# Kuiper contest
# Findings & Analysis Report

2022-01-26

## Table of contents

## Overview

### About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 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 Kuiper contest smart contract system written in Solidity. The code contest took place between September 16—September 22 2021.

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

### Wardens

30 Wardens contributed reports to the Kuiper contest:

1. WatchPug (jtp and ming)
2. cmichel
3. kenzo
4. 0xRajeev
5. itsmeSTYJ
6. goatbug
7. jonah1005
8. leastwood
9. gpersoon
10. 0xsanson
11. csanuragjain

12. hrkrshnn
13. JMukesh
14. pauliax
15. hack3r-0m
16. joeysantoro
17. nikitastupin
18. defsec
19. jah
20. bw
21. ye0lde
22. aga7hokakological
23. shenwilly
24. 0xalpharush
25. johnsterlacci
26. loop
27. t11s
28. chasemartin01
29. Alex the Entreprenerd

This contest was judged by Alex the Entreprenerd. The judge also competed in the contest as a warden, but forfeited their winnings.

Final report assembled by itsmetechjay and CloudEllie.

## Summary

The C4 analysis yielded an aggregated total of 71 unique vulnerabilities and 156 total findings. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 3 received a risk rating in the category of HIGH severity, 23 received a risk rating in the category of MEDIUM severity, and 45 received a risk rating in the category of LOW severity.

C4 analysis also identified 39 non-critical recommendations and 46 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 528 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 (3)

### [H-01] Re-entrancy in `settleAuction` allow stealing all funds

*Submitted by cmichel*

Note that the `Basket` contract approved the `Auction` contract with all tokens and the `settleAuction` function allows the auction bonder to transfer all funds out of the basket to themselves. The only limiting factor is the check afterwards that needs to be abided by. It checks if enough tokens are still in the basket after settlement:

```
    // this is the safety check if basket still has all the tokens a
    for (uint256 i = 0; i < pendingWeights.length; i++) {
        uint256 tokensNeeded = basketAsERC20.totalSupply() * pending
        require(IERC20(pendingTokens[i]).balanceOf(address(basket)) 
    }
```

The bonder can pass in any `inputTokens` , even malicious ones they created. This allows them to re-enter the `settleAuction` multiple times for the same auction.

Calling this function at the correct time (such that `bondTimestamp - auctionStart` makes `newRatio < basket.ibRatio()` ), the attacker can drain more funds each time, eventually draining the entire basket.

## Proof Of Concept

Assume that the current `basket.ibRatio` is `1e18` (the initial value). The basket publisher calls `basket.publishNewIndex` with some tokens and weights. For simplicity, assume that the pending `tokens` are the same as tokens as before, only the weights are different, i.e., this would just rebalance the portfolio. The function call then starts the auction.

The important step to note is that the `tokensNeeded` value in `settleAuction` determines how many tokens need to stay in the `basket` . If we can continuously lower this value, we can keep removing tokens from the `basket` until it is empty.

The `tokensNeeded` variable is computed as `basketAsERC20.totalSupply() * pendingWeights[i] * newRatio / BASE / BASE` . The only variable that changes in the computation when re-entering the function is `newRatio` (no basket tokens are burned, and the pending weights are never cleared).

Thus if we can show that `newRatio` decreases on each re-entrant call, we can move out more and more funds each time.

### newRatio decreases on each call

After some time, the attacker calls `bondForRebalance`. This determines the `bondTimestamp - auctionStart` value in `settleAuction`. The attack is possible as soon as `newRatio < basket.ibRatio()`. For example, using the standard parameters the calculation would be:

```
// a = 2 * ibRatio
uint256 a = factory.auctionMultiplier() * basket.ibRatio();
// b = (bondTimestamp - auctionStart) * 1e14
uint256 b = (bondTimestamp - auctionStart) * BASE / factory.auct
// newRatio = a - b = 2 * ibRatio - (bondTimestamp - auctionStar
uint256 newRatio = a - b;
```

With our initial assumption of `ibRatio = 1e18` and calling `bondForRebalance` after 11,000 seconds (~3 hours) we will get our result that `newRatio` is less than the initial `ibRatio`:

```
newRatio = a - b = 2 * 1e18 - (11000) * 1e14 = 2e18 - 1.1e18 = 0
```

> This seems to be a reasonable value (when the pending tokens and weights are equal in value to the previous ones) as no other bonder would want to call this earlier such when `newRatio > basket.ibRatio` as they would put in more total value in tokens as they can take out of the basket.

#### re-enter on settleAuction

The attacker creates a custom token `attackerToken` that re-enters the `Auction.settleAuction` function on `transferFrom` with parameters we will specify.

They call `settleAuction` with `inputTokens = [attackerToken]` to re-enter several times.

In the inner-most call where `newRatio = 0.9e18`, they choose the `inputTokens` / `outputTokens` parameters in a way to pass the initial `require(IERC20(pendingTokens[i]).balanceOf(address(basket)) >=`

`tokensNeeded);` check - transferring out any other tokens of `basket` with `outputTokens`.

The function will continue to run and call `basket.setNewWeights();` and `basket.updateIBRatio(newRatio);` which will set the new weights (but not clear the pending ones) and set the new `basket.ibRatio`.

Execution then jumps to the 2nd inner call after the `IERC20(inputTokens[i]=attackerToken).safeTransferFrom(...)` and has the chance to transfer out tokens again. It will compute `newRatio` with the new lowered `basket.ibRatio` of `0.9e18`: `newRatio = a - b = 2 * 0.9e18 - 1.1e18 = 0.7e18`. Therefore, `tokensNeeded` is lowered as well and the attacker was allowed to transfer out more tokens having carefully chosen `outputWeights`.

This repeats with `newRatio = 0.3`.

The attack is quite complicated and requires carefully precomputing and then setting the parameters, as well as sending back the `bondAmount` tokens to the `auction` contract which are then each time transferred back in the function body. But I believe this should work.

## Impact

The basket funds can be stolen.

## Recommended Mitigation Steps

Add re-entrancy checks (for example, OpenZeppelin's "locks") to the `settleAuction` function.

frank-beard (Kuiper) confirmed

Alex the Entreprenerd (judge) commented:

> Let's dissect the finding to prove whether it's valid or not.

> First of all, we do have the pre-conditions for re-entrancy:

- Allows any token as input (untrusted input)
- No re-entrancy modifier
- Checks are performed at the beginning, transfers in the middle, and state changes at the end (violation of `check effect interaction` pattern)

> So in any case this is a report for reEntrancy (medium severity)

> However, the warden is showing a specific attack vector that, if proven, allows to steal the majority of funds from the basket.

> Let's investigate (with a single re-entrant call example):

> The require checks at the top pass, as we're rebalancing the basket, we'll make sure to use an additional inputToken (malicious token), that will call `settleAuction` again.

> This second call (Call B), will execute as normal, extracting the correct amount of value in return for rebalancing, the new ibRatio is 0.9 as shown by the warden POC.

> Call B ends by setting ibRatio to 0.9, and `hasBonded` to false (which will cause a revert if you try to perform this without re-entrancy)

> However, we have already entered and Call A can resume, it now has ibRatio set to 0.9 which allows it to further extract value (as `tokensNeeded` decreases as ibRatio decreases)

> This can be extended to have further re-entrant calls and can be effectively executed until the basket is hollowed out

> This is a Miro Board to highlight the dynamics of the exploit: [https://miro.com/app/board/uXjVOZk4gxw=/?invite_link_id=246345621880](https://miro.com/app/board/uXjVOZk4gxw=/?invite_link_id=246345621880)

> Huge props to the warden, brilliant find!

[Alex the Entreprenerd (judge) commented]():

> For mitigation:

- Adding a re-entrancy check would be the place to start

- Requiring the list of input tokens to match the output can also be useful to avoid any other shenanigans
- Setting the time difference (`bondTimestamp`, `auctionStart`) to be 0 would also negate the ability to further manipulate the `ibRatio`

## [H-02] `Basket.sol#auctionBurn()` A failed auction will freeze part of the funds

*Submitted by WatchPug*

[https://github.com/code-423n4/2021-09-defiProtocol/blob/main/contracts/contracts/Basket.sol#L102-L108](https://github.com/code-423n4/2021-09-defiProtocol/blob/main/contracts/contracts/Basket.sol#L102-L108)

Given the `auctionBurn()` function will `_burn()` the auction bond without updating the `ibRatio`. Once the bond of a failed auction is burned, the proportional underlying tokens won't be able to be withdrawn, in other words, being frozen in the contract.

### Proof of Concept

With the configuration of:

basket.ibRatio = 1e18 factory.bondPercentDiv = 400 basket.totalSupply = 400 basket.tokens = [BTC, ETH] basket.weights = [1, 1]

1. Create an auction;
2. Bond with 1 BASKET TOKEN;
3. Wait for 24 hrs and call `auctionBurn()`;

`basket.ibRatio` remains to be 1e18; basket.totalSupply = 399.

Burn 1 BASKET TOKEN will only get back 1 BTC and 1 ETH, which means, there are 1 BTC and 1 ETH frozen in the contract.

### Recommended Mitigation Steps

Change to:

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

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

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

**frank-beard (Kuiper) confirmed**

**Alex the Entreprenerd (judge) commented**:

> The warden has identified a way for funds to be stuck without a way to recoup them, this is because `ibRatio` is not updated, while `totalSupply` is.

> Because this is a specific accounting error, which is effectively a bug in the logic of the protocol, and funds can be irrevocably lost, this is a high severity finding

## [H-03] Reentrancy in settleAuction(): malicious publisher can bypass index timelock mechanism, inject malicious index, and rug the basket

*Submitted by kenzo, also found by itsmeSTYJ and jonah1005*

The `settleAuction()` function calls `withdrawBounty()` before setting `auctionOngoing = false`, thereby allowing reentrancy.

### Impact

A malicious publisher can bypass the index timelock mechanism and publish new index which the basket's users won't have time to respond to. At worst case, this means setting weights that allow the publisher to withdraw all the basket's underlying funds for himself, under the guise of a valid new index.

### Proof of Concept

1. The publisher (a contract) will propose new valid index and bond the auction.

   To settle the auction, the publisher will execute the following steps in the same transaction:

2. Add a bounty of an ERC20 contract with a malicious `transfer()` function.
3. Settle the valid new weights correctly (using `settleAuction()` with the correct parameters, and passing the malicious bounty id).
4. `settleAuction()` will call `withdrawBounty()` which upon transfer will call the publisher's malicious ERC20 contract.
5. The contract will call `settleAuction()` again, with empty parameters. Since the previous call's effects have already set all the requirements to be met, `settleAuction()` will finish correctly and call `setNewWeights()` which will set the new valid weights and set `pendingWeights.pending = false`.
6. Still inside the malicious ERC20 contract transfer function, the attacker will now call the basket's `publishNewIndex()`, with weights that will transfer all the funds to him upon his burning of shares. This call will succeed to set new pending weights as the previous step set `pendingWeights.pending = false`.
7. Now the malicious `withdrawBounty()` has ended, and the original `settleAuction()` is resuming, but now with malicious weights in `pendingWeights` (set in step 6). `settleAuction()` will now call `setNewWeights()` which will set the basket's weights to be the malicious pending weights.
8. Now `settleAuction` has finished, and the publisher (within the same transaction) will `burn()` all his shares of the basket, thereby transferring all the tokens to himself.

POC exploit: Password to both files: "exploit". AttackPublisher.sol , to be put under contracts/contracts/Exploit: [https://pastebin.com/efHZjstS](https://pastebin.com/efHZjstS) ExploitPublisher.test.js , to be put under contracts/test: [https://pastebin.com/knBtcWkk](https://pastebin.com/knBtcWkk)

## Tools Used

Manual analysis, hardhat.

## Recommended Mitigation Steps

In `settleAuction()`, move `basketAsERC20.transfer()` and `withdrawBounty()` to the end of the function, conforming with Checks Effects Interactions pattern.

frank-beard (defiProtocol) confirmed

Alex the Entreprenerd (judge) commented:

> This is a re-entrancy finding.

> There is no denying that the code is vulnerable to re-entrancy

> The warden identified the way to exploit re-entrancy by using a malicious bounty token.

> I think the finding is valid and the warden has shown how to run re-entrnacy.

> That said the POC the warden shows requires calling `publishNewIndex` which is a `onlyPublisher` function. This exploit would be contingent on the publisher rugging the basket.

> The code is:

- Vulnerable to re-entancy
- The warden showed how to trigger it

> Despite the fact that the POC is flawed, I believe this finding highlights a different vector for re-entrancy (bounty token transfers) as such I agree with a high severity

## Medium Risk Findings (23)

### [M-01] Use safeTransfer instead of transfer

*Submitted by hack3r-0m, also found by itsmeSTYJ, JMukesh, leastwood, and shenwilly*

https://github.com/code-423n4/2021-09-defiProtocol/blob/main/contracts/contracts/Auction.sol#L146 `transfer()` might return false instead of reverting, in this case, ignoring return value leads to considering it successful.

use `safeTransfer()` or check the return value if length of returned data is > 0.

**frank-beard (Kuiper) confirmed**

**Alex the Entreprenerd (judge) commented:**

> Agree with finding, agree with severity given the specific example given as the funds would be stuck in the contract

## [M-02] Fee on transfer tokens can lead to incorrect approval

*Submitted by hrkrshnn, also found by itsmeSTYJ*

### Fee on transfer tokens can lead to incorrect approval

The createBasket function does not account for tokens with fee on transfer.

```
    function createBasket(uint256 idNumber) external override return
        // ...
        for (uint256 i = 0; i < bProposal.weights.length; i++) {
            IERC20 token = IERC20(bProposal.tokens[i]);
            token.safeTransferFrom(msg.sender, address(this), bPropo
            token.safeApprove(address(newBasket), bProposal.weights[
        }
        // ...
    }
```

The function `safeTransferFrom` may not transfer exactly `bProposal.weights[i]` amount of tokens, for tokens with a fee on transfer. This means that the `safeApprove` call in the next line would be approving more tokens than what was received, leading to accounting issues.

### Recommended Mitigation Steps

It is recommended to find the balance of the current contract before and after the `transferFrom` to see how much tokens were received, and approve only what was received.

[frank-beard (Kuiper) confirmed]:

> the protocol for now is only expected to work with defi safe, standard erc-20 tokens.

[Alex the Entreprenerd (judge) commented]:

> This finding is similar to #206 , but in contrast to it, it shows a specific way to brick / grief the protocol, as per the docs:

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

> This is an hypothetical attack with stated assumptions

> Will not mark as duplicate and will consider this as a valid finding as it shows a specific vulnerability when paired with `feeOnTransfer` tokens

> Mitigation can be as simple as never using `feeOnTransfer` tokens

## [M-03] `onlyOwner` Role Can Unintentionally Influence `settleAuction()`

*Submitted by leastwood*

### Impact

The `onlyOwner` role is able to make changes to the protocol with an immediate affect, while other changes made in `Basket.sol` and `Auction.sol` incur a one day timelock. As a result, an `onlyOwner` role may unintentionally frontrun a `settleAuction()` transaction by making changes to `auctionDecrement` and `auctionMultiplier` , potentially causing the auction bonder to over compensate during a rebalance. Additionally, there is no way for an auction bonder to recover their tokens in the event this does happen.

### Proof of Concept

- https://github.com/code-423n4/2021-09-defiProtocol/blob/main/contracts/contracts/Factory.sol#L39-L59
- https://github.com/code-423n4/2021-09-defiProtocol/blob/main/contracts/contracts/Auction.sol#L89-L99

### Tools Used

Manual code review

### Recommended Mitigation Steps

Consider adding a timelock delay to all functions affecting protocol execution. Alternatively, `bondForRebalance()` can set state variables for any external calls made to `Factory.sol` (i.e. `factory.auctionMultiplier()` and `factory.auctionDecrement()`), ensuring that `settleAuction()` is called according to these expected results.

frank-beard (Kuiper) acknowledged:

> it is assumed the owner is trustworthy in this version of the protocol, however we will add mitigations and further decentralization in future updates

Alex the Entreprenerd (judge) commented:

> Agree with the finding, users are taking “owner privileges” risks while interacting with the protocol. The warden has identified a specific grief / DOS that the owner can cause

## [M-04] User can mint miniscule amount of shares, later withdraw miniscule more than deposited

*Submitted by kenzo*

If a user is minting small amount of shares (like 1 - amount depends on baskets weights), the calculated amount of tokens to pull from the user can be less than 1, and therefore no tokens will be pulled. However the shares would still be minted. If the user does this a few times, he could then withdraw the total minted shares and end up with more tokens than he started with - although a miniscule amount.

## Impact

User can end up with more tokens than he started with. However, I didn't find a way for the user to get an amount to make this a feasible attack. He gets dust. However he can still get more than he deserves. If for some reason the basket weights grow in a substantial amount, this could give the user more tokens that he didn't pay for.

## Proof of Concept

Add the following test to `Basket.test.js`. The user starts with 5e18 UNI, 1e18 COMP, 1e18 AAVE, and ends with 5e18+4, 1e18+4, 1e18+4.

```
    it("should give to user more than he deserves", async () => {
        await UNI.connect(owner).mint(ethers.BigNumber.from(UNI_WEIG
        await COMP.connect(owner).mint(ethers.BigNumber.from(COMP_WE
        await AAVE.connect(owner).mint(ethers.BigNumber.from(AAVE_WE

        await UNI.connect(owner).approve(basket.address, ethers.BigN
        await COMP.connect(owner).approve(basket.address, ethers.Big
        await AAVE.connect(owner).approve(basket.address, ethers.Big

        console.log("User balance before minting:");
        console.log("UNI balance: " + (await UNI.balanceOf(owner.add
        console.log("COMP balance: " + (await COMP.balanceOf(owner.a
        console.log("AAVE balance: " + (await AAVE.balanceOf(owner.a


        await basket.connect(owner).mint(ethers.BigNumber.from(1).di
        await basket.connect(owner).mint(ethers.BigNumber.from(1).di
        await basket.connect(owner).mint(ethers.BigNumber.from(1).di
        await basket.connect(owner).mint(ethers.BigNumber.from(1).di
        await basket.connect(owner).mint(ethers.BigNumber.from(1).di

        console.log("\nUser balance after minting 1 share 5 times:")
        console.log("UNI balance: " + (await UNI.balanceOf(owner.add
        console.log("COMP balance: " + (await COMP.balanceOf(owner.a
        console.log("AAVE balance: " + (await AAVE.balanceOf(owner.a

        await basket.connect(owner).burn(await basket.balanceOf(owne
        console.log("\nUser balance after burning all shares:");
        console.log("UNI balance: " + (await UNI.balanceOf(owner.add
        console.log("COMP balance: " + (await COMP.balanceOf(owner.a
        console.log("AAVE balance: " + (await AAVE.balanceOf(owner.a
    });
```

## Tools Used

Manual analysis, hardhat.

## Recommended Mitigation Steps

Add a check to `pullUnderlying`:

```
    require(tokenAmount > 0);
```

I think it makes sense that if a user is trying to mint an amount so small that no tokens could be pulled from him, the mint request should be denied. Per my tests, for an initial ibRatio, this number (the minimal amount of shares that can be minted) is 2 for weights in magnitude of 1e18, and if the weights are eg. smaller by 100, this number will be 101.

**frank-beard (Kuiper) confirmed**

**Alex the Entreprenerd (judge) commented**:

> Great find, because this finding shows a clear POC of how to extract value from the system, I agree with medium severity

## [M-05] Bonding mechanism allows malicious user to DOS auctions

*Submitted by kenzo, also found by goatbug*

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.

## 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 and marked as duplicate:

> https://github.com/code-423n4/2021-09-defiprotocol-findings/issues/276

Alex the Entreprenerd (judge) commented:

> I think the fundamental issue here is that there can only be one bonder that rebalances, an elegant solution would be to move to a mapping of bonders that rebalance.

> Allowing anyone to bond and rebalance while still allowing the same dynamic of burning their bonds

> This would be more in line with the idea of offering bonds for discounts (Olympus Pro) to anyone as long as there's enough underlying

> I agree with the finding and believe the severity is appropriate as this effectively slows down the rebalancing by at least 2 days

## [M-06] 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](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:

```
    it("should divide by 0", async () => {
    await basket.connect(addr1).burn(await basket.balanceOf(addr1.ad
    await basket.connect(addr2).burn(await basket.balanceOf(addr2.ad

        await UNI.connect(addr1).approve(basket.address, ethers.BigN
        await COMP.connect(addr1).approve(basket.address, ethers.Big
        await AAVE.connect(addr1).approve(basket.address, ethers.Big
        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**

**Alex the Entreprenerd (judge) commented:**

> I feel like this can also happen right after the contract was created `mintTo` will call `handleFees` which will revert due to division by 0

> Finding is valid

## [M-07] No minimum rate in the auction may break the protocol under network failure

*Submitted by jonah1005*

### Impact

The aution contract decides a new `ibRatio` in the function `settleAuction`.
[Auction.sol#L89-L91](Auction.sol#L89-L91)

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

There's a chance that `newRatio` would be really close to zero. This imposes too much risk on the protocol. The network may not really be healthy all the time. Solana and Arbitrum were down and Ethereum was suffered a forking issue recently. Also, the network may be jammed from time to time. This could cause huge damage to a protocol. Please refer to [Black Thursday for makerdao 8.32 million was liquidated for 0 dai](Black Thursday for makerdao 8.32 million was liquidated for 0 dai)

Given the chance that all user may lose their money, I consider this is a medium-risk issue.

### Proof of Concept

[Black Thursfay for makerdao 8.32 million was liquidated for 0 dai](Black Thursfay for makerdao 8.32 million was liquidated for 0 dai) [bug-impacting-over-50-of-ethereum-clients-leads-to-fork](bug-impacting-over-50-of-ethereum-clients-leads-to-fork)

### Tools Used

None

## Recommended Mitigation Steps

I recommend setting a minimum `ibRatio` when a publisher publishes a new index. The auction should be killed if the `ibRatio` is too low.

[frank-beard (Kuiper) confirmed]

[Alex the Entreprenerd (judge) commented]:

> Agree that `newRatio` decreases over time, if nothing is done it will eventually get close to 0, at which point `tokensNeeded` will tend to 0, which would mean that the amount of underlying necessary to redeem the bond decreases over time

> This can hypothetically allow to redeem the bond for extremely cheap if not for free

> The bond is denominated in the basketToken which effectively represents the value of the “mixture” of the tokens

> Over time, you’re getting the same bond (basket) for less tokens (tokensNeeded), which also means that the basket itself is loosing value (because you can extra the excess value via bonding)

> Sounds to me like the entire `settleAuction` mechanism is devaluing the basket over time which arguably is a high severity vulnerability.

> Will continue the judging and may end up getting in touch with the sponsor over some additional risks.

> The finding is valid because it shows that the protocol can break given a specific circumstance, I do believe the warden could have found a higher severity by digging deeper

## [M-08] settleAuction may be impossible if locked at a wrong time.

*Submitted by jonah1005*

### Impact

The auction contract decides a new `ibRatio` in the function `settleAuction`. Auction.sol#L89-L91

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

In this equation, `a` would not always be greater than `b`. The `auctionBonder` may lock the token in `bondForRebalance()` at a point that `a-b` would always revert.

The contract should not allow users to lock the token at the point that not gonna succeed. Given the possible (huge) loss of the user may suffer, I consider this is a medium-risk issue.

## Proof of Concept

Here's a web3.py script to trigger this bug.

```
basket.functions.publishNewIndex([dai.address], [deposit_amount]

for i in range(4 * 60 * 24):
    w3.provider.make_request('evm_mine', [])
basket.functions.publishNewIndex([dai.address], [deposit_amount]

print('auction on going', auction.functions.auctionOngoing().cal
for i in range(20000):
    w3.provider.make_request('evm_mine', [])

all_token = basket.functions.balanceOf(user).call()
basket.functions.approve(auction.address, all_token).transact()
auction.functions.bondForRebalance().transact()
# error Log
# {'code': -32603, 'message': 'Error: VM Exception while process
auction.functions.settleAuction([], [], [], [], []).transact()
```

## Tools Used

None

### Recommended Mitigation Steps

Recommend to calculate the new irate in `bondForRebalance`. I understand the `auctionBonder` should take the risk to get the profit. However, the contract should protect the user in the first place when this auction is doomed to fail.

[frank-beard (Kuiper) confirmed]

[Alex the Entreprenerd (judge) commented]:

> I do not believe the users are at risk of a specific loss as they can always redeem their shares for underlying `Basket.burn` However, this is still a valid finding, and the external conditions make medium severity appropriate

## [M-09] Fee calculation is potentially incorrect

*Submitted by itsmeSTYJ*

### Impact

More fees are actually charged than intended

### Mitigation Steps

[Basket.sol line 118]

Assume that license fee is 10% i.e. 1e17 and time diff = half a year.

When you calculate `feePct`, you expect to get 5e16 since that's 5% and the actual amount of fee to be charged should be totalSupply * feePct (5) / BASE (100) but on line 118, we are actually dividing by BASE - feePct i.e. 95.

5 / 95 = 0.052 instead of the intended 0.05.

Solution is to replace `BASE - feePct` in the denominator with `BASE`.

[frank-beard (Kuiper) confirmed]

[Alex the Entreprenerd (judge) commented]:

> The warden identified an inconsistency with the math that charged more fees than intended

## [M-10] `burn` and `mintTo` in `Basket.sol` vulnerable to reentrancy

*Submitted by 0xalpharush, also found by hack3r-0m, JMukesh, and johnsterlacci*

### Impact

The functions mintTo and burn make external calls prior to updating the state. If a basket contains an ERC777 token, attackers can mint free basket tokens.

### Proof of Concept

An attacker could reenter the `mintTo` function when the contract pulls an ERC777 token from the user and mint more tokens than they deposited.

### Tools Used

Slither

### Recommended Mitigation Steps

Move external calls after state updates. It is best practice to make external calls after updating state in accordance with the check-effect-interact pattern.

frank-beard (Kuiper) acknowledged:

> For now we are only concerned with 'Defi Safe' tokens that conform to the erc-20 standard. It is expected that publishers and users should do due diligence when adding assets to a basket

Alex the Entreprenerd (judge) commented:

> I agree with the warden finding that the function can be subject to re-entrancy.

> Because the exploit is dependent on the token of choice, this is not a guarantee but rather the possibility of the system being vulnerable.

> I highly recommend the sponsor to add re-entrancy checks.

> As for the finding, because it is conditional on using a vulnerable token, I'll downgrade to medium severity as it requires an external condition for the re-entrance to be possible

[Alex the Entreprenerd (judge) commented](#):

> After re-review I agree with medium severity on reentrancy findings on burn and mint, in contract with the "generic" re-entrancy finding with the additional bounties that lack any poc.

> The findings with specific poc have been rated as separate findings (some of which have high risk) as they show an actual way to exploit the protocol

## [M-11] Owner can steal all Basket funds during auction

*Submitted by Oxsanson*

### Impact

The owner of Factory contract can modify the values of `auctionMultiplier` and `auctionDecrement` at any time. During an auction, these values are used to calculate `newRatio` and thereby `tokensNeeded`: specifically, it's easy to set the factory parameters so that `tokensNeeded = 0` (or close to zero) for every token. This way the owner can participate at an auction, change the parameters, and get the underlying tokens from a Basket without transferring any pending tokens.

### Proof of Concept

[https://github.com/code-423n4/2021-09-defiProtocol/blob/main/contracts/contracts/Auction.sol#L89-L99](https://github.com/code-423n4/2021-09-defiProtocol/blob/main/contracts/contracts/Auction.sol#L89-L99)

### Tools Used

editor

### Recommended Mitigation Steps

Consider adding a Timelock to these Factory functions. Otherwise a way to not modify them if an auction is ongoing (maybe Auction saves the values it reads when `startAuction` is called).

[frank-beard (Kuiper) confirmed and disagreed with severity]():

> the owner for this is intended to be a dao that acts in support of the protocol, however this is a good point to the centralization concerns for the protocol, we will most likely manage this by adding a timelock to these function

[Alex the Entreprenerd (judge) commented]():

> Agree with the finding from the warden, highly recommend the sponsor to add these possibilities in their docs to make it easy for users to understand them.

> That said, a possible remediation would also be to add checks in the setters, to avoid for these specific edge cases.

> Because the exploit is dependent on an external condition (setting to a specific value by governance), the finding is of medium severity

## [M-12] Factory.sol - lack of checks in `setAuctionDecrement` will cause reverts in Auction::settleAuction()

*Submitted by Alex the Entreprenerd, also found by goatbug*

### Impact

`setAuctionDecrement` doesn't check for a min nor a max amount This means we can change `auctionDecrement` which would allow `owner` to set `auctionDecrement` to 0

This will cause the function `settleAuction` in Auction.sol to revert

This allows the owner to block auctions from being settled

### Proof of Concept

`setAuctionDecrement(0)` Now `settleAuction` will revert due to division by 0

### Recommended Mitigation Steps

Add checks in `setAuctionDecrement` Refactor to

```
    function setAuctionDecrement(uint256 newAuctionDecrement) public
        require(newAuctionDecrement > AMOUNT);
        require(newAuctionDecrement <= AMOUNT\_2);
        auctionDecrement = newAuctionDecrement;
    }
```

**frank-beard (Kuiper) acknowledged and disagreed with severity:**

> the owner for this is intended to be a dao that acts in support of the protocol, however this is a good point to the centralization concerns for the protocol, we will most likely manage this by adding a timelock to these function

**Alex the Entreprenerd (judge) commented:**

> I wrote the finding and in being the contest judge will be forfeiting all winnings.

> After thinking about it, this is a medium severity finding because it is a DOS that relies on a specific condition. While the argument that the owner can be being always hold up, it is important that protocol users understand the risk, so it's important to flag up admin privileges

## [M-13] lack of checks in `Factory::setBondPercentDiv` allow owner to prevent bonding in Auction::bondForRebalance()

*Submitted by Alex the Entreprenerd, also found by goatbug*

### Impact

`setBondPercentDiv` has no checks for min and max

Setting `bondPercentDiv` to 0 will cause `Auction::bondForRebalance()` to revert

This allows the owner to prevent bonding by setting the `bondPercentDiv` to 0

### Recommended Mitigation Steps

Refactor to

```
    function setBondPercentDiv(uint256 newBondPercentDiv) public ove
        require(newBondPercentDiv > AMOUNT);
        require(newBondPercentDiv <= AMOUNT_2);
        bondPercentDiv = newBondPercentDiv;
    }
```

**frank-beard (Kuiper) acknowledged and disagreed with severity:**

> the owner for this is intended to be a dao that acts in support of the protocol, however this is a good point to the centralization concerns for the protocol, we will most likely manage this by adding a timelock to these function

**Alex the Entreprenerd (judge) commented:**

> Similarly to #119

- Am forfeiting my wins
- Always important to flag up admin privileges so people can use the protocol while being aware of the risks they are undertaking
- Medium severity as depends on a specific config

## [M-14] `Basket.sol#handleFees()` could potentially cause disruption of minting and burning

*Submitted by WatchPug*

https://github.com/code-423n4/2021-09-defiProtocol/blob/main/contracts/contracts/Basket.sol#L110-L129

```
    function handleFees() private {
        if (lastFee == 0) {
            lastFee = block.timestamp;
        } else {
            uint256 startSupply = totalSupply();

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

            _mint(publisher, fee * (BASE - factory.ownerSplit()) / B
            _mint(Ownable(address(factory)).owner(), fee * factory.o
            lastFee = block.timestamp;

            uint256 newIbRatio = ibRatio * startSupply / totalSupply
            ibRatio = newIbRatio;

            emit NewIBRatio(ibRatio);
        }
    }
```

`timeDiff * licenseFee` can be greater than `ONE_YEAR` when `timeDiff` and/or `licenseFee` is large enough, which makes `feePct` to be greater than `BASE` so that `BASE - feePct` will revert on underflow.

## Impact

Minting and burning of the basket token are being disrupted until the publisher update the `licenseFee`.

## Proof of Concept

1. Create a basket with a `licenseFee` of `1e19` or 1000% per year and mint 1 basket token;
2. The basket remain inactive (not being minted or burned) for 2 months;
3. Calling `mint` and `burn` reverts at `handleFees()`.

## Recommended Mitigation Steps

Limit the max value of `feePct`.

frank-beard (Kuiper) confirmed

Alex the Entreprenerd (judge) commented:

> The finding is valid, there are conditions that would cause `feePct` to be greater than `BASE`

> The conditions to trigger this seem to be:

- Wait enough time
- Have a high enough fee

> Because this can happen under specific conditions, I will grade this finding as medium severity:

> I would highly recommend the sponsor to consider the possibility of capping the `licenseFee` to make it easier to predict cases in which the operation can revert

## [M-15] `Auction.sol#settleAuction()` late auction bond could potentially not being able to be settled, cause funds loss to bonder

*Submitted by WatchPug*

The `newRatio` that determines `tokensNeeded` to settle the auction is calculated based on `auctionMultiplier`, `bondTimestamp - auctionStart` and `auctionDecrement`.

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

However, if an auction is bonded late (`bondTimestamp - auctionStart` is a large number), and/or the `auctionMultiplier` is small enough, and/or the `auctionDecrement` is small enough, that makes `b` to be greater than `a`, so that `uint256 newRatio = a - b;` will revert on underflow.

This might seem to be an edge case issue, but considering that a rebalance auction of a bag of shitcoin to high-value tokens might just end up being bonded at the last minute, with a `newRatio` near zero. When we take the time between the bonder submits the transaction and it got packed into a block, it's quite possible that the final `bondTimestamp` gets large enough to revet `a - b`.

### Impact

An auction successfully bonded by a regular user won't be able to be settled, and the user will lose the bond.

### Proof of Concept

With the configuration of:

basket.ibRatio = 1e18 factory.auctionDecrement = 5760 (Blocks per day) factory.auctionMultiplier = 2

1. Create an auction;
2. The auction remain inactive (not get bonded) for more than 2 days (>11,520 blocks);
3. Call `bondForRebalance()` and it will succeed;
4. Calling `settleAuction()` will always revert.

### Recommended Mitigation Steps

Calculate and require `newRatio > 0` in `bondForRebalance()`, or limit the max value of decrement and make sure newRatio always > 0 in `settleAuction()`.

frank-beard (Kuiper) confirmed

Alex the Entreprenerd (judge) commented:

> Finding is valid, there are cases that can cause a - b to revert

> The finding is reliant on external condition (multiplier being low and / or bonding late) as such I believe this to be a medium severity finding.

> Note for the sponsor: The cause for reverts should get clearer with time (as people use the protocol), you definitely want to model these revert behaviour to avoid edge cases that will make funds stuck

## [M-16] `Auction.sol#settleAuction()` Mishandling bounty state could potentially disrupt `settleAuction()`

*Submitted by WatchPug*

[https://github.com/code-423n4/2021-09-defiProtocol/blob/main/contracts/contracts/Auction.sol#L143](https://github.com/code-423n4/2021-09-defiProtocol/blob/main/contracts/contracts/Auction.sol#L143)

```
    function withdrawBounty(uint256[] memory bountyIds) internal {
        // withdraw bounties
        for (uint256 i = 0; i < bountyIds.length; i++) {
            Bounty memory bounty = _bounties[bountyIds[i]];
            require(bounty.active);

            IERC20(bounty.token).transfer(msg.sender, bounty.amount)
            bounty.active = false;

            emit BountyClaimed(msg.sender, bounty.token, bounty.amou
        }
    }
```

In the `withdrawBounty` function, `bounty.active` should be set to `false` when the bounty is claimed.

However, since `bounty` is stored in memory, the state update will not succeed.

## Impact

An auction successfully bonded by a regular user won’t be able to be settled if they passed seemly active bountyIds, and the bonder will lose the bond.

## Proof of Concept

1. Create an auction;
2. Add a bounty;
3. Auction settled with bounty claimed;
4. Create a new auction;
5. Add a new bounty;
6. Calling `settleAuction()` with the bountyIds of the 2 seemly active bounties always reverts.

### Recommended Mitigation Steps

Change to:

```
    Bounty storage bounty = _bounties[bountyIds[i]];
```

[frank-beard (Kuiper) confirmed and marked as duplicate]:**

> duplicate of https://github.com/code-423n4/2021-09-defiprotocol-findings/issues/168

[Alex the Entreprenerd (judge) commented]:

> Finding is valid, because the warden didn't provide a POC of how to steal user funds, the finding is of medium severity

## [M-17] Unsafe approve would halt the auction and burn the bond

*Submitted by jonah1005, also found by 0xRajeev, 0xsanson, cmichel, and itsmeSTYJ*

### Impact

[Basket.sol#L224-L228] calls approve that do not handle non-standard erc20 behavior.

1. Some token contracts do not return any value.
2. Some token contracts revert the transaction when the allowance is not zero.

Since the auction contract calls `setNewWeights` in function `settleAuction`, `auctionBonder` may lock its bond and never successfully settles the auction. This leads to the `auctionBonder` loss he's bond and the basket and the auction becomes suspended.

`auctionBonder` would lose his bond and the contract would be suspended. I consider this a high-risk issue.

### Proof of Concept

USDT may be a classic non-standard erc20 token.

Here’s a short POC.

```
usdt.functions.approve(basket.address, 100).transact()
## the second tx would be reverted as the allowance is not zero
usdt.functions.approve(basket.address, 50).transact()
```

### Tools Used

None

### Recommended Mitigation Steps

Recommend to use `safeApprove` instead and set the allowance to 0 before calling it.

```
function approveUnderlying(address spender) private {
    for (uint256 i = 0; i < weights.length; i++) {
        IERC20(tokens[i]).safeApprove(spender, 0);
        IERC20(tokens[i]).safeApprove(spender, type(uint256).max
    }
}
```

**[frank-beard (Kuiper) marked as duplicate](https://github.com/code-423n4/2021-09-defiprotocol-findings/issues/260):**

> duplicate of [https://github.com/code-423n4/2021-09-defiprotocol-findings/issues/260](https://github.com/code-423n4/2021-09-defiprotocol-findings/issues/260)

**Alex the Entreprenerd (judge) commented:**

> Agree with the finding, because this is contingent on an external condition (token being USDT) the finding is of medium severity

## [M-18] licenseFee can be greater than BASE

*Submitted by itsmeSTYJ*

### Impact

Worst case - no functions that contains `handleFees()` can pass because line 118 will always underflow and revert. You only need `feePct` to be bigger than `BASE` for the `handleFees()` function to fail which will result in a lot of gas wasted and potentially bond burnt.

I did not classify this as high risk because a simple fix would be to simply reduce the licenseFee via `changeLicenseFee`.

### Recommended Mitigation Steps

Add these require statement to the following functions:

- Basket.changeLicenseFee()
  - `require(newLicenseFee <= BASE, "changeLicenseFee: license fee cannot be greater than 100%");`
- Factory.proposeBasketLicense()
  - `require(licenseFee <= BASE, "proposeBasketLicense: license fee cannot be greater than 100%");`

**frank-beard (Kuiper) acknowledged**

**Alex the Entreprenerd (judge) commented**:

> Agree with the finding, the warden highlighted an admin exploit that allows to DOS the basket. Adding a check not only prevents the exploit, but gives a security guarantee to the protocol users

## [M-19] Scoop ERC20 tokens from basket contract

*Submitted by gpersoon*

### Impact

Suppose some unrelated ERC20 tokens end up in the basket contract (via an airdrop, a user mistake etc)

Then anyone can do a `bondForRebalance()` and `settleAuction()` to scoop these tokens.

The function `settleAuction()` allows you to specify an outputToken, so also completely unrelated tokens. Thus you can retrieve additional tokens with `settleAuction()`

### Proof of Concept

[https://github.com/code-423n4/2021-09-defiProtocol/blob/main/contracts/contracts/Auction.sol#L69](https://github.com/code-423n4/2021-09-defiProtocol/blob/main/contracts/contracts/Auction.sol#L69) function settleAuction(.. address[] memory outputTokens, uint256[] memory outputWeights) public override { … for (uint256 i = 0; i < outputTokens.length; i++) { IERC20(outputTokens[i]).safeTransferFrom(address(basket), msg.sender, outputWeights[i]); }

### Recommended Mitigation Steps

Check outputTokens are part of the previous basket tokens (e.g. `basket.tokens()` )

frank-beard (Kuiper) acknowledged

Alex the Entreprenerd (judge) commented:

> If the Auction contract has any ERC20 that is not checked against the require, those tokens can be taken away for free. The warden didn't directly mention this, but this can also apply to bounties. Any bounty specified in a token that is not protected can be taken without claiming the bounty (as the require won't check for it)

> I think that the finding has appropriate severity, although the warden didn't directly mention the ability to steal bounties

## [M-20] Auction multiplier set to zero

*Submitted by goatbug*

### Impact

```
    function setAuctionMultiplier(uint256 newAuctionMultiplier) publ
        auctionMultiplier = newAuctionMultiplier;
    }
```

auction multiplier can be set to zero by factory owner. This would stop the auction settling, function would always revert.

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

causing a safe math error and `newRatio` to revert.

## Proof of Concept

Provide direct links to all referenced code in GitHub. Add screenshots, logs, or any other relevant proof that illustrates the concept.

## Recommended Mitigation Steps

**[frank-beard (Kuiper) acknowledged and disagreed with severity](#):**

> it is assumed the owner is trustworthy in this version of the protocol, however we will add mitigations and further decentralization in future updates

**[Alex the Entreprenerd (judge) commented](#):**

> Agree with the finding, because the warden showed a specific "admin privilege" that DOSses the protocol, the finding is valid and of medium severity

## [M-21] Zero weighted baskets are allowed to steal funds

*Submitted by csanuragjain*

### Impact

It was observed that Publisher is allowed to create a basket with zero token and weight. This can lead to user fund stealing as described in below poc The issue was discovered in `validateWeights` function of Basket contract

## Proof of Concept

1. User proposes a new Basket with 0 tokens and weights using `proposeBasketLicense` function in Factory contract

```
Proposal memory proposal = Proposal({
        licenseFee: 10,
        tokenName: abc,
        tokenSymbol: aa,
        proposer: 0xabc,
        tokens: {},
        weights: {},
        basket: address(0)
});
```

2. `validateWeights` function is called and it returns success as the only check performed is `\_tokens.length == \_weights.length (0=0)`

```
function validateWeights(address[] memory _tokens, uint256[] mem
    require(_tokens.length == _weights.length);
    uint256 length = _tokens.length;
    address[] memory tokenList = new address[](length);

    // check uniqueness of tokens and not token(0)

    for (uint i = 0; i < length; i++) {
        ...
    }
}
```

3. A new proposal gets created

```
    _proposals.push(proposal);
```

4. User creates new Basket with this proposal using `createBasket` function

```
function createBasket(uint256 idNumber) external override return
    Proposal memory bProposal = _proposals[idNumber];
    require(bProposal.basket == address(0));

    ....

    for (uint256 i = 0; i < bProposal.weights.length; i++) {
        ...
    }
    ...
    return newBasket;
}
```

5. Since no weights and tokens were in this proposal so no token transfer is required (`bProposal.weights.length` will be 0 so loop won't run)
6. Basket gets created and user becomes publisher for this basket

```
newBasket.mintTo(BASE, msg.sender);
_proposals[idNumber].basket = address(newBasket);
```

7. Publisher owned address calls the mint function with say amount 10 on `Basket.sol` contract

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

function mintTo(uint256 amount, address to) public override {
    ...

    pullUnderlying(amount, msg.sender);

    _mint(to, amount);

    ...
}
```

8. Since there is no weights so `pullUnderlying` function does nothing (weights.length is 0)

```
function pullUnderlying(uint256 amount, address from) private {
    for (uint256 i = 0; i < weights.length; i++) {
        uint256 tokenAmount = amount * weights[i] * ibRatio / BA
        IERC20(tokens[i]).safeTransferFrom(from, address(this), 
    }
}
```

9. Full amount 10 is minted to Publisher owned address setting `balanceOf(msg.sender) = 10`

```
_mint(to, amount);
```

10. Now Publisher calls the `publishNewIndex` to set new weights. Since `pendingWeights.pending` is false, else condition gets executed

```
function publishNewIndex(address[] memory _tokens, uint256[] mem
    validateWeights(_tokens, _weights);

    if (pendingWeights.pending) {
        require(block.number >= pendingWeights.block + TIMELOCK_
        if (auction.auctionOngoing() == false) {
            auction.startAuction();

            emit PublishedNewIndex(publisher);
        } else if (auction.hasBonded()) {

        } else {
            auction.killAuction();

            pendingWeights.tokens = _tokens;
            pendingWeights.weights = _weights;
            pendingWeights.block = block.number;
        }
    } else {
        pendingWeights.pending = true;
        pendingWeights.tokens = _tokens;
        pendingWeights.weights = _weights;
        pendingWeights.block = block.number;
    }
}
```

11. Publisher calls the `publishNewIndex` again which starts the Auction. This auction is later settled using the `settleAuction` function in Auction contract
12. Publisher owned address can now call burn and get the amount 10 even though he never made the payment since his `balanceOf(msg.sender) = 10` (Step 9)

```
function burn(uint256 amount) public override {
    require(auction.auctionOngoing() == false);
    require(amount > 0);
    require(balanceOf(msg.sender) >= amount);

    handleFees();

    pushUnderlying(amount, msg.sender);
    _burn(msg.sender, amount);

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

## Recommended Mitigation Steps

Change `validateWeights` to check for 0 length token

```
function validateWeights(address[] memory _tokens, uint256[] mem
    require(_tokens.length>0);
    ...
}
```

**frank-beard (defiProtocol) confirmed**

**Alex the Entreprenerd (judge) commented**:

> The warden identified a way to the publisher to effectively mint a ton of shares by setting weights to zero, for them to then change the weights and use their shares to dilute depositors.

> This may be a potential feature of the protocol at this time as there's no check for how the ratios relate to themselves

> The implications are actually pretty deep and I highly recommend the sponsor to consider the fact that if weights can be changed at any time by the publisher, they can effectively dilute / re-base the basket at any time very aggressively, to the detriment of users.

> As for the the finding by the warden, they have shown a specific way for the publisher to dilute other depositors, by setting up a basket with zero tokens initially.

> Becasue this is a specific attack that relies on external conditions, I'm downgrading the severity to medium

## [M-22] Incorrect data location specifier can be abused to cause DoS and fund loss

*Submitted by OxRajeev, also found by bw and pauliax*

### Impact

The `withdrawBounty()` loops through the `\_bounties` array looking for active bounties and transferring amounts from active ones. However, the data location specifier used for bounty is memory which makes a copy of the `\_bounties` array member instead of a reference. So when bounty.active is set to false, this is changing only the memory copy and not the array element of the storage variable. This results in bounties never being set to inactive, keeping them always active forever and every `withdrawBounty()` will attempt to transfer bounty amount from the Auction contract to the msg.sender.

Therefore, while the transfer will work the first time, subsequent attempts to claim this bounty will revert on transfer (because the Auction contract will not have required amount of bounty tokens) causing `withdrawBounty()` to always revert and therefore preventing settling of any auction.

A malicious attacker can add a tiny bounty on any/every Auction contract to prevent any reindexing on that contract to happen because it will always revert on auction settling. This can be used to cause DoS on any `auctionBonder` so as to make them lose their bondAmount because their bonded auction cannot be settled.

### Proof of Concept

- https://github.com/code-423n4/2021-09-defiProtocol/blob/52b74824c42acbcd64248f68c40128fe3a82caf6/contracts/contracts/Auction.sol#L143
- https://github.com/code-423n4/2021-09-defiProtocol/blob/52b74824c42acbcd64248f68c40128fe3a82caf6/contracts/contracts/Auction.sol#L143-L147
- https://docs.soliditylang.org/en/v0.8.7/types.html#data-location-and-assignment-behaviour

#### Tools Used

Manual Analysis

#### Recommended Mitigation Steps

Recommend changing storage specifier of bounty to "storage" instead of "memory".

frank-beard (Kuiper) confirmed

Alex the Entreprenerd (judge) commented:

> Great find, adding the same bounty multiple times can allow the exploiter to drain the entire contract as long as the bounties are denominated in the underlying token

Alex the Entreprenerd (judge) commented:

> Because the warden shows a way to cause DOS, which is contingent on setting it as well as redeeming that bounty (which can be avoided by simply not adding it to the bountiesID calldata), I will rate this finding as Medium Severity

> The DOS is low severity as it can be sidestep very easily, see #82

> However, the warden has identified a technical issue with the protocol (unflagging of bounty), as such I agree with a medium severity

### [M-23] Auction settler can steal user funds if bond timestamp is high enough

*Submitted by kenzo*

After an auction has started, as time passes and according to the `bondTimestamp`, `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 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-423n4/2021-09-defiProtocol/blob/52b74824c42acbcd64248f68c40128fe3a82caf6/contracts/contracts/Auction.sol#L89:#L99

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

    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 bondTimestamp 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());
```

frank-beard (Kuiper) confirmed

Alex the Entreprenerd (judge) commented:

> Would need confirmation from the sponsor here (this finding was also submitted on the more recent contest) As discount is part of the protocol (I think)

Alex the Entreprenerd (judge) commented:

> @frank-beard is the discount a feature or a bug?

Alex the Entreprenerd (judge) commented:

> As per this discussion: https://github.com/code-423n4/2021-10-defiprotocol-findings/issues/51

> The finding is valid and of medium severity

## Low Risk Findings (45)

- [L-01] Not handling approve return value *Submitted by WatchPug*
- [L-02] licenseFee state variable not checked for maximum value (Basket.sol) *Submitted by yeOlde*
- [L-03] No input validation on parameter changes *Submitted by anon*
- [L-04] block.timestamp is a better timer than block.number *Submitted by anon*
- [L-05] Add nonreentrant modifiers to external methods in 2 files *Submitted by anon, also found by OxRajeev, Oxsanson, goatbug, and pauliax*

- [L-06] Missing Zero-Address Checks *Submitted by shenwilly, also found by 0xRajeev and JMukesh*
- [L-07] Check the actual amounts transferred *Submitted by pauliax*
- [L-08] Variable assignment has no effect *Submitted by nikitastupin*
- [L-09] Lack of Event Logging and Input Validation *Submitted by leastwood*
- [L-10] Inaccurate log emitted at deleteNewIndex *Submitted by kenzo*
- [L-11] Cannot change pending while timelocked *Submitted by joeysantoro*
- [L-12] DAO is fee recipient / cannot revoke owner *Submitted by joeysantoro*
- [L-13] tokensNeeded can potentially be 0 *Submitted by itsmeSTYJ*
- [L-14] Proposals can never get created due to reaching `block.gaslimit` *Submitted by hrkrshnn*
- [L-15] Sanity checks when the contract parameters are updated *Submitted by hrkrshnn*
- [L-16] Event params are of no practical value *Submitted by hack3r-0m*
- [L-17] handleFees() will revert if licenseFee is too high *Submitted by gpersoon*
- [L-18] initialize of Basket.sol is missing initializer *Submitted by gpersoon, also found by 0xRajeev and loop*
- [L-19] Lack of zero ratio validation *Submitted by defsec*
- [L-20] Global bounties variable and 0 bounty allow dos in bounty functionality of basket *Submitted by csanuragjain*
- [L-21] `newIbRatio` update math depends on how often it's called *Submitted by cmichel*
- [L-22] Setting wrong publisher cannot be undone *Submitted by cmichel*
- [L-23] Re-entrancy in `Factory.createBasket` *Submitted by cmichel*
- [L-24] Wrong constant for `ONE_DAY` *Submitted by cmichel, also found by 0xRajeev*
- [L-25] specs not according to the docs *Submitted by JMukesh*
- [L-26] lack of checking of array length *Submitted by JMukesh, also found by 0xRajeev*
- [L-27] lack of checks in Factory.setAuctionMultiplier *Submitted by Alex the Entreprenerd*

- [L-28] Factory.sol - lack of checks for setMinLicenseFee *Submitted by Alex the Entreprenerd*
- [L-29] `handleFees` reverts if supply is zero *Submitted by 0xsanson*
- [L-30] Recognize the risk of using approve *Submitted by 0xRajeev*
- [L-31] Resetting partial struct fields is risky *Submitted by 0xRajeev*
- [L-32] Publisher role cannot be renounced *Submitted by 0xRajeev*
- [L-33] Missing support for (preventing) use of deflationary tokens in baskets *Submitted by 0xRajeev*
- [L-34] Risk of duplicate/scam basket token names/symbols may trick users *Submitted by 0xRajeev*
- [L-35] Missing sanity/threshold checks on critical contract parameter initializations *Submitted by 0xRajeev*
- [L-36] Missing timelocks for critical protocol parameter setters by owner *Submitted by 0xRajeev*
- [L-37] Timelock period may be less than 24 hours because it depends on `block.number` instead of `block.timestamp` *Submitted by nikitastupin*
- [L-38] Missing Transfer Ownership Pattern *Submitted by leastwood*
- [L-39] initialize function in basket.sol can be front-run *Submitted by jah*
- [L-40] use of approve() instead of safeApprove() *Submitted by JMukesh, also found by defsec*
- [L-41] block timestamp manipulation can cause fees change *Submitted by aga7hokakological*
- [L-42] `Auction.sol#settleAuction()` addBounty with a fake token could potentially disrupt `settleAuction()` *Submitted by WatchPug*
- [L-43] Incorrectly used new publisher and new licenseFee cannot be changed *Submitted by 0xRajeev*
- [L-44] `Factory.sol` Lack of two-step procedure and/or input validation routines for critical operations leaves them error-prone *Submitted by WatchPug, also found by 0xRajeev*
- [L-45] malicious tokens could be added with addBounty *Submitted by gpersoon, also found by jonah1005 and leastwood*

## Non-Critical Findings (39)

- [N-01] Using delete to clear variables instead of zero assignment *Submitted by 0xRajeev*
- [N-02] Protocol owner fee limit not verified correctly (Factory.sol) *Submitted by yeOlde*
- [N-03] Style issues *Submitted by pauliax*
- [N-04] emit NewIBRatio in function initialize *Submitted by pauliax*
- [N-05] Hardcoding numbers is error-prone *Submitted by pauliax*
- [N-06] Inconvenient to find bounty ids *Submitted by pauliax*
- [N-07] Timelocked functions doesn't emit proposal events *Submitted by nikitastupin, also found by JMukesh and WatchPug*
- [N-08] `proposal` declared as both a function and a Proposal in Factory *Submitted by loop*
- [N-09] Use of uint rather than uint256 *Submitted by loop*
- [N-10] BLOCK_DECREMENT not used *Submitted by gpersoon, also found by itsmeSTYJ, kenzo, and WatchPug*
- [N-11] Basket will break and lock all user funds if not used in 100 years *Submitted by kenzo*
- [N-12] Suggestion for incentive alignment *Submitted by itsmeSTYJ*
- [N-13] Misleading variable names *Submitted by itsmeSTYJ*
- [N-14] Use CEI pattern to align w/ best practices *Submitted by itsmeSTYJ*
- [N-15] More readable constants *Submitted by gpersoon*
- [N-16] `Auction.sol#bondTimestamp` Misleading name *Submitted by WatchPug*
- [N-17] Settle Time Limit not set correctly (Auction.sol) *Submitted by yeOlde*
- [N-18] Code lacking comments/spec *Submitted by loop*
- [N-19] Naming *Submitted by goatbug*
- [N-20] Uninitialized Implementation Contracts *Submitted by bw*
- [N-21] Events not emitted for parameter changes *Submitted by bw*
- [N-22] Event BasketLicenseProposed needs an idNumber *Submitted by 0xsanson*

- [N-23] `bondTimestamp` is not a timestamp but a blocknumber *Submitted by Oxsanson*
- [N-24] Lack of revert messages *Submitted by Oxsanson, also found by OxRajeev*
- [N-25] Max approvals are risky if contract is malicious/compromised *Submitted by OxRajeev*
- [N-26] Missing emission of basket ID and token composition *Submitted by OxRajeev*
- [N-27] Hardcoded constants are risky *Submitted by OxRajeev*
- [N-28] Lack of indexed event parameters will affect offchain monitoring *Submitted by OxRajeev*
- [N-29] Missing interfaces to determine available bounties *Submitted by OxRajeev*
- [N-30] Missing check for auctionOngoing is risky *Submitted by OxRajeev*
- [N-31] Lack of guarded launch approach may be risky *Submitted by OxRajeev*
- [N-32] Lack of input validation in initialize function of Basket.sol *Submitted by jah*
- [N-33] Lack of input validation in initialize function of Auction.sol *Submitted by jah*
- [N-34] Limit on growth size of pool - bond size *Submitted by goatbug*
- [N-35] Missing events for critical protocol parameter setters by owner *Submitted by OxRajeev*
- [N-36] 2-step change of publisher address and licenseFee does not generate warning event *Submitted by OxRajeev*
- [N-37] Using the latest compiler version may be susceptible to undiscovered bugs *Submitted by OxRajeev*
- [N-38] `mintTo` arguments order *Submitted by Oxsanson*
- [N-39] Malicious tokens can execute arbitrary code *Submitted by OxRajeev*

## Gas Optimizations (46)

- [G-01] Packing storage variables in Auction would save gas *Submitted by t11s, also found by OxRajeev, jah, JMukesh, kenzo, leastwood, and loop*

- [G-02] settleAuction should be external and arguments should use calldata *Submitted by t11s*
- [G-03] Eliminate hasBonded *Submitted by pauliax*
- [G-04] newIbRatio is not really useful *Submitted by pauliax*
- [G-05] Mint fees can be simplified *Submitted by pauliax*
- [G-06] Gas Optimization Wrt. Token Uniqueness *Submitted by leastwood, also found by cmichel, csanuragjain, itsmeSTYJ, joeysantoro, kenzo, and pauliax*
- [G-07] Set functions to external. *Submitted by goatbug, also found by 0xRajeev, 0xsanson, chasemartin01, hack3r-0m, pauliax, and WatchPug*
- [G-08] Dead code *Submitted by pauliax*
- [G-09] Double division by BASE *Submitted by pauliax*
- [G-10] Redundant call to external contract, result can be saved *Submitted by kenzo, also found by pauliax and WatchPug*
- [G-11] Replace `tokenList.length` by existing variable `length` *Submitted by hrkrshnn, also found by pauliax, and WatchPug*
- [G-12] Restore state to 0 if not needed anymore *Submitted by kenzo, also found by gpersoon*
- [G-13] Unnecessary initializing of variable to 0 *Submitted by kenzo, also found by 0xRajeev*
- [G-14] Some variables type should be changed *Submitted by jah*
- [G-15] Gas Saving by changing the visibility of initialize function from public to externa in basket.sol *Submitted by jah*
- [G-16] Only validateWeights when it is needed *Submitted by itsmeSTYJ*
- [G-17] set lastFee in initialize() function *Submitted by itsmeSTYJ*
- [G-18] Transfer tokens directly to the basket *Submitted by itsmeSTYJ*
- [G-19] Runtime constants not defined as immutable *Submitted by bw, also found by 0xRajeev and hrkrshnn*
- [G-20] The increment in for loop post condition can be made unchecked *Submitted by hrkrshnn*
- [G-21] Use `calldata` instead of `memory` for function parameters *Submitted by hrkrshnn*

- [G-22] redundant code (unused variables) *Submitted by hack3r-0m, also found by goatbug*
- [G-23] handleFees() only mint when necessary *Submitted by gpersoon, also found by bw and WatchPug*
- [G-24] Same tokens added to bounty *Submitted by goatbug*
- [G-25] pack structs *3 *Submitted by goatbug*
- [G-26] Unecessary transfer trips *Submitted by goatbug*
- [G-27] Gas optimation proposal struct *Submitted by goatbug*
- [G-28] Gas saving: pack struct *Submitted by goatbug*
- [G-29] Gas: Can save an sload in `changeLicenseFee` *Submitted by cmichel*
- [G-30] Gas: Can save an sload in `changePublisher` *Submitted by cmichel*
- [G-31] Gas: Factory parameter can be removed from Auction *Submitted by cmichel*
- [G-32] Use calldata instead of memory in function parameter declarations *Submitted by chasemartin01*
- [G-33] Require statement can be moved to start of function *Submitted by bw*
- [G-34] `Auction.sol#initialize()` Use msg.sender rather than factory_ parameter can save gas *Submitted by WatchPug, also found by 0xRajeev*
- [G-35] Adding unchecked directive can save gas *Submitted by WatchPug, also found by 0xRajeev*
- [G-36] Bounty list is never pruned to remove inactive bounties *Submitted by 0xRajeev*
- [G-37] Choose either explicit return or named return, not both *Submitted by 0xRajeev*
- [G-38] Avoiding unnecessary return can save gas *Submitted by 0xRajeev*
- [G-39] Loop can be skipped for i == 0 *Submitted by 0xRajeev*
- [G-40] Unnecessary initialization of loop index variable *Submitted by 0xRajeev*
- [G-41] Caching return values of external calls in local/memory variables avoids CALLs to save gas *Submitted by 0xRajeev*
- [G-42] Caching state variables in local/memory variables avoids SLOADs to save gas *Submitted by 0xRajeev*
- [G-43] Avoiding state variables in emits saves gas *Submitted by 0xRajeev*

- [G-44] Redundant Balance Check *Submitted by shenwilly*
- [G-45] Unnecessary require check *Submitted by anon*
- [G-46] Unused constant *Submitted by 0xRajeev*

## 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