrong. It should only get divided by BASE and not (BASE - feePct)

Proof of Concept

This shows dividing only by BASE is correct:
Assumptions:

- BASE is 1e18 accordign to the code
- timeDiff is exactly ONE_YEAR (for easier calculations)
- startSupply is 1e18 (exactly one basket token, also represents 100% in fee terms)
- licenseFee is 1e15 (0.1%)

If we calculate the fee of one whole year and startSupply is one token (1e18, equal to 100%), the fee should be exactly the licenseFee (1e15, 0.1%),

uint256 timeDiff = ONE*YEAR;*
*uint256 feePct = timeDiff * licenseFee / ONE*YEAR;
=> therefore we have: feePct = licenseFee which is 1e15 (0.1%) according to our assumptions
uint256 fee = startSupply * feePct / BASE; // only divide by BASE
=> insert values => fee = 1e18 * licenseFee / 1e18 = licenseFee

This shows the math is wrong:

Assumptions:

- BASE is 1e18 according to the code
- timeDiff is exactly ONE_YEAR (for easier calculations)
- startSupply is 1e18 (exactly one basket token, also represents 100% in fee terms)
- licenseFee is 1e15 (0.1%)

If we calculate the fee of one whole year and startSupply is one token (1e18, equal to 100%), the fee should be exactly the licenseFee (1e15, 0.1%), but the fee is bigger than that.

uint256 timeDiff = ONE*YEAR;*
*uint256 feePct = timeDiff * licenseFee / ONE*YEAR;
=> therefore we have: feePct = licenseFee which is 1e15 (0.1%) according to our assumptions

uint256 fee = startSupply * feePct / (BASE - feePct);
insert the values => fee = 1e18 * 1e15 / (1e18 - 1e15) => (factor out 1e15) => fee = 1e15 * 1e18 / (1e15 * (1e3 - 1) => (cancel 1e15) => 1e18 / (1e3 - 1)

math: if we increase the divisor but the dividend stays the same we get a smaller number e.g. (1 / (2-1)) is bigger than (1 / 2)
apply this here => 1e18 / (1e3 - 1) > 1e18 / 1e3 => 1e18 / (1e3 - 1) > 1e15 this shows that the fee is higher than 1e15

Basket.sol#L133
Basket.sol#L141

Recommended Mitigation Steps

Only divide by BASE.

frank-beard (Kuiper) confirmed

0xleastwood (judge) commented:

> I think this is valid. Calculating fees as `startSupply * feePct / (BASE - feePct)` leads to an overestimation of fees charged on users.

[M-10] Fee calculation is slightly off

Submitted by gzeon

The fee calculation

```
    uint256 timeDiff = (block.timestamp - lastFee);
    uint256 feePct = timeDiff * licenseFee / ONE_YEAR;
    uint256 fee = startSupply * feePct / (BASE - feePct);
```

tries to calculate a fee such that fee/(supply+fee) = %fee using a simple interest formula (i.e. no compounding), this lead to slightly less fee collected when fee are collected more frequently (small timeDiff) vs less frequently (big timeDiff).

Proof of Concept

Basket.sol#L133

frank-beard (Kuiper) acknowledged, but disagreed with Medium severity and commented:

> While this is technically true, the actual precision loss should be very negligible.

Oxleastwood (judge) commented:

> I think any precision loss or value leakage qualifies for a `medium` severity issue. This seems like it would lead to an inconsistent fee calculation and is probably worthwhile fixing long-term.

[M-11] `Basket:handleFees()`: fees are overcharged

Submitted by GiveMeTestEther

The fee calculation is based on the totalSupply of the basket token. But some amount of the totalSupply represents the fees paid to the publisher/ protocol owner. Therefore the fees are “overcharged”: because the fee amount is calculated on a part of already “paid” fees, should only take into account what is “owned” by the users and not the publisher/protocol owner.

Proof of Concept

L141: the fee percent is multiplied by startSupply (=basket token total supply)
L144 & L145: publisher / protocol owner receive basket tokens as fees payment
Basket.sol#L141

Recommended Mitigation Steps

- account the fee amount in a storage variable: uint256 feeAmount;
- subtract feeAmount from startSupply L141: uint256 fee = (startSupply - feeAmount) * feePct / (BASE - feePct); // note the other bug about only dividing by BASE
- add the fee to feeAmount after the calculation: feeAmount += fee;
- if publisher/protocol owner burn basket token, reduce the feeAmount etc.

frank-beard (Kuiper) acknowledged and commented:

> Generally we think that the amount of overcharged fees from this matter will be very negligible in the long term. It is worth noting that the fix proposed would not really solve the problem as not all tokens owned by the publisher/owner may be

> fees as well as they could just transfer those fees to another account and burn from there.

0xleastwood (judge) commented:

> I'm not sure if this is correct. `startSupply` is indeed equal to `totalSupply()` but it is queried before new tokens are minted to the factory owner and publisher. As such, the fees appear to be correctly charged. I'll have @frank-beard confirm this as I may have missed something.

0xleastwood (judge) commented:

> Considering @frank-beard has not replied to this, I'm gonna make the final judgement and keep this open. I think it makes sense to track the fees paid out to the factory owner and publisher and have this amount excluded from the fee calculations. Upon either party burning tokens, this fee amount tracker must be updated accordingly. Due to the added complexity, it is understandable that this issue would be deemed a wontfix by the sponsor.

frank-beard (Kuiper) commented:

> Yeah so I do agree that it does make sense generally to track the fees paid out and exclude, however in this case the impact is very low and yeah the complexity to deal with it doesn't seem worth it.

Low Risk Findings (8)

- [L-01] Extra payments for an auction gets stucks *Submitted by 0x0x0x*
- [L-02] `maxSupply` can be exceeded *Submitted by 0x0x0x*
- [L-03] Bonding doesn't seem to perform any meaningful role and leads to inefficient auctions *Submitted by TomFrenchBlockchain*
- [L-04] Factory can block auctions *Submitted by 0x0x0x*
- [L-05] `Basket.sol` Pending licenseFee may unable to be canceled when current licenseFee is `0` *Submitted by WatchPug*
- [L-06] `Basket.sol` should use the Upgradeable variant of OpenZeppelin Contracts *Submitted by WatchPug*

- [L-07] `Basket.sol#changeLicenseFee()` Unable to set `licenseFee` to 0 *Submitted by WatchPug*
- [L-08] changeLicenseFee() and fees for previous period *Submitted by gpersoon*

Non-Critical Findings (29)

- [N-01] TODO comments should be resolved *Submitted by Jujic, also found by hagrid, Meta0xNull, and robee*
- [N-02] `Ownable` Contract Does Not Implement Two-Step Transfer Ownership Pattern *Submitted by hagrid*
- [N-03] Missing Revert Messages *Submitted by hagrid, also found by robee*
- [N-04] Use of deprecated `safeApprove()` function *Submitted by broccolirob, also found by GiveMeTestEther, gzeon, Meta0xNull, robee, and sirhashalot*
- [N-05] Not verified function inputs of public / external functions *Submitted by robee*
- [N-06] Incorrent visibility for "initialized" variable *Submitted by neslinesli93*
- [N-07] `BasketLicenseProposed` better emit proposal id *Submitted by gzeon*
- [N-08] setAuctionDecrement() Lack of Input Validation May Break Other Function *Submitted by Meta0xNull, also found by gpersoon*
- [N-09] setAuctionMultiplier() Lack of Input Validation May Break Other Function *Submitted by Meta0xNull, also found by gpersoon*
- [N-10] Broken unit tests due to incorrect values *Submitted by bw*
- [N-11] `NewIndexSubmitted` event is not emitted in some case *Submitted by WatchPug*
- [N-12] Factory:setOwnerSplit owner fee split can be set to exactly 20% *Submitted by GiveMeTestEther*
- [N-13] Emit for publishNewIndex / killAuction part *Submitted by gpersoon*
- [N-14] Use of Require statement without reason message *Submitted by 0x1f8b*
- [N-15] Remove override keyword from Auction *Submitted by 0x1f8b*
- [N-16] Remove override keyword from Basket *Submitted by 0x1f8b*
- [N-17] Lack of event indexing *Submitted by 0x1f8b*

- [N-18] Lack of input verification *Submitted by 0x1f8b*
- [N-19] Remove override keyword *Submitted by 0x1f8b*
- [N-20] Lack of message in require statments *Submitted by BouSalman*
- [N-21] Lack of revert reason strings *Submitted by TomFrenchBlockchain*
- [N-22] `Factory.sol` Lack of two-step procedure and/or input validation routines for critical operations leaves them error-prone *Submitted by WatchPug*
- [N-23] Critical operations should emit events *Submitted by WatchPug*
- [N-24] Outdated compiler version *Submitted by WatchPug*
- [N-25] Missing error messages in require statements *Submitted by WatchPug*
- [N-26] Wrong syntax in test leads to wrong test results *Submitted by kenzo*
- [N-27] Unnecessary variable initialization and TODO in code *Submitted by neslinesli93*
- [N-28] Possible division by zero in `settleAuction` *Submitted by neslinesli93*
- [N-29] Open TODOs *Submitted by yeOlde*

Gas Optimizations (40)

- [G-01] Check for tokenAmount > 0 is missing in pushUnderlying function [basket.sol] *Submitted by rishabh*
- [G-02] For uint `> 0` can be replaced with `!= 0` for gas optimization *Submitted by 0x0x0x, also found by Jujic, pmerkleplant, and yeOlde*
- [G-03] Loops can be implemented more efficiently *Submitted by 0x0x0x, also found by Jujic, Meta0xNull, pmerkleplant, rishabh, and WatchPug*
- [G-04] Use negate(!) rather than `== false` *Submitted by 0x0x0x, also found by yeOlde*
- [G-05] `++i` is more efficient than `i++` *Submitted by WatchPug, also found by Jujic and pedroais*
- [G-06] `mintTo` has not an extra require statement *Submitted by 0x0x0x, also found by danb and TomFrenchBlockchain*
- [G-07] `Basket.sol#handleFees()` Check if `timeDiff > 0` can save gas *Submitted by WatchPug, also found by 0x0x0x*

- [G-08] Division with `BASE` twice can be optimized *Submitted by 0x0x0x*
- [G-09] Auction:settleAuction() cache address(basket) *Submitted by GiveMeTestEther*
- [G-10] Auction:bondForRebalance() store calculation of bondAmount in local variable *Submitted by GiveMeTestEther*
- [G-11] Auction:bondBurn(): cache bondAmount *Submitted by GiveMeTestEther*
- [G-12] Cache external call results can save gas *Submitted by WatchPug, also found by TomFrenchBlockchain*
- [G-13] Factory:constructor don't need to zero initialize storage variable *Submitted by GiveMeTestEther, also found by gzeon, Jujic, kenzo, sirhashalot, TomFrenchBlockchain, WatchPug, and yeOlde*
- [G-14] Basket:initialize() reuse function argument instead of storage variable *Submitted by GiveMeTestEther*
- [G-15] Basket:handleFees() use unchecked to save gas *Submitted by GiveMeTestEther*
- [G-16] Function handleFees #L148-L151 and updateIBRatio (Basket.sol) can be refactored for efficiency and clarity *Submitted by yeOlde, also found by GiveMeTestEther, kenzo, and WatchPug*
- [G-17] Basket:pushUnderlying()/pullUnderlying() cache ibRatio to save gas *Submitted by GiveMeTestEther, also found by WatchPug*
- [G-18] `validateWeights()` Limit loop to a meaningful bound can save gas *Submitted by WatchPug, also found by danb, GiveMeTestEther, kenzo, and TomFrenchBlockchain*
- [G-19] Useless imports *Submitted by Jujic*
- [G-20] Avoid Initialization of Loop Index If It Is 0 to Save Gas *Submitted by Meta0xNull*
- [G-21] Extra ERC20 approvals/transfers on Basket deployment *Submitted by TomFrenchBlockchain*
- [G-22] Auction.auctionOngoing variable is unnecessary *Submitted by TomFrenchBlockchain*
- [G-23] Auction.hasBonded variable is unnecessary *Submitted by TomFrenchBlockchain*

- [G-24] Excessive checking of basket totalsupply *Submitted by TomFrenchBlockchain, also found by kenzo*
- [G-25] Minted and Burned events are unnecessary *Submitted by TomFrenchBlockchain*
- [G-26] Publisher switch logic can be simplified *Submitted by TomFrenchBlockchain, also found by WatchPug*
- [G-27] Gas Optimization: Reorder storage layout *Submitted by gzeon, also found by saian*
- [G-28] `auctionImpl` and `basketImpl` in factory can be made immutable for gas savings *Submitted by TomFrenchBlockchain, also found by bw, gzeon, and robee*
- [G-29] `Basket.sol#approveUnderlying()` Cache and read storage variables from the stack can save gas *Submitted by WatchPug*
- [G-30] `Auction.sol#auctionOngoing` Switching between 1, 2 instead of true, false is more gas efficient *Submitted by WatchPug*
- [G-31] Use free functions to replace external calls can save gas *Submitted by WatchPug*
- [G-32] Unnecessary checked arithmetic in for loops *Submitted by WatchPug*
- [G-33] Adding unchecked directive can save gas *Submitted by WatchPug*
- [G-34] `Auction.sol#initialize()` Use msg.sender rather than factory_ parameter can save gas *Submitted by WatchPug*
- [G-35] `Basket.sol#initialize()` Remove redundant assertion can save gas *Submitted by WatchPug*
- [G-36] Gas: Redundant check in `setNewMaxSupply` *Submitted by cmichel*
- [G-37] Unused imports *Submitted by WatchPug, also found by robee*
- [G-38] Gas Optimization: Use calldata instead of memory *Submitted by gzeon*
- [G-39] Function changePublisher, changeLicenseFee, and setNewMaxSupply can be refactored for efficiency and clarity *Submitted by yeOlde, also found by neslinesli93*
- [G-40] Storage double reading. Could save SLOAD *Submitted by robee*

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.