Yieldy contest
Findings & Analysis Report

2022-09-27

Table of contents

- [M-03] MINTER*BURNER*ROLE can burn any amount of Yieldy from an arbitrary address
- [M-04] Arbitrage on `stake()`
- [M-05] Possible DOS (out-of-gas) on loops.
- [M-06] User can initiate withdraw for previous epoch if rebase hasn't been called since end of epoch
- [M-07] Withdrawals initiated after cycle withdrawal request won't be withdrawn in the correct cycle
- [M-08] Rebases can be frontrun with very little token downtime even when warmUpPeriod > 0
- [M-09] Users of Migration.sol may forfeit rebase rewards
- [M-10] No way to set CURVE_POOL approval after setting new curve pool address
- [M-11] Burn access control can be bypassed
- [M-12] Inconsistent balance when fee-on transfer tokens.
- [M-13] Sending batch withdrawal requests can possibly DoS
- [M-14] Incorrect rebase percentage calculation
- [M-15] token transfers in LiquidityReserve and Staking contract don't support deflationary ERC20 tokens, and user funds can be lost if stacking token was deflationary
- [M-16] `_storeRebase()` is called with the wrong parameters
- [M-17] Staking: rebase() does not rebase according to the status of the current epoch.
- [M-18] Removal of liquidity from the reserve can be griefed
- [M-19] Staking: the rebase function needs to be called before calling the function in the Yieldy contract that uses the rebasingCreditsPerToken variable
- [M-20] User fund lose in addLiquidity() of LiquidityReserve by increasing (totalLockedValue / totalSupply()) to very large number by attacker
- [M-21] Cannot mint to exactly max supply using `_mint` function
- [M-22] MINIMUM_LIQUIDITY checks missing - Bringing Liquidity below required min

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 Yieldy smart contract system written in Solidity. The audit contest took place between June 21—June 26, 2022.

Wardens

110 Wardens contributed reports to the Yieldy contest:

1. Lambda
2. Picodes
3. 0x1f8b
4. 0x52
5. cccz
6. unforgiven
7. IIIIIII
8. csanuragjain
9. berndartmueller
10. rfa
11. BowTiedWardens (BowTiedHeron, BowTiedPickle, m4rio_eth, Dravee, and BowTiedFirefox)
12. GalloDaSballo
13. asutorufos
14. sashik_eth
15. minhquanym
16. skoorch

17. WatchPug (jtp and ming)
18. MiloTruck
19. 0x29A (0x4non and rotcivegaf)
20. elprofesor
21. hansfriese
22. StErMi
23. pashov
24. shung
25. Chom
26. 0xNineDec
27. zzzitron
28. robee
29. hake
30. TrungOre
31. parashar
32. defsec
33. oyc_109
34. kenta
35. 0x1337
36. hubble (ksk2345 and shri4net)
37. PwnedNoMore (izhuer, ItsNio, and papr1ka2)
38. m_Rassska
39. 0xDjango
40. Metatron
41. neumo
42. reassor
43. _Adam
44. 0xNazgul
45. joestakey

46. TomJ
47. FudgyDRS
48. scaraven
49. Bnke0x0
50. fatherOfBlocks
51. antonttc
52. GimelSec (rayn and sces60107)
53. exd0tpy
54. 0xf15ers (remora and twojoy)
55. Waze
56. ladboy233
57. Sm4rty
58. Noah3o6
59. Funen
60. Limbooo
61. sikorico
62. aga7hokakological
63. delfin454000
64. ElKu
65. JC
66. Kaiziron
67. simon135
68. mics
69. UnusualTurtle
70. 0xmint
71. ych18
72. pedr02b2
73. ajtra
74. Fabble

75. 0xc0ffEE
76. cryptphi
77. dipp
78. samruna
79. ak1
80. sseefried
81. PumpkingWok
82. tchkvsky
83. 0xkatana
84. 0xKitsune
85. RedOneN
86. Tomio
87. Nyamcil
88. Randyyy
89. c3phas
90. 8olidity
91. Fitraldys
92. saian
93. 0v3rf10w
94. ACai
95. bardamu
96. sach1r0
97. s3cunda
98. slywaters
99. ignacio

This contest was judged by the Float Capital team: moose-code, JasoonS & denhampreen.

Final report assembled by itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 31 unique vulnerabilities. Of these vulnerabilities, 4 received a risk rating in the category of HIGH severity and 27 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 70 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 70 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Yieldy contest repository, and is composed of 5 smart contracts written in the Solidity programming language and includes 892 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/non-critical.

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 (4)

[H-01] No withdrawal possible for ETH TOKE pool

Submitted by Lambda

The `withdraw` function of the ETH Tokemak pool has an additional parameter `asEth`. This can be seen in the Tokemak Github repository or also when looking at the deployed code of the ETH pool. Compare that to e.g. the USDC pool, which does not have this parameter.

This means that the call to `withdraw` will when the staking token is ETH / WETH and no withdrawals would be possible.

Proof of Concept

A new `Staking` contract with ETH / WETH as the staking token is deployed. Deposits in Tokemak work fine, so users stake their tokens. However, because of the previously described issue, no withdrawal is possible, leaving the funds locked.

Recommended Mitigation Steps

Handle the case where the underlying asset is WETH / ETH separately and pass this boolean in that case.

toshiSat (Yieldy) confirmed and resolved

[H-02] `Staking.sol#stake()` DoS by staking 1 wei for the recipient when `warmUpPeriod > 0`

Submitted by WatchPug, also found by BowTiedWardens, cccz, minhquanym, parashar, pashov, shung, and zzzitron

```
    if (warmUpPeriod == 0) {
        IYieldy(YIELDY_TOKEN).mint(_recipient, _amount);
    } else {
        // create a claim and mint tokens so a user can claim them c
```

```
        warmUpInfo[_recipient] = Claim({
            amount: info.amount + _amount,
            credits: info.credits +
                IYieldy(YIELDY_TOKEN).creditsForTokenBalance(_amount
            expiry: epoch.number + warmUpPeriod
        });

        IYieldy(YIELDY_TOKEN).mint(address(this), _amount);
    }
```

`Staking.sol#stake()` is a public function and you can specify an arbitrary address as the `_recipient`.

When `warmUpPeriod > 0`, with as little as 1 wei of `YIELDY_TOKEN`, the `_recipient`'s `warmUpInfo` will be push back til `epoch.number + warmUpPeriod`.

Recommended Mitigation Steps

Consider changing to not allow deposit to another address when `warmUpPeriod > 0`.

Dravee (warden) commented:

> Should be high right? Funds are locked. See https://github.com/code-423n4/2022-06-yieldy-findings/issues/245#issuecomment-1167616593

moose-code (judge) increased severity to High and commented:

> Agree this should be high. The cost of the attack is negligible and could cause basic perpetual grievance on all users with one simple script.

toshiSat (Yieldy) confirmed

[H-03] Denial of Service by wrong `BatchRequests.removeAddress` logic

Submitted by 0x1f8b, also found by rfa, berndartmueller, BowTiedWardens, csanuragjain, Lambda, neumo, and StErMi

Note: issues #283, 115, 82, 89, 61, and 241 were originally broken out as a separate medium issue. Approximately 1 week after judging and awarding were finalized, the judging team re-assessed that these should have all been grouped under H-03. Accordingly, the 6 warden names have been added as submitters above.

https://github.com/code-423n4/2022-06-yieldy/blob/34774d3f5e9275978621fd20af4fe466d195a88b/src/contracts/BatchRequests.sol#L93

https://github.com/code-423n4/2022-06-yieldy/blob/34774d3f5e9275978621fd20af4fe466d195a88b/src/contracts/BatchRequests.sol#L57

https://github.com/code-423n4/2022-06-yieldy/blob/34774d3f5e9275978621fd20af4fe466d195a88b/src/contracts/BatchRequests.sol#L37

Impact

The `BatchRequests.removeAddress` logic is wrong and it will produce a denial of service.

Proof of Concept

Removing the element from the array is done using the `delete` statement, but this is not the proper way to remove an entry from an array, it will just set that position to `address(0)`.

Append dummy data:

- `addAddress('0x0000000000000000000000000000000000000001')`
- `addAddress('0x0000000000000000000000000000000000000002')`
- `addAddress('0x0000000000000000000000000000000000000003')`
- `getAddresses() => address[]: 0x0000000000000000000000000000000000000001,0x0000000000000000000000000000000000000002,0x0000000000000000000000000000000000000003`

Remove address:

- `removeAddress(0x0000000000000000000000000000000000000002)` (or `0x0000000000000000000000000000000000000003`)
- `getAddresses() => address[]: 0x0000000000000000000000000000000000000001,0x0000000000000000000000000000000000000002,0x0000000000000000000000000000000000000003`

Service is denied because it will try to call `canBatchContracts` to `address(0)`.

Recommended Mitigation Steps

- To remove an entry in an array you have to use `pop` and move the last element to the removed entry position.

Oxean (Yieldy) confirmed and resolved

JasoonS (judge) commented:

> Agree this is high, if the team (owner) didn't know this they could cause some issues for sure.

[H-04] Yield of `LiquidityReserve` can be stolen

Submitted by Picodes

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/LiquidityReserve.sol#L126

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/LiquidityReserve.sol#L176

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/LiquidityReserve.sol#L206

Impact

Using sandwich attacks and JIT (Just-in-time liquidity), the yield of `LiquidityReserve` could be extracted for liquidity providers.

Proof of Concept

The yield of `LiquidityReserve` is distributed when a user calls `instantUnstakeReserve()` in `Staking`. Then, in `instantUnstake`, `totalLockedValue` increases with the fee paid by the user withdrawing. The fee is shared between all liquidity providers as they all see the value of their shares increase.

Therefore, an attacker could do the following sandwich attack when spotting a call to `instantUnstakeReserve()`.

- In a first tx before the user call, borrow a lot of `stakingToken` and `addLiquidity`
- The user call to `instantUnstakeReserve()` leading to a fee of say `x\`
- In a second tx after the user call, `removeLiquidity` and repay the loan, taking a large proportion of the user fee

The problem here is that you can instantly add and remove liquidity without penalty, and that the yield is instantly distributed.

Recommended Mitigation Steps

To mitigate this, you can

- store the earned fees and distribute them across multiple blocks to make sure the attack wouldn't be worth it
- add a small fee when removing liquidity, which would make the attack unprofitable
- prevent users from withdrawing before X blocks or add a locking mechanism

Oxean (Yieldy) disagreed with severity and commented:

> This is not unique to the protocol and is a vulnerability in almost all of the LP designs that are prevalent today. There is no loss of user funds here either.

> Would downgrade to Low or QA.

Picodes (warden) commented:

> In standard cases of JIT, for example in a DEX, the attacker takes a risk as the liquidity he adds is used during the swap, and this liquidity is useful for the protocol as leads to a better price for the user, which is not the case here

Oxean (Yieldy) commented:

> @Picodes - that is fair but the liquidity is still useful and I still don't see how this qualifies as high severity. Eventually it would mean that the liquidity reserve would need less liquidity parked in it if JITers always where hitting it.

Picodes (warden) commented:

> To me it's high because: (correct me if I am missing things)

- JIT is not useful here at all for the protocol, the liquidity they bring is not useful as does not get locked. It's totally risk free, and as you said it's a commun attack so it's likely that someone uses it
- It leads to a loss of LP funds: Assume there is 100k unlocked in the pool, and someone `instantUnstake` 100k, it'll lock all the LP liquidity. But if someone JITs this, the fees will go to the attacker and not the LP which provided the service by accepting to have its liquidity locked.
- From a protocol point of view, LPing becomes unattractive as all the fees are stolen, breaking the product design

moose-code (judge) commented:

> Agree going to leave this as high. Any whale that does a large unstake will be susceptible to having more of the fee's eroded to a predatory sandwich attack which provides no value to the system.

Medium Risk Findings (27)

[M-01] Unsecure `transferFrom`

Submitted by 0x1f8b, also found by 0xNineDec and StErMi

The security of the `Yieldy` contract is delegated to the compiler used.

Proof of Concept

The `allowance` of an account does not have to reflect the real balance of an account, however in the `transferFrom` method, it is the value that is checked in order to verify that the user has enough balance to make the transfer.

```
    function transferFrom(
        address _from,
        address _to,
        uint256 _value
    ) public override returns (bool) {
        require(_allowances[_from][msg.sender] >= _value, "Allow
```

However, the real balance of the `Yieldy` contract is based on the calculation made by the `creditsForTokenBalance` method, so an underflow could be made in the subtraction of the balance of the `from` account.

```
        uint256 creditAmount = creditsForTokenBalance(_value);
        creditBalances[_from] = creditBalances[_from] - creditAm
        creditBalances[_to] = creditBalances[_to] + creditAmount
        emit Transfer(_from, _to, _value);
```

This means that the security of the contract is delegated to the checks added by the compiler depending on the pragma used, it must be taken into account that these checks may appear and disappear in future versions of the compiler, so they must be checked at the level of smart contracts.

Affected source code:

- Yieldy.sol#L212

Recommended Mitigation Steps

- Check that the from account has a `creditAmount` balance.

toshiSat (Yieldy) confirmed and commented:

> Looking into this, the balance isn't calculated through the `creditsForTokenBalance` method, it's calculated through the `balanceOf` method, which in this case the functionality is correct. We aren't transferring credits, we are transferring the value and adding to the credits. Allowance is for value amounts, not credits, also balance can only go up against credits, so if the balance is valid then credits are inherently valid too. I'm unsure of what to label this as, because we do need to check to see if the user has the correct balance. I feel like this issue is partially correct.

toshiSat (Yieldy) resolved:

> https://github.com/shapeshift/foxy/pull/130/files for the fix.

[M-02] It's possible to perform DOS and fund lose in Stacking by transferring tokens directly to contract

Submitted by unforgiven

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/Yieldy.sol#L78-L102

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/Staking.sol#L698-L719

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/Staking.sol#L401-L417

Impact

Function `rebase()` in contract `Staking` calls `Yieldy.rebase(profit, )` and `Yieldy.rebase(profit, )` would revert if `rebasingCredits / updatedTotalSupply` was equal to `0`. it's possible to transfer some `STAKING_TOKEN` directly to `Stacking` contract before or after deployment of `Staking` and make `rebasingCredits / updatedTotalSupply` equal to `0`, and then most of the functionalities of `Staking` would not work because they call `rebase()` which will revert. it's possible to perform this DOS for any token by transferring some tokens `STAKING_TOKEN` to contract address and then staking `1 wei` in contract. or for tokens with low precisions and low price it's even possible to perform when `totalSupply()` of `Yieldy` is low.

Also attacker can only make `rebasingCredits / updatedTotalSupply` too low and so rounding error would be significant when `rebasingCredits / updatedTotalSupply` gets too low and users funds would be lost because of rounding error.

Proof of Concept

This is `rebase()` code in `Yieldy`:

```
    function rebase(uint256 _profit, uint256 _epoch)
        external
        onlyRole(REBASE_ROLE)
    {
        uint256 currentTotalSupply = _totalSupply;
        require(_totalSupply > 0, "Can't rebase if not circulati

        if (_profit == 0) {
            emit LogSupply(_epoch, block.timestamp, currentTotal
            emit LogRebase(_epoch, 0, getIndex());
        } else {
            uint256 updatedTotalSupply = currentTotalSupply + _p

            if (updatedTotalSupply > MAX_SUPPLY) {
                updatedTotalSupply = MAX_SUPPLY;
            }

            rebasingCreditsPerToken = rebasingCredits / updatedT
            require(rebasingCreditsPerToken > 0, "Invalid change
```

```
            _totalSupply = updatedTotalSupply;

            _storeRebase(updatedTotalSupply, _profit, _epoch);
        }
    }
```

As you can see if `rebasingCredits / updatedTotalSupply == 0` then the code will revert. `updatedTotalSupply` is equal to `_totalSupply + _profit` and `rebasingCredits` is `wad` in the first. `Yieldy.rebase(profit,)` is called by `Staking.rebase()` :

```
    function rebase() public {
        // we know about the issues surrounding block.timestamp,
        if (epoch.endTime <= block.timestamp) {
            IYieldy(YIELDY_TOKEN).rebase(epoch.distribute, epoch

            epoch.endTime = epoch.endTime + epoch.duration;
            epoch.timestamp = block.timestamp;
            epoch.number++;

            uint256 balance = contractBalance();
            uint256 staked = IYieldy(YIELDY_TOKEN).totalSupply()

            if (balance <= staked) {
                epoch.distribute = 0;
            } else {
                epoch.distribute = balance - staked;
            }
        }
    }
```

As you can see the value of `_profit` is set to `epoch.distribute` which is `contractBalance() - IYieldy(YIELDY_TOKEN).totalSupply()` and `contractBalance()` is sum of `STAKING_TOKEN` and `TOKE` balance of `Staking` contract. so if attacker transfers `X` amount of `STAKCING_TOKEN` directly to `Staking` contract then the value of `_profit` which is going to send to `Yieldy.rebase(profit,)` would be higher than `X` . to exploit this attacker call `stake(1 wei)` after `Staking` deployment and then transfer `STAKING_TOKEN`

directly to contract. then the value of `rebasingCredits` in `Yieldy` would be `2 wad` and the value of `_profit` sent to `Yieldy.rebase(profit,)` would be bigger than `2 wad` and `rebasingCredits / updatedTotalSupply` would be `0` and from now on all calls to `Staking.rebase()` would revert and that means functions `Stake()` and `instantUnstakeReserve()` and `instantUnstakeCurve()` wouldnt work anymore.

It's possible to perform this attack for low precision tokens with low price `STAKING_TOKEN` too. the only thing attacker needs to do is that in early stage of `Staking` deployment sends more than `rebasingCredits` of `STAKING_TOKEN` token directly to `Staking` contract address. then in `rebase()` contract send that amount as `profit` to `Yieldy.rebase()` and that call would revert which will cause most of the logics of `Staking` to revert and not work.

and when `rebasingCredits / updatedTotalSupply` is low, the rounding error would be high enough that the compounding yield won't show itself. attacker can make `rebasingCredits / updatedTotalSupply` too low but not `0` and from then user's funds would be lost because of rounding error (wrong number of `Yieldy` token would be mint for user).

Tools Used

VIM

Recommended Mitigation Steps

The default initial value of `rebasingCredits` should be very high that attacker couldn't perform this attack.

[toshiSat (Yieldy) acknowledged and commented](#):

> `rebasingCredits` is in credits and `updatedTotalSupply` is in token amounts. So if even a small amount is staked, the attacker will have to send a large amount to go through with this attack vector for no financial gain.

> This seems like a low priority issue mainly because user funds won't get lost as if this were to occur then most likely no one has staked and the worst case scenario

> we will redeploy the contract. Also, we will be staking on every yieldy as we deploy.

> I think we would like to eventually solve this, but for now this seems like it won't fall in our list of fixes for this iteration.

JasoonS (judge) decreased severity to Medium:

> Downgrading to medium.

[M-03] MINTER*BURNER*ROLE can burn any amount of Yieldy from an arbitrary address

Submitted by 0x1f8b

Using the `burn()` function of `Yieldy`, an address with `MINTER_BURNER_ROLE` can burn an arbitrary amount of tokens from any address.

We believe this is unnecessary and poses a serious centralization risk.

A malicious or compromised `MINTER_BURNER_ROLE` address can take advantage of this.

Recommended Mitigation Steps

Consider removing the `MINTER_BURNER_ROLE` and change `burn()` function to:

```
function burn(uint256 _amount) external override
{
    _burn(_msgSender(), _amount);
}
```

toshiSat (Yieldy) acknowledged and commented:

> There's tons of centralization risks already, this is acknowledged, but for yieldies to work, there needs to be a trusted party.

JasoonS (judge) commented:

> Leaving as medium - the code can be upgraded but the code is being assessed as is.

[M-04] Arbitrage on `stake()`

Submitted by BowTiedWardens, also found by WatchPug

Issue: there is a huge arb opportunity for people who deposit 1 block before the `rebase()`.

Consequences: then they can call `instantUnstakeReserve` or `instantUnstakeCurve` to unstake the staked amount, in this way the profit that needs to be distributed on the next rebase increases, he also messes up the rewards for the other holders as the `instantUnstakeReserve` does not burn the `YIELD_TOKEN`. Even if there is a fee on the `instantUnstakeReserve`, there is still a chance for profit.

Affected Code

```
File: Staking.sol
406:     function stake(uint256 _amount, address _recipient) pub
407:         // if override staking, then don't allow stake
408:         require(!isStakingPaused, "Staking is paused");
409:         // amount must be non zero
410:         require(_amount > 0, "Must have valid amount");
411:
412:         uint256 yieldyTotalSupply = IYieldy(YIELDY_TOKEN).t
413:
414:         // Don't rebase unless tokens are already staked or
415:         if (yieldyTotalSupply > 0) {
416:             rebase();
417:         }
418:
419:         IERC20Upgradeable(STAKING_TOKEN).safeTransferFrom(
420:             msg.sender,
421:             address(this),
422:             _amount
423:         );
424:
425:         Claim storage info = warmUpInfo[_recipient];
```

```
426:
427:         // if claim is available then auto claim tokens
428:         if (_isClaimAvailable(_recipient)) {
429:             claim(_recipient);
430:         }
431:
432:         _depositToTokemak(_amount);
433:
434:         // skip adding to warmup contract if period is 0
435:         if (warmUpPeriod == 0) {
436:             IYieldy(YIELDY_TOKEN).mint(_recipient, _amount)
437:         } else {
438:             // create a claim and mint tokens so a user car
439:             warmUpInfo[_recipient] = Claim({
440:                 amount: info.amount + _amount,
441:                 credits: info.credits +
442:                     IYieldy(YIELDY_TOKEN).creditsForTokenBa
443:                 expiry: epoch.number + warmUpPeriod
444:             });
445:
446:             IYieldy(YIELDY_TOKEN).mint(address(this), _amou
447:         }
448:
449:         sendWithdrawalRequests();
450:     }
```

Recommended Mitigation Steps

Burn the `YIELD_TOKEN` amount in the `instantUnstakeReserve`.

Oxean (Yieldy) acknowledged, but disagreed with severity and commented:

> Yes, the fee on instant Unstake needs to be set high enough to make this not profitable.

> If a curve pool exists, then this does become possible to arb the rebase and something that should be fixed, potentially with not allowing the warm up period to be violated for instant unstaking (through curve at the very least).

> I would qualify this as Medium severity, and leaking value.

> 2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.

JasoonS (judge) decreased severity to Medium and commented:

> I took another look, medium seems reasonable too.

[M-05] Possible DOS (out-of-gas) on loops.

Submitted by 0x29A, also found by minhquanym

https://github.com/code-423n4/2022-06-yieldy/blob/main/src/contracts/BatchRequests.sol#L16

https://github.com/code-423n4/2022-06-yieldy/blob/main/src/contracts/BatchRequests.sol#L36

https://github.com/code-423n4/2022-06-yieldy/blob/main/src/contracts/BatchRequests.sol#L91

Impact

It is possible to get an out-of-gas issue while iterating the for loop.

Please take a look at this link.

Proof of Concept

Let's say I want to run the function on `BatchRequests.sol#L14` and I got a lot of `contracts` pending for withdrawal.

Recommended Mitigation Steps

Use this pattern;

```
        /**
            @notice sendWithdrawalRequests on all addresses in contr
```

```
     */
    function sendWithdrawalRequests(uint256 from, uint256 to) ex
        uint256 contractsLength = contracts.length;
        require(from < contractsLength, "Invalid from");
        require(to <= contractsLength, "Invalid to");
        for (uint256 i = from; i < to; ) {
            if (
                contracts[i] != address(0) &&
                IStaking(contracts[i]).canBatchTransactions()
            ) {
                IStaking(contracts[i]).sendWithdrawalRequests();
            }
            unchecked {
                ++i;
            }
        }
    }
```

Oxean (Yieldy) acknowledged

[M-06] User can initiate withdraw for previous epoch if rebase hasn't been called since end of epoch

Submitted by 0x52

User is able to withdraw unstaked asset sooner than they should be.

Proof of Concept

`Unstake()` allows the user to bypass the rebase() call by setting _trigger to false. Since rebase() is bypassed, epoch.number could potentially be stale i.e. doesn't match the Tokemak epoch. A user could potentially call unstake() with _trigger = false immediately after an epoch has ended but expiry would be set using the stale epoch.number because it wouldn't be updated by rebase(). This would allow the user to withdraw early before their funds were actually available in the contract because their withdrawal would be considered to be in the epoch before they actually initiated the withdrawal.

Recommended Mitigation Steps

`Rebase()` cannot be optional when calling unstake.

toshiSat (Yieldy) acknowledged and commented:

> We use a coolDownAmount of 2 to get around this.

[M-07] Withdrawals initiated after cycle withdrawal request won't be withdrawn in the correct cycle

Submitted by 0x52, also found by lllllll

Some user withdrawals won't be available for withdrawal even though it should be.

Proof of Concept

`sendWithdrawalRequest` can only happen once per cycle. canBatchTransactions (L386) must return true for the actual withdrawal request to happen. It checks in L362 that currentCycleIndex > lastTokeCycleIndex and in L397 of sendWithdrawlRequest, lastTokeCycleIndex = currentCycleIndex. This means that for the rest of the cycle, canBatchTransactions will return false. This means that if a user requests a withdrawal towards the end of epoch after the withdrawal has been submitted but before the end of the epoch, their withdrawal will be set to the current epoch but the actual token amount of their withdrawal won't be processed. This will lead to a discrepancy between the number of tokens withdrawn and the number of tokens allowed to be withdrawn by users, which means that not all users who are "eligible" for withdrawal will actually be able to withdraw because there won't be enough tokens for everyone.

Recommended Mitigation Steps

The requirement in L362 should be removed. As noted in the contract, "TOKE's requestWithdrawal overwriting the amount if you call it more than once per cycle". Overwriting the withdrawal is perfectly okay though because it already uses requestWithdrawalAmount which is a cumulative measure of the tokens that need to be withdrawn because it is only decreased when the asset it actually received from a withdrawal.

toshiSat (Yieldy) acknowledged and commented:

> We get around this by having a coolDownPeriod of 2. It saves on gas and we only have to call `sendWithdrawalRequest` once per cycle.

[M-08] Rebases can be frontrun with very little token downtime even when warmUpPeriod > 0

Submitted by 0x52, also found by elprofesor

Rebases can be frontrun with very little token downtime even when `warmUpPeriod > 0`.

Proof of Concept

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/Staking.sol#L415-L417

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/Staking.sol#L703

A user can call stake the block before epoch.endTime <= block.timestamp, allowing the user to bypass the forced rebase called in L416 of the the stake function. If warmUpPeriod > 0 then the user will receive a “warmUpInfo” with the value of their deposit. The very next block, the user can then call instantUnstakeCurve.

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/Staking.sol#L600-L627

This will call rebase again in L633 and this time epoch.endTime <= block.timestamp will be true and it will trigger an actual rebase, distributing the pending rewards. _retrieveBalanceFromUser (L617) will then allows the user to unstake all the funds locked in warm up. The issue is that when unstaking it uses userWarmInfo.credits meaning that any rebalance rewards are kept. This allows the user to get in, collect the rebase, then immediately get out.

Recommended Mitigation Steps

Being able to unstake tokens even when in the warm up period is a useful feature but tokens unstaked during that period should not be allowed to accumulate any rebases or it can lead to situations like this. L537-L539 should be changed to: `warmUpBalance = userWarmInfo.amount`.

toshiSat (Yieldy) acknowledged

[M-09] Users of Migration.sol may forfeit rebase rewards

Submitted by 0x52, also found by berndartmueller

Users of `moveFundsToUpgradedContract()` in migration.sol may forfeit rebase rewards.

Proof of Concept

L54 calls instantUnstake(false) meaning that it skips the optional rebase. If there is a pending rebase then the user calling the function will not get this rebase and miss out on potential rewards.

Recommended Mitigation Steps

Add an input bool `\_trigger` then add the following code to the start of the function:

if (_trigger) { rebase(); }

This allows users to optionally call rebase if they are concerned about losing pending rebase rewards.

Oxean (Yieldy) acknowledged

[M-10] No way to set CURVE_POOL approval after setting new curve pool address

Submitted by 0xDjango, also found by BowTiedWardens, cccz, hansfriese, Metatron, shung, ych18, and zzzitron

`Staking.setCurvePool()` allows the owner to set a new `CURVE_POOL` address, however, there is no way to set token approvals to the new address. The only calls to `token.approve()` are found in the constructor. Therefore, there's no true way to set a new curve pool. All calls to `ICurvePool(CURVE_POOL).exchange()` will fail.

Recommended Mitigation Steps

Set approvals for the new curve pool address in the same `setCurvePool()` function.

Oxean (Yieldy) acknowledged

[M-11] Burn access control can be bypassed

Submitted by pashov, also found by csanuragjain, hake, kenta, m_Rassska, and oyc_109

`transferFrom` method does not check if `_to` argument is the zero address.

Impact

This can lead to token burns without calling the `burn` function, which has access control `onlyRole(MINTER_BURNER_ROLE)` but here this can be bypassed by passing the zero address as the value of `_to`.

Recommended Mitigation Steps

Add a non-zero address check for `_to` argument in `transferFrom`.

toshiSat (Yieldy) confirmed and resolved

[M-12] Inconsistent balance when fee-on transfer tokens.

Submitted by asutorufos

There are ERC20 tokens that may make certain customizations to their ERC20 contracts.

One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`.

Proof of Concept

https://github.com/code-423n4/2022-06-yieldy/blob/main/src/contracts/Staking.sol#:~:text=Invalid%20address%22)%3B-,uint256%20totalTokeAmount%20%3D%20IERC20Upgradeable(TOKE_TOKEN).balanceOf(,)%3B,-%7D

When `IERC20Upgradeable(TOKE_TOKEN)` get set to `totalTokeAmount` it will be different once `safetransfer` have fees as some types of tokens may charge a certain fee for transfer and transferfrom.

It may be better to get the before balance then `safetransferfrom` then get the after balance to make sure no fees were added.

toshiSat (Yieldy) acknowledged and commented:

> We will not support deflationary tokens. We will document this.

[M-13] Sending batch withdrawal requests can possibly DoS

Submitted by berndartmueller

The function `BatchRequests.sendWithdrawalRequests` allows calling the `sendWithdrawalRequests` function on all of the Yieldy contracts at once. However, due to the unbounded `for` loop, if many Yieldy contracts are added to `contracts`, this function can potentially DoS due to reaching the block gas limit.

Proof of Concept

BatchRequests.sendWithdrawalRequests

```
    function sendWithdrawalRequests() external {
        uint256 contractsLength = contracts.length;
        for (uint256 i; i < contractsLength; ) {
            if (
                contracts[i] != address(0) &&
                IStaking(contracts[i]).canBatchTransactions()
            ) {
                IStaking(contracts[i]).sendWithdrawalRequests();
            }
            unchecked {
                ++i;
            }
        }
    }
```

Recommended mitigation steps

Add `offset` and `limit` function parameters to implement a "paginated" for loop.

toshiSat (Yieldy) acknowledged and commented:

> Only the owner of the contract can add addresses to the contract.

JasoonS (judge) commented:

> Hmm, would consider making this Low. But keeping it medium highlights the importance to be aware of this.

[M-14] Incorrect rebase percentage calculation

Submitted by csanuragjain

Note: this issue had originally been grouped with M-15. Approximately 1 week after judging and awarding were finalized, the judging team re-assessed that this issue should have been classified as a unique issue. It has been broken out here accordingly.

https://github.com/code-423n4/2022-06-yieldy/blob/main/src/contracts/Yieldy.sol#L91

It was observed that if updatedTotalSupply > MAX*SUPPLY then updatedTotalSupply becomes MAX*SUPPLY. This means _profit amount is not fully used. But _storeRebase function is still called with _profit amount.

This becomes a problem since _storeRebase function caluclates rebasePercent using this incorrect _profit amount.

Proof of Concept

1. REBASE_ROLE calls rebase function with say profit 10. Assume currentTotalSupply is 90
2. updatedTotalSupply is calculated as updatedTotalSupply = currentTotalSupply + _profit. Thus updatedTotalSupply becomes 90+10=100
3. Assume MAX*SUPPLY is 91. Since updatedTotalSupply>MAX*SUPPLY so updatedTotalSupply is updated to be 91
4. Now _storeRebase function is called with updatedTotalSupply (91), _profit(10)
5. This is incorrect since 10 amount from _profit is not utilized and only 1 amount is utilized. This becomes a problem in rebasePercent calculation where it is calculated on full 10 amount instead of 1

Recommended Mitigation Steps

Use below:

```
    if (updatedTotalSupply > MAX_SUPPLY) {
    _profit=_profit - (updatedTotalSupply-MAX_SUPPLY);
                    updatedTotalSupply = MAX_SUPPLY;

                }
```

toshiSat (Yieldy) acknowledged and commented:

> Max Supply is nearly the max amount in uint256. The protection is there, but will most likely never hit.

JasoonS (judge) commented:

> Potentially, this should be a medium issue.

JasoonS (judge) decreased severity to Medium and commented:

> Downgrading to medium

[M-15] token transfers in LiquidityReserve and Staking contract don't support deflationary ERC20 tokens, and user funds can be lost if stacking token was deflationary

Submitted by unforgiven, also found by hake, robee, and TrungOre

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/Staking.sol#L419-L445

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/LiquidityReserve.sol#L120-L126

Impact

If the token is deflationary then contract will receive less token that requested `amount` but contract don't check for the real transferred amount. because this is happening in receiving `stacking_token` in `addLiquidity()` of `LiquidityReserve` and `stake()` of `Staking` then those logics for minting `YIELDY_TOKEN` or `LP` token is wrong. (contract receive less than `amount` but mint or transfer `amount` to user). This can cause other users which staked to lose funds.

Proof of Concept

This is the related code where transfer happens (`stake()` and `addLiquidity()`):

```
    function stake(uint256 _amount, address _recipient) public {
        // if override staking, then don't allow stake
        require(!isStakingPaused, "Staking is paused");
        // amount must be non zero
        require(_amount > 0, "Must have valid amount");

        uint256 yieldyTotalSupply = IYieldy(YIELDY_TOKEN).totalS
```

```
    // Don't rebase unless tokens are already staked or coul
    if (yieldyTotalSupply > 0) {
        rebase();
    }

    IERC20Upgradeable(STAKING_TOKEN).safeTransferFrom(
        msg.sender,
        address(this),
        _amount
    );

    Claim storage info = warmUpInfo[_recipient];

    // if claim is available then auto claim tokens
    if (_isClaimAvailable(_recipient)) {
        claim(_recipient);
    }

    _depositToTokemak(_amount);

    // skip adding to warmup contract if period is 0
    if (warmUpPeriod == 0) {
        IYieldy(YIELDY_TOKEN).mint(_recipient, _amount);
    } else {
        // create a claim and mint tokens so a user can clai
        warmUpInfo[_recipient] = Claim({
            amount: info.amount + _amount,
            credits: info.credits +
                IYieldy(YIELDY_TOKEN).creditsForTokenBalance
            expiry: epoch.number + warmUpPeriod
        });

        IYieldy(YIELDY_TOKEN).mint(address(this), _amount);
    }



function addLiquidity(uint256 _amount) external {
    require(isReserveEnabled, "Not enabled yet");
    uint256 stakingTokenBalance = IERC20Upgradeable(stakingT
        address(this)
    );
    uint256 rewardTokenBalance = IERC20Upgradeable(rewardTok
        address(this)
    );
```

```
        uint256 lrFoxSupply = totalSupply();
        uint256 coolDownAmount = IStaking(stakingContract)
            .coolDownInfo(address(this))
            .amount;
        uint256 totalLockedValue = stakingTokenBalance +
            rewardTokenBalance +
            coolDownAmount;

        uint256 amountToMint = (_amount * lrFoxSupply) / totalLc
        IERC20Upgradeable(stakingToken).safeTransferFrom(
            msg.sender,
            address(this),
            _amount
        );
        _mint(msg.sender, amountToMint);
    }
```

As you can see contract transfers `amount` of `STAKE_TOKEN` and assumes it is going to receive that amount and then mint the same `amount` of `YIELDY_TOKEN` or `LP` token. So user receive more funds which belongs to other users. protocol logics are not suitable for deflationary tokens and funds would be lost.

Tools Used

VIM

Recommended Mitigation Steps

Check the real amount of tokens that the contract receives.

toshiSat (Yieldy) acknowledged and commented:

> We will not be supporting deflationary tokens in Yieldy. We will document this.

[M-16] `_storeRebase()` is called with the wrong parameters

Submitted by BowTiedWardens, also found by hansfriese, hubble, minhquanym, PwnedNoMore, shung, TrungOre, and WatchPug

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/Yieldy.sol#L110-L114

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/Yieldy.sol#L97-L100

Vulnerability Details

`_storeRebase()` 's signature is as such:

- Yieldy.sol#_storeRebase()

```
File: Yieldy.sol
104:     /**
105:         @notice emits event with data about rebase
106:         @param _previousCirculating uint
107:         @param _profit uint
108:         @param _epoch uint
109:      */
110:     function _storeRebase(
111:         uint256 _previousCirculating,
112:         uint256 _profit,
113:         uint256 _epoch
114:     ) internal {
```

However, instead of being called with the expected `_previousCirculating` value, it's called with the current circulation value:

- Yieldy.sol#rebase()

```
File: Yieldy.sol
89:             uint256 updatedTotalSupply = currentTotalSupply
...
103:             _totalSupply = updatedTotalSupply;
104:
105:             _storeRebase(updatedTotalSupply, _profit, _epoc
```

As a consequence, the functionality isn't doing what it was created for.

Recommended Mitigation Steps

Consider calling `_storeRebase()` with `currentTotalSupply`:

```
File: Yieldy.sol
- 105:              _storeRebase(updatedTotalSupply, _profit, _ep
+ 105:              _storeRebase(currentTotalSupply, _profit, _ep
```

toshiSat (Yieldy) confirmed and resolved

[M-17] Staking: rebase() does not rebase according to the status of the current epoch.

Submitted by cccz

In the staking contract, the rebase function can only be called once per epoch.

In the rebase function, the rewards of the current epoch are used in the next epoch, which can cause the rewards to be updated incorrectly and lead to incorrect distribution of user rewards.

```
    function rebase() public {
        // we know about the issues surrounding block.timestamp,
        if (epoch.endTime <= block.timestamp) {
            IYieldy(YIELDY_TOKEN).rebase(epoch.distribute, epoch

            epoch.endTime = epoch.endTime + epoch.duration;
            epoch.timestamp = block.timestamp;
            epoch.number++;

            uint256 balance = contractBalance();
            uint256 staked = IYieldy(YIELDY_TOKEN).totalSupply()

            if (balance <= staked) {
                epoch.distribute = 0;
            } else {
                epoch.distribute = balance - staked;
```

```
            }
        }
    }
```

Proof of Concept

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/Staking.sol#L701-L719

Recommended Mitigation Steps

Put `IYieldy(YIELDY_TOKEN).rebase` after epoch.distribute update

```
    function rebase() public {
        // we know about the issues surrounding block.timestamp,
        if (epoch.endTime <= block.timestamp) {
            uint256 balance = contractBalance();
            uint256 staked = IYieldy(YIELDY_TOKEN).totalSupply()

            if (balance <= staked) {
                epoch.distribute = 0;
            } else {
                epoch.distribute = balance - staked;
            }
            IYieldy(YIELDY_TOKEN).rebase(epoch.distribute, epoch

            epoch.endTime = epoch.endTime + epoch.duration;
            epoch.timestamp = block.timestamp;
            epoch.number++;
        }
    }
```

toshiSat (Yieldy) acknowledged and commented:

> This is how the system is designed.

JasoonS (judge) decreased severity to Medium and commented:

> Changing to Medium. It makes sense that the rebase happens after rewards so that those who enter later don't affect the distribution of rewards before they

joined.

[M-18] Removal of liquidity from the reserve can be griefed

Submitted by lllllll

Users may be unable to withdraw/remove their liquidity from the `LiquidityReserve` if a user decides to grief the contract.

Proof of Concept

This is the only function in this contract that is able to unstake funds, so that they can be withdrawn/removed:

```
File: src/contracts/LiquidityReserve.sol   #1

214       function unstakeAllRewardTokens() public {
215           require(isReserveEnabled, "Not enabled yet");
216           uint256 coolDownAmount = IStaking(stakingContract)
217               .coolDownInfo(address(this))
218               .amount;
219           if (coolDownAmount == 0) {
220               uint256 amount = IERC20Upgradeable(rewardToker
221                   address(this)
222               );
223               if (amount > 0) IStaking(stakingContract).unst
224           }
225       }
```

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/LiquidityReserve.sol#L214-L225

The function requires that the `coolDownAmount` is zero, or else it skips the `unstake()` call. A malicious user can make `coolDownAmount` non-zero by calling `Staking.instantUnstakeReserve()` when the previous reward is claimed, with just a large enough amount to satisfy the transfer of the amount and of the fee, so there is essentially zero left for other users to withdraw. The function calls `LiquidityReserve.instantUnstake()`:

File: src/contracts/Staking.sol #2

```
571       function instantUnstakeReserve(uint256 _amount) exterr
572           require(_amount > 0, "Invalid amount");
573           // prevent unstaking if override due to vulnerabil
574           require(
575               !isUnstakingPaused && !isInstantUnstakingPause
576               "Unstaking is paused"
577           );
578
579           rebase();
580           _retrieveBalanceFromUser(_amount, msg.sender);
581
582           uint256 reserveBalance = IERC20Upgradeable(STAKINC
583               LIQUIDITY_RESERVE
584           );
585
586           require(reserveBalance >= _amount, "Not enough fur
587
588           ILiquidityReserve(LIQUIDITY_RESERVE).instantUnstak
589               _amount,
590               msg.sender
591           );
592       }
```

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/Staking.sol#L571-L592

Which boosts the cooldown amount above zero in its call to `unstakeAllRewardTokens()` and then `IStaking.unstake()`:

File: src/contracts/LiquidityReserve.sol #3

```
188       function instantUnstake(uint256 _amount, address _reci
189           external
190           onlyStakingContract
191       {
192           require(isReserveEnabled, "Not enabled yet");
193           // claim the stakingToken from previous unstakes
194           IStaking(stakingContract).claimWithdraw(address(th
195
```

```
196             uint256 amountMinusFee = _amount - ((_amount * fee
197
198             IERC20Upgradeable(rewardToken).safeTransferFrom(
199                 msg.sender,
200                 address(this),
201                 _amount
202             );
203
204             IERC20Upgradeable(stakingToken).safeTransfer(
205                 _recipient,
206                 amountMinusFee
207             );
208             unstakeAllRewardTokens();
209         }
210
211         /**
212             @notice find balance of reward tokens in contract
213          */
214         function unstakeAllRewardTokens() public {
215             require(isReserveEnabled, "Not enabled yet");
216             uint256 coolDownAmount = IStaking(stakingContract)
217                 .coolDownInfo(address(this))
218                 .amount;
219             if (coolDownAmount == 0) {
220                 uint256 amount = IERC20Upgradeable(rewardToker
221                     address(this)
222                 );
223                 if (amount > 0) IStaking(stakingContract).unst
224             }
225         }
```

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/LiquidityReserve.sol#L188-L225

```
File: src/contracts/Staking.sol   #4

674         function unstake(uint256 _amount, bool _trigger) exter
675             // prevent unstaking if override due to vulnerabil
676             require(!isUnstakingPaused, "Unstaking is paused")
677             if (_trigger) {
678                 rebase();
679             }
```

```
680             _retrieveBalanceFromUser(_amount, msg.sender);
681
682             Claim storage userCoolInfo = coolDownInfo[msg.senc
683
684             // try to claim withdraw if user has withdraws to
685             claimWithdraw(msg.sender);
686
687             coolDownInfo[msg.sender] = Claim({
688                 amount: userCoolInfo.amount + _amount,
689                 credits: userCoolInfo.credits +
690                     IYieldy(YIELDY_TOKEN).creditsForTokenBalar
691                 expiry: epoch.number + coolDownPeriod
692             });
```

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/Staking.sol#L674-L692

If the malicious user is a miner, that miner can make sure that the block where the previous cooldown expires and is claimed, is the same block where the miner griefs by doing an instant unstake of a small amount, preventing larger amounts from going through. Until the miner decides to stop this behavior, funds will be locked in the contract.

Recommended Mitigation Steps

Keep track of submitted amounts during the cooldown, and batch-submit them during the next open window, rather than making it first-come-first-served

Oxean (Yieldy) disputed, disagreed with severity and commented:

> The warden does identify a potential attack, but the assumptions that are being made for it to work are pretty hard to imagine. For one, It would require a miner to always be able to process a specific block in order to continually DOS the contract. This is very infeasible.

> Additionally, if this scenario was to occur, we could pause instant unstaking and wait for the cooldown to expire in order to retrieve funds. The ability to toggle the instant unstake negates the call path the warden has suggested since `Staking.instantUnstakeReserve()` would revert.

> Given all of this, I would put this entire attack vector as super low risk and suggest its downgraded to QA.

JasoonS (judge) decreased severity to Medium and commented:

> I'm going to make this a Medium. While I agree the assumptions are out of the imaginable in reality, it is something that should be looked into for the contracts more seriously than the typical QA.

[M-19] Staking: the rebase function needs to be called before calling the function in the Yieldy contract that uses the rebasingCreditsPerToken variable

Submitted by cccz

In the Yieldy contract, functions such as balanceOf/creditsForTokenBalance/tokenBalanceForCredits/transfer/transferFrom/burn/mint will use the rebasingCreditsPerToken variable, so before calling these functions in the Staking contract, make sure that the rebase of this epoch has occurred. Therefore, the rebase function should also be called in the unstake/claim/claimWithdraw function of the Staking contract.

Proof of Concept

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/Staking.sol#L674-L696

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/Staking.sol#L465-L508

Recommended Mitigation Steps

```
    function claim(address _recipient) public {
        Claim memory info = warmUpInfo[_recipient];
+       rebase();
...
```

```
    function claimWithdraw(address _recipient) public {
        Claim memory info = coolDownInfo[_recipient];
+       rebase();
...
    function unstake(uint256 _amount, bool _trigger) external {
        // prevent unstaking if override due to vulnerabilities
        require(!isUnstakingPaused, "Unstaking is paused");
-        if (_trigger) {
            rebase();
-        }
```

[toshiSat (Yieldy) confirmed and resolved]

[JasoonS (judge) commented]:

> Yes, seems like a logic flaw, makes sense as Medium.

[M-20] User fund lose in addLiquidity() of LiquidityReserve by increasing (totalLockedValue / totalSupply()) to very large number by attacker

Submitted by unforgiven

Function `addLiquidity()` suppose to do add Liquidity for the `staking Token` and receive `lrToken` in exchange. to calculate amount of `IrToken` codes uses this calculation: `amountToMint = (_amount * lrFoxSupply) / totalLockedValue` but it's possible for attacker to manipulate `totalLockedValue` (by sending tokens directly to `LiquidityReserve` address) and make `totalLockedValue/lrFoxSupply` very high in early stage of contract deployment so because of rounding error in calculation of `amountToMint` the users would receive very lower `IrToken` and users funds would be lost and attacker can steal them.

Attacker can perform this attack by sending tokens before even `LiquidityReserve` deployed because the contract address would be predictable and attacker can perform front-run or sandwich attack too.

Also it's possible to perform this attack for `STAKING_TOKEN` with low precision and low price even if `LiquidityReserve` had some balances.

Proof of Concept

This is `addLiquidity()` code in `LiquidityReserve`:

```
    function addLiquidity(uint256 _amount) external {
        require(isReserveEnabled, "Not enabled yet");
        uint256 stakingTokenBalance = IERC20Upgradeable(stakingT
            address(this)
        );
        uint256 rewardTokenBalance = IERC20Upgradeable(rewardTok
            address(this)
        );
        uint256 lrFoxSupply = totalSupply();
        uint256 coolDownAmount = IStaking(stakingContract)
            .coolDownInfo(address(this))
            .amount;
        uint256 totalLockedValue = stakingTokenBalance +
            rewardTokenBalance +
            coolDownAmount;

        uint256 amountToMint = (_amount * lrFoxSupply) / totalLo
        IERC20Upgradeable(stakingToken).safeTransferFrom(
            msg.sender,
            address(this),
            _amount
        );
        _mint(msg.sender, amountToMint);
    }
```

As you can see code uses this calculation: `amountToMint = (_amount * lrFoxSupply) / totalLockedValue;` to find the amount of `IrToken` that is going to mint for user. but attacker can send `stakingToken` or `rewardToken` directly to `LiquidityReserve` address when the there is no liqudity in the contract and make `totalLockedValue` very high. then attacker call `addLiquidity()` and mint some `IrToken` for himself and from then anyone tries to call `addLiquidity()` because of rounding error is going to lose some funds (receives less `IrToken` than he is supposed to)

Tools Used

VIM

Recommended Mitigation Steps

Add more precision when calculating `IrToken` so this attack wouldn't be feasible to perform.

Oxean (Yieldy) disagreed with severity and commented:

> The contract locks a minimum liquidity amount which blocks the feasibility attack for the most part. Please see `enableLiquidityReserve` for the code where the locking occurs.

moose-code (judge) decreased severity to Medium and commented:

> Some good worthwhile ideas from the warden but after reviewing the `enableLiquidityReserve` going to downgrade this to medium. After reading the code and the described attack, its not very clear how the attacker would benefit and bring the contract into this state.

> By sending tokens directly to the contract (expensive) and increasing total totalLockedValue, this will decrease the amount the amountToMint for the user but unclear that this cost is worth it or how an attacker could actually benefit (from what I can see).

> Think its still worth exploring this vector in more depth as its a creative attack. Warrants medium and further investigation.

[M-21] Cannot mint to exactly max supply using `_mint` function

Submitted by Chom, also found by hansfriese and minhquanym

Cannot mint to exactly max supply using `_mint` function.

Proof of Concept

```
    require(_totalSupply < MAX_SUPPLY, "Max supply");
```

if `_totalSupply == MAX_SUPPLY` this assert will be failed and reverted.

But it shouldn't be reverted as `_totalSupply == MAX_SUPPLY` is valid.

Recommended Mitigation Steps

Change to

```
    require(_totalSupply <= MAX_SUPPLY, "Max supply");
```

JasoonS (judge) commented:

> Feels potentially too generous giving this a medium since it isn't clear what the exploit would be, but it is a bug. I'll be generous...

toshiSat (Yieldy) acknowledged and commented:

> Yea we aren't going to implement this one due to nearly every example of rebasing tokens are using this calculation. It will be very unlikely that total supply ever hits max supply, so the risk isn't worth the reward for changing it.

[M-22] MINIMUM_LIQUIDITY checks missing - Bringing Liquidity below required min

Submitted by csanuragjain

Whale who provided most liquidity to the contract can simply use removeLiquidity function and can remove all of his liquidity. This can leave the residual liquidity to be less than MINIMUM_LIQUIDITY which is incorrect.

Proof of Concept

1. Whale A provided initial liquidity plus more liquidity using enableLiquidityReserve and addLiquidity function
2. There are other small liquidity providers as well
3. Now Whale A decides to remove all the liquidity provided
4. This means after liquidity removal the balance liquidity will even drop below MINIMUM_LIQUIDITY which is incorrect

Recommended Mitigation Steps

Add below check

```
    require(
                IERC20Upgradeable(stakingToken).balanceOf(address(th
                    amountToWithdraw,
                "Not enough funds"
            );
```

toshiSat (Yieldy) confirmed and resolved

[M-23] Incorrect withdrawal requested

Submitted by csanuragjain, also found by MiloTruck

`\_requestWithdrawalFromTokemak` function :: Instead of sending amountToRequest for requestWithdrawal, contract is asking _amount for requestWithdrawal. This becomes a problem when balance < _amount and only balance could be withdrawn

Proof of Concept

1. In _requestWithdrawalFromTokemak function, amountToRequest is calculated as

```
    // the only way balance < _amount is when using unstakeAllFromTo
            uint256 amountToRequest = balance < _amount ? balance :
```

2. Now assuming balance < _amount then amountToRequest becomes balance
3. But tokePoolContract.requestWithdrawal is called over _amount instead of amountToRequest which means withdrawal is requested over an extra amount

Recommended Mitigation Steps

Modify Staking.sol#L326 to

```
if (amountToRequest > 0) tokePoolContract.requestWithdrawal(amou
```

[toshiSat (Yieldy) confirmed and resolved]

[M-24] Staking `preSign` could use some basic validations

Submitted by Alex the Entreprenerd

The function `preSign` accepts any `orderUid`.

```
function preSign(bytes calldata orderUid) external onlyOwner
```

Because of how Cowswap works, accepting any `orderUid` can be used as a rug-vector.

This is because the orderData contains a `receiver` which in lack of validation could be any address.

You'd also be signing other parameters such as minOut and how long the order could be filled for, which you may or may not want to validate to give stronger security guarantees to end users.

Recomended Mitigation Steps

I'd recommend adding basic validation for tokenOut, minOut and receiver.

Feel free to check the work we've done at Badger to validate order parameters, giving way stronger guarantees to end users.
https://github.com/GalloDaSballo/fair-

selling/blob/44c0c0629289a0c4ccb3ca971cc5cd665ce5cb82/contracts/CowSwapSeller.sol#L194

Also notice how through the code above we are able to re-construct the `orderUid`, feel free to re-use that code which has been validated by the original Cowswap / GPv2 Developers.

toshiSat (Yieldy) confirmed, resolved and commented:

> Thanks for the functions, I like what you guys did. Our cowswap function is only called using the `onlyOwner` modifier, so I think it's pretty safe, but I agree some validation would be better than none.

[M-25] coolDown & warmUp period do not work when a low _firstEpochEndTime is passed to initialize

Submitted by Lambda

In the constructor of `Staking.sol`, it is not enforced that the `_firstEpochEndTime` is larger than the current `block.timestamp`. If a low value is accidentally passed (or even 0), `rebase` can be called multiple times in sucession, causing the `epoch.number` to increase. Therefore, the coolDown & warmUp period can be circumvented in such a scenario, as `epoch.number >= info.expiry` (in `_isClaimAvailable` and `_isClaimWithdrawAvailable`) will return true after `rebase` caused several increases of `epoch.number`.

Recommended Mitigation Steps

Either require that `_firstEpochEndTime` is larger than `block.timestamp` or set the expiry of the first epoch to `block.timestamp + _epochDuration`.

toshiSat (Yieldy) acknowledged and commented:

> This is something we thought about and will most likely set the period temporarily 1 higher when launching with low initial epoch durations.

[M-26] instantUnstake function can be frontrunned with fee increase

Submitted by sashik_eth

`instantUnstake()` allows user to unstake their stakingToken for a fee paid to the liquidity providers. This fee could be changed up to 100% any moment by admin.

Malicious admin could frontrun users `instantUnstake()` transaction and set `fee` to any value (using `setFee()`) and get all users unstaking asset.

It's even could lead to a situation when non-malicious admin accidentally frontrun unstaking user by increasing fee to a new rate, which user wasn't expected.

```
/**
    @notice sets Fee (in basis points eg. 100 bps = 1%) for
    @param _fee uint - fee in basis points
 */
function setFee(uint256 _fee) external onlyOwner {
    // check range before setting fee
    require(_fee <= BASIS_POINTS, "Out of range");
    fee = _fee;

    emit FeeChanged(_fee);
}
```

Recommended Mitigation Steps

Consider introducing an upper limit for fees so users can know the maximum fess available in protocol and adding timelock to change fee size. This way, frontrunning will be impossible, and users will know which fee they agree to.

toshiSat (Yieldy) acknowledged

JasoonS (judge) commented:

> Checks should be in place for this. Saying the code is upgradeable isn't an excuse for not having sanity checks in admin functions in the code.

> For example script could have a bug that sets this value wrong (for example making it 1e18 times bigger than it should be or something).

[M-27] instantUnstake fee can be avoided

Submitted by skoorch

Users can utilize the `instantUnstake` function without paying the liquidity provider fee using rounding errors in the fee calculation. This attack only allows for a relatively small amount of tokens to be unstaked in each call, so is likely not feasible on mainnet. However, on low-cost L2s and for tokens with a small decimal precision it is likely a feasible workaround.

Proof of Concept

The `instantUnstake` fee is handled by sending the user back `amount - fee`. We can work around the fee by unstaking small amounts (`amount < BASIS_POINTS / fee`) in a loop until reaching the desired amount.

Recommended Mitigation Steps

Avoid using subtraction to calculate the fee as this causes the fee to be rounded down rather than the amount. I'd propose calculating amount less fee using a muldiv operation over (1 - fee). In this case, the fee is effectively rounded up instead of down, so it can never be 0 unless fee is 0. Uniswapv2 uses a similar solution for their LP fee: https://github.com/Uniswap/v2-core/blob/8b82b04a0b9e696c0e83f8b2f00e5d7be6888c79/contracts/UniswapV2Pair.sol#L180-L182

It might look like the following:

```
uint256 amountMinusFee = amount * (BASIS_POINTS - fee) / BASIS_I
```

toshiSat (Yieldy) confirmed and resolved

Low Risk and Non-Critical Issues

For this contest, 70 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **IIIIIII** received the top score from the judge.

The following wardens also submitted reports: BowTiedWardens, defsec, robee, 0x1337, 0xNazgul, hubble, reassor, 0x29A, berndartmueller, GalloDaSballo, joestakey, Lambda, pashov, Picodes, pedr02b2, 0xc0ffEE, csanuragjain, exd0tpy, GimelSec, shung, scaraven, StErMi, zzzitron, elprofesor, unforgiven, 0xf15ers, FudgyDRS, hansfriese, oyc_109, hake, Waze, 0x1f8b, cccz, _Adam, aga7hokakological, Bnke0x0, cryptphi, dipp, fatherOfBlocks, Funen, ladboy233, Limbooo, MiloTruck, Noah3o6, samruna, sikorico, TomJ, 0xNineDec, 0xDjango, ak1, Chom, UnusualTurtle, sseefried, 0xmint, antonttc, delfin454000, ElKu, JC, Kaiziron, kenta, Metatron, mics, PumpkingWok, PwnedNoMore, simon135, Sm4rty, tchkvsky, TrungOre, *and* 0x52.

Low Risk Issues

	Issue	Instances
L-01	Batch-related functions will revert if `removeAddress()` is called	2
L-02	Staking contract's token not verified to be the same token as the staking token	1
L-03	Missing infinite approval functionality	1
L-04	Missing checks that the end time matches the duration	1
L-05	Missing input validations and timelocks	5
L-06	Front-runable initializer	2

Total: 12 instances over 6 issues

[L-01] Batch-related functions will revert if `removeAddress()` is called

`removeAddress()` removes entries from the storage array that holds batch contract addresses, but it doesn't fill in the deleted entries with a replacement value. When

other functions hit these null entries and try to call functions on the zero address, they'll revert, causing the whole function to fail

There are 2 instances of this issue. (For in-depth details on this and all further low and non-critical items with multiple instances, see the warden's full report.)

[L-02] Staking contract's token not verified to be the same token as the staking token

There may be a mismatch between the token that `_stakingContract` is in charge of, and the actual token used by the `LiquidityReserve`. This code should check that they are in fact the same

There is 1 instance of this issue:

```
File: src/contracts/LiquidityReserve.sol   #1

57        function enableLiquidityReserve(address _stakingContrac
58            external
59            onlyOwner
60        {
61            require(!isReserveEnabled, "Already enabled");
62            require(_stakingContract != address(0), "Invalid ac
63
64            uint256 stakingTokenBalance = IERC20Upgradeable(sta
65                msg.sender
66            );
67            // require address has minimum liquidity
68            require(
69                stakingTokenBalance >= MINIMUM_LIQUIDITY,
70                "Not enough staking tokens"
71            );
72:           stakingContract = _stakingContract;
```

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/LiquidityReserve.sol#L57-L72

[L-03] Missing infinite approval functionality

Most other contracts in the repository use `type(uint256).max` to mean infinite approval, rather than a specific approval amount. Not doing the same thing here will mean inconsistent behavior between the components, will mean that approvals will eventually run down to zero, and will mean that there will be hard-to-track-down issues when things eventually start failing

There is 1 instance of this issue:

```
File: src/contracts/Yieldy.sol   #1

210           require(_allowances[_from][msg.sender] >= _value, '
211
212           uint256 newValue = _allowances[_from][msg.sender] -
213           _allowances[_from][msg.sender] = newValue;
214:          emit Approval(_from, msg.sender, newValue);
```

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/Yieldy.sol#L210-L214

[L-04] Missing checks that the end time matches the duration

There is 1 instance of this issue:

```
File: src/contracts/Staking.sol   #1

95                duration: _epochDuration,
96                number: 1,
97                timestamp: block.timestamp, // we know about th
98:               endTime: _firstEpochEndTime,
```

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/Staking.sol#L95-L98

[L-05] Missing input validations and timelocks

The following instances are missing checks for zero addresses and or valid ranges for values. Even if the DAO is the one setting these values, it's important to add sanity checks in case someone does a fat-finger operation that is missed by DAO participants who may not be very technical. There are also no timelocks involved, which should be rectified

There are 5 instances of this issue.

[L-06] Front-runable initializer

There is nothing preventing another account from calling the initializer before the contract owner. In the best case, the owner is forced to waste gas and re-deploy. In the worst case, the owner does not notice that his/her call reverts, and everyone starts using a contract under the control of an attacker

There are 2 instances of this issue.

Non-Critical Issues

	Issue	Instances
N-01	Return values of `approve()` not checked	2
N-02	Misleading variable names	1
N-03	`public` functions not called by the contract should be declared `external` instead	1
N-04	`constant` s should be defined rather than using magic numbers	3
N-05	Use a more recent version of solidity	3
N-06	Typos	10
N-07	NatSpec is incomplete	2
N-08	Event is missing `indexed` fields	12

Total: 34 instances over 8 issues

[N-01] Return values of `approve()` not checked

Not all `IERC20` implementations `revert()` when there's a failure in `approve()`. The function signature has a `boolean` return value and they indicate errors that way instead. By not checking the return value, operations that should have marked as failed, may potentially go through without actually approving anything

There are 2 instances of this issue.

[N-02] Misleading variable names

There is 1 instance of this issue:

```
File: src/contracts/LiquidityReserve.sol   #1

/// @audit code is no longer FOX-specific
112:         uint256 lrFoxSupply = totalSupply();
```

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/LiquidityReserve.sol#L112

[N-03] `public` functions not called by the contract should be declared `external` instead

Contracts are allowed to override their parents' functions and change the visibility from `external` to `public`.

There is 1 instance of this issue:

```
File: src/contracts/Staking.sol   #1

370:     function unstakeAllFromTokemak() public onlyOwner {
```

https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/Staking

.sol#L370

[N-04] `constant`s should be defined rather than using magic numbers

Even assembly can benefit from using readable constants instead of hex/numeric literals

There are 3 instances of this issue.

[N-05] Use a more recent version of solidity

Use a solidity version of at least 0.8.13 to get the ability to use `using for` with a list of free functions

There are 3 instances of this issue.

[N-06] Typos

There are 10 instances of this issue.

[N-07] NatSpec is incomplete

There are 2 instances of this issue.

[N-08] Event is missing `indexed` fields

Each `event` should use three `indexed` fields if there are three or more fields

There are 12 instances of this issue.

moose-code (judge) commented:

> Low risk issues:
> 1 - Agree
> 2 - Agree
> 3 - Agree
> 4 - Agree

> 5 - Agree
> 6 -Agree

> Informational:
> 1- Agree
> 2 -Agree
> 3 - Agree
> 4 - Strongly agree, this is very helpful.
> 5 - Agree
> 6 - Agree
> 7 - Agree
> 8 - Agree

Gas Optimizations

For this contest, 70 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **BowTiedWardens** received the top score from the judge.

The following wardens also submitted reports: lllllll, MiloTruck, _Adam, ajtra, Fabble, GalloDaSballo, 0xkatana, 0xKitsune, defsec, joestakey, reassor, TomJ, 0x29A, antonttc, Bnke0x0, fatherOfBlocks, FudgyDRS, minhquanym, RedOneN, Tomio, PwnedNoMore, 0x1f8b, 0xf15ers, 0xNazgul, hansfriese, kenta, ladboy233, Lambda, m_Rassska, Nyamcil, pashov, Randyyy, scaraven, Sm4rty, Waze, Noah3o6, c3phas, delfin454000, ElKu, GimelSec, JC, Kaiziron, oyc_109, simon135, mics, 0xmint, 8olidity, asutorufos, Fitraldys, Funen, Picodes, robee, saian, sashik_eth, TrungOre, UnusualTurtle, 0v3rf10w, ACai, bardamu, Chom, exd0tpy, sach1r0, StErMi, Limbooo, s3cunda, sikorico, slywaters, aga7hokakological, *and* ignacio.

Table of Contents

- G-06. `LiquidityReserveStorage` : Tightly pack storage variables
- G-07. `YieldyStorage` : Tightly pack storage variables
- G-08. Duplicated conditions should be refactored to a modifier or function to save deployment costs
- G-09. A modifier used only once and not being inherited should be inlined to save gas
- G-10. Pre-Solidity `0.8.13` : `> 0` is less efficient than `!= 0` for unsigned integers (with proof)
- G-11. `>=` is cheaper than `>` (and `<=` cheaper than `<`)
- G-12. Splitting `require()` statements that use `&&` saves gas
- G-13. Using private rather than public for constants saves gas
- G-14. Amounts should be checked for 0 before calling a transfer
- G-15. `++i` costs less gas compared to `i++` or `i += 1` (same for `--i` vs `i--` or `i -= 1`)
- G-16. Public functions to external
- G-17. It costs more gas to initialize variables with their default value than letting the default value be applied
- G-18. Upgrade pragma
- G-19. Use Custom Errors instead of Revert Strings to save Gas
- G-20. Functions guaranteed to revert when called by normal users can be marked `payable`
- G-21. Use `1000` rather than exponentiation `10**3`

[G-01] Duplicated external function call

External function calls are expensive. This one seems like a copy-paste error:

- Staking.sol#initialize()

```
File: Staking.sol
84:         IERC20Upgradeable(YIELDY_TOKEN).approve(
85:             LIQUIDITY_RESERVE,
86:             type(uint256).max
```

```
87:         );
- 88:         IERC20Upgradeable(YIELDY_TOKEN).approve(
- 89:             LIQUIDITY_RESERVE,
- 90:             type(uint256).max
- 91:         );
```

[G-02] Wrong use of the `memory` keyword for a Struct

When copying a state struct in memory, there are as many SLOADs and MSTOREs as there are slots. When reading the whole struct multiple times is not needed, it's better to actually only read the relevant field(s). When only some of the fields are read several times, these particular values should be cached instead of the whole state struct.

Here, the `storage` keyword should be used instead of `memory`:

- Saving 1 STORE and 1 MSTORE: https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/Staking.sol#L259-L266

```
File: Staking.sol
259:     function _isClaimAvailable(address _recipient)
260:         internal
261:         view
262:         returns (bool)
263:     {
- 264:         Claim memory info = warmUpInfo[_recipient]; //@au
+ 264:         Claim storage info = warmUpInfo[_recipient]; //@a
+ 264:         uint256 _expiry = info.expiry; //@audit 1 SLOAD,
- 265:         return epoch.number >= info.expiry && info.expiry
+ 265:         return epoch.number >= _expiry && _expiry != 0;
266:     }
```

- Saving 2 MSTOREs and 2 MLOADs: https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/Staking.sol#L281-L289

```
File: Staking.sol
- 281:         RequestedWithdrawalInfo memory requestedWithdrawa
```

```
+ 281:         RequestedWithdrawalInfo storage requestedWithdraw
282:           .requestedWithdrawals(address(this));
283:       uint256 currentCycleIndex = tokeManager.getCurrentC
284:       return
285:           epoch.number >= info.expiry &&
286:           info.expiry != 0 &&
287:           info.amount != 0 &&
288:           ((requestedWithdrawals.minCycle <= currentCycle
289:               requestedWithdrawals.amount + withdrawalAmo
```

- Saving 1 SLOAD and 1 MSTORE: https://github.com/code-423n4/2022-06-yieldy/blob/524f3b83522125fb7d4677fa7a7e5ba5a2c0fe67/src/contracts/Staking.sol#L466-L474

```
File: Staking.sol
465:     function claim(address _recipient) public {
- 466:         Claim memory info = warmUpInfo[_recipient]; //@au
+ 466:         Claim storage info = warmUpInfo[_recipient]; //@a
+ 466:         uint256 _credits = info.expiry; //@audit 1 SLOAD,
467:       if (_isClaimAvailable(_recipient)) {
468:           delete warmUpInfo[_recipient];
469:
- 470:           if (info.credits > 0) {
+ 470:           if (_credits > 0) {
471:             IYieldy(YIELDY_TOKEN).transfer(
472:                 _recipient,
- 473:                 IYieldy(YIELDY_TOKEN).tokenBalanceFor
+ 473:                 IYieldy(YIELDY_TOKEN).tokenBalanceFor
474:             );
475:           }
476:       }
477:     }
```

[G-03] Caching storage values in memory

The code can be optimized by minimizing the number of SLOADs.

SLOADs are expensive (100 gas after the 1st one) compared to MLOADs/MSTOREs (3 gas each). Storage values read multiple times should instead be cached in memory the first time (costing 1 SLOAD) and then read from this cache to avoid multiple SLOADs.

- contracts[i]

```
File: BatchRequests.sol
18:                 contracts[i] != address(0) &&
19:                 IStaking(contracts[i]).canBatchTransactions(
20:             ) {
21:                 IStaking(contracts[i]).sendWithdrawalRequest
```

- contracts[i]

```
File: BatchRequests.sol
37:             bool canBatch = IStaking(contracts[i]).canBatchT
38:             batch[i] = Batch(contracts[i], canBatch);
```

- contracts[_index]

```
File: BatchRequests.sol
56:             contracts[_index],
57:             IStaking(contracts[_index]).canBatchTransactions
```

- stakingToken

```
File: LiquidityReserve.sol
64:         uint256 stakingTokenBalance = IERC20Upgradeable(stak
...
75:         IERC20Upgradeable(stakingToken).safeTransferFrom(
```

- stakingContract

```
File: LiquidityReserve.sol
72:         stakingContract = _stakingContract;
...
81:         IERC20Upgradeable(rewardToken).approve(
- 82:             stakingContract,
+ 82:             _stakingContract,
```

- stakingToken

```
File: LiquidityReserve.sol
106:         uint256 stakingTokenBalance = IERC20Upgradeable(sta
...
121:         IERC20Upgradeable(stakingToken).safeTransferFrom(
```

- stakingToken

```
File: LiquidityReserve.sol
171:             IERC20Upgradeable(stakingToken).balanceOf(addre
...
177:         IERC20Upgradeable(stakingToken).safeTransfer(
```

- affiliateFee and FEE_ADDRESS

```
File: Staking.sol
129:     function _sendAffiliateFee(uint256 _amount) internal {
130:         if (affiliateFee != 0 && FEE_ADDRESS != address(0))
131:             uint256 feeAmount = (_amount * affiliateFee) /
132:             IERC20Upgradeable(TOKE_TOKEN).safeTransfer(FEE_
133:         }
134:     }
```

- TOKE_TOKEN

```
File: Staking.sol
144:         uint256 totalTokeAmount = IERC20Upgradeable(TOKE_TO
145:             address(this)
146:         );
147:         IERC20Upgradeable(TOKE_TOKEN).safeTransfer(
```

- requestWithdrawalAmount

```
File: Staking.sol
392:             if (requestWithdrawalAmount > 0) {
```

```
393:                 _requestWithdrawalFromTokemak(requestWithdr
394:             }
```

- YIELDY_TOKEN

```
File: Staking.sol
519:         uint256 walletBalance = IERC20Upgradeable(YIELDY_TC
...
522:         uint256 warmUpBalance = IYieldy(YIELDY_TOKEN).toker
...
545:                     IYieldy(YIELDY_TOKEN).creditsForTokenBa
546:                 uint256 remainingAmount = IYieldy(YIELDY_TC
...
559:             IERC20Upgradeable(YIELDY_TOKEN).safeTransferFrc
```

- LIQUIDITY_RESERVE

```
File: Staking.sol
583:             LIQUIDITY_RESERVE
...
588:         ILiquidityReserve(LIQUIDITY_RESERVE).instantUnstake
```

- CURVE_POOL / STAKING_TOKEN / TOKE_POOL

```
File: Staking.sol
633:         if (CURVE_POOL != address(0)) {
634:             address address0 = ICurvePool(CURVE_POOL).coins
635:             address address1 = ICurvePool(CURVE_POOL).coins
636:             int128 from = 0;
637:             int128 to = 0;
638:
639:             if (TOKE_POOL == address0 && STAKING_TOKEN == a
640:                 to = 1;
641:             } else if (TOKE_POOL == address1 && STAKING_TOK
642:                 from = 1;
643:             }
644:             require(from == 1 || to == 1, "Invalid Curve Pc
645:
646:             curvePoolFrom = from;
```

```
647:             curvePoolTo = to;
648:
649:             emit LogSetCurvePool(CURVE_POOL, curvePoolTo, c
650:         }
```

- `_totalSupply`

```
File: Yieldy.sol
82:         uint256 currentTotalSupply = _totalSupply;
- 83:         require(_totalSupply > 0, "Can't rebase if not cir
+ 83:         require(currentTotalSupply > 0, "Can't rebase if r
```

- `_totalSupply`

```
File: Yieldy.sol
122:                 totalStakedAfter: _totalSupply,
...
129:         emit LogSupply(_epoch, block.timestamp, _totalSuppl
```

[G-04] Avoid emitting a storage variable when a memory value is available

When they are the same, consider emitting the memory value instead of the storage value:

```
File: Staking.sol
646:             curvePoolFrom = from;
647:             curvePoolTo = to;
648:
- 649:             emit LogSetCurvePool(CURVE_POOL, curvePoolTo,
+ 649:             emit LogSetCurvePool(CURVE_POOL, to, from);
```

[G-05] Unchecking arithmetics operations that can't underflow/overflow

Solidity version 0.8+ comes with implicit overflow and underflow checks on unsigned integers. When an overflow or an underflow isn't possible (as an example, when a comparison is made before the arithmetic operation), some gas can be saved by using an `unchecked` block: https://docs.soliditylang.org/en/v0.8.10/control-structures.html#checked-or-unchecked-arithmetic

Consider wrapping with an `unchecked` block here:

```
File: Yieldy.sol
210:          require(_allowances[_from][msg.sender] >= _value, '
211:
- 212:          uint256 newValue = _allowances[_from][msg.sender]
+ 212:          unchecked { uint256 newValue = _allowances[_from]
```

```
File: Yieldy.sol
190:          require(creditAmount <= creditBalances[msg.sender],
191:
- 192:          creditBalances[msg.sender] = creditBalances[msg.s
+ 192:          unchecked { creditBalances[msg.sender] = creditBa
```

```
File: Staking.sol
713:              if (balance <= staked) {
714:                  epoch.distribute = 0;
715:              } else {
- 716:                  epoch.distribute = balance - staked;
+ 716:                  unchecked { epoch.distribute = balance -
717:              }
```

```
File: LiquidityReserve.sol
- 196:          uint256 amountMinusFee = _amount - ((_amount * fe
+ 196:          unchecked { uint256 amountMinusFee = _amount - (
```

[G-06] `LiquidityReserveStorage` : Tightly pack storage variables

Here, variables can be tightly packed from to save 1 SLOT:

```
File: LiquidityReserveStorage.sol
04: contract LiquidityReserveStorage {
05:     address public stakingToken; // staking token address
06:     address public rewardToken; // reward token address
07:     address public stakingContract; // staking contract addr
+ 8:      bool public isReserveEnabled; // ensures we are fully i
08:     uint256 public fee; // fee for instant unstaking
09:     uint256 public constant MINIMUM_LIQUIDITY = 10**3; // lc
10:     uint256 public constant BASIS_POINTS = 10000; // 100% ir
- 11:     bool public isReserveEnabled; // ensures we are fully
12: }
```

[G-07] `YieldyStorage` : Tightly pack storage variables

Here, variables can be tightly packed from to save 1 SLOT:

```
File: YieldyStorage.sol
06: contract YieldyStorage {
07:     address public stakingContract;
+ 08:     uint8 internal decimal;
08:     Rebase[] public rebases;
09:     uint256 public index;
10:     bytes32 public constant ADMIN_ROLE = keccak256("ADMIN");
11:     bytes32 public constant MINTER_BURNER_ROLE =
12:         keccak256("MINTER_BURNER_ROLE");
13:     bytes32 public constant REBASE_ROLE = keccak256("REBASE_
14:
15:     uint256 internal WAD;
16:     uint256 internal constant MAX_UINT256 = ~uint256(0);
17:
18:     uint256 internal constant MAX_SUPPLY = ~uint128(0); //
19:     uint256 public rebasingCreditsPerToken; // gonsPerFragme
20:     uint256 public rebasingCredits; // total credits in syst
21:     mapping(address => uint256) public creditBalances; // go
22:
- 23:     uint8 internal decimal; //@audit can be tightly packed
24: }
```

[G-08] Duplicated conditions should be refactored to a modifier or function to save deployment costs

```
LiquidityReserve.sol:105:         require(isReserveEnabled, "Not
LiquidityReserve.sol:192:         require(isReserveEnabled, "Not
LiquidityReserve.sol:215:         require(isReserveEnabled, "Not
```

[G-09] A modifier used only once and not being inherited should be inlined to save gas

Affected code:

```
src/contracts/LiquidityReserve.sol:
   24:     modifier onlyStakingContract() {

  188     function instantUnstake(uint256 _amount, address _rec
  189         external
  190:        onlyStakingContract
  191     {
```

[G-10] Pre-Solidity `0.8.13 : > 0` is less efficient than `!= 0` for unsigned integers (with proof)

Up until Solidity `0.8.13 : != 0` costs less gas compared to `> 0` for unsigned integers in `require` statements with the optimizer enabled (6 gas)

Proof: While it may seem that `> 0` is cheaper than `!=`, this is only true without the optimizer enabled and outside a require statement. If you enable the optimizer AND you're in a `require` statement, this will save gas. You can see this tweet for more proofs: https://twitter.com/gzeon/status/1485428085885640706

Consider changing `> 0` with `!= 0` here:

```
Staking.sol:118:         require(_recipient.amount > 0, "Must ent
Staking.sol:410:         require(_amount > 0, "Must have valid am
Staking.sol:572:         require(_amount > 0, "Invalid amount");
```

```
Staking.sol:604:         require(_amount > 0, "Invalid amount");
Yieldy.sol:83:         require(_totalSupply > 0, "Can't rebase if
Yieldy.sol:96:           require(rebasingCreditsPerToken > 0, '
```

Also, please enable the Optimizer.

[G-11] `>=` is cheaper than `>` (and `<=` cheaper than `<`)

Strict inequalities (`>`) are more expensive than non-strict ones (`>=`). This is due to some supplementary checks (ISZERO, 3 gas). This also holds true between `<=` and `<`.

Consider replacing strict inequalities with non-strict ones to save some gas here:

```
Staking.sol:324:         uint256 amountToRequest = balance < _amo
```

[G-12] Splitting `require()` statements that use `&&` saves gas

If you're using the Optimizer at 200, instead of using the `&&` operator in a single require statement to check multiple conditions, Consider using multiple require statements with 1 condition per require statement:

```
LiquidityReserve.sol:45:             _stakingToken != address(0)
Migration.sol:21:           _oldContract != address(0) && _newC
Staking.sol:55:           _stakingToken != address(0) &&
Staking.sol:56:               _yieldyToken != address(0) &&
Staking.sol:57:               _tokeToken != address(0) &&
Staking.sol:58:               _tokePool != address(0) &&
Staking.sol:59:               _tokeManager != address(0) &&
Staking.sol:60:               _tokeReward != address(0) &&
Staking.sol:575:           !isUnstakingPaused && !isInstantUnst
Staking.sol:606:           CURVE_POOL != address(0) &&
Staking.sol:612:           !isUnstakingPaused && !isInstantUnst
```

Please, note that this might not hold true at a higher number of runs for the Optimizer (10k). However, it indeed is true at 200.

[G-13] Using private rather than public for constants saves gas

If needed, the value can be read from the verified contract source code. Savings are due to the compiler not having to create non-payable getter functions for deployment calldata, and not adding another entry to the method ID table

```
LiquidityReserveStorage.sol:9:    uint256 public constant MINIMU
LiquidityReserveStorage.sol:10:    uint256 public constant BASIS
StakingStorage.sol:39:    uint256 public constant BASIS_POINTS =
```

[G-14] Amounts should be checked for 0 before calling a transfer

Checking non-zero transfer values can avoid an expensive external call and save gas.

Consider adding a non-zero-value check here:

```
LiquidityReserve.sol:121:        IERC20Upgradeable(stakingToken)
LiquidityReserve.sol:177:        IERC20Upgradeable(stakingToken)
LiquidityReserve.sol:198:        IERC20Upgradeable(rewardToken).
LiquidityReserve.sol:204:        IERC20Upgradeable(stakingToken)
```

[G-15] `++i` costs less gas compared to `i++` or `i += 1` (same for `--i` vs `i--` or `i -= 1`)

Pre-increments and pre-decrements are cheaper.

For a `uint256 i` variable, the following is true with the Optimizer enabled at 10k:

Increment:

- `i += 1` is the most expensive form
- `i++` costs 6 gas less than `i += 1`

- `++i` costs 5 gas less than `i++` (11 gas less than `i += 1`)

Decrement:

- `i -= 1` is the most expensive form
- `i--` costs 11 gas less than `i -= 1`
- `--i` costs 5 gas less than `i--` (16 gas less than `i -= 1`)

Note that post-increments (or post-decrements) return the old value before incrementing or decrementing, hence the name *post-increment*:

```
uint i = 1;
uint j = 2;
require(j == i++, "This will be false as i is incremented after
```

However, pre-increments (or pre-decrements) return the new value:

```
uint i = 1;
uint j = 2;
require(j == ++i, "This will be true as i is incremented before
```

In the pre-increment case, the compiler has to create a temporary variable (when used) for returning `1` instead of `2`.

Affected code:

```
Staking.sol:708:                epoch.number++;
```

Consider using pre-increments and pre-decrements where they are relevant (meaning: not where post-increments/decrements logic are relevant).

[G-16] Public functions to external

An external call cost is less expensive than one of a public function. The following functions could be set external to save gas and improve code quality (extracted from

Slither).

```
Staking.sol:370:    function unstakeAllFromTokemak() public only
```

[G-17] It costs more gas to initialize variables with their default value than letting the default value be applied

If a variable is not set/initialized, it is assumed to have the default value (`0` for `uint` , `false` for `bool` , `address(0)` for address...). Explicitly initializing it with its default value is an anti-pattern and wastes gas.

As an example: `for (uint256 i = 0; i < numIterations; ++i) {` should be replaced with `for (uint256 i; i < numIterations; ++i) {`

Affected code:

```
Staking.sol:636:            int128 from = 0;
Staking.sol:637:            int128 to = 0;
```

Consider removing explicit initializations for default values.

[G-18] Upgrade pragma

Using newer compiler versions and the optimizer give gas optimizations. Also, additional safety checks are available for free.

The advantages here are:

- **Contract existence checks** (>= 0.8.10): external calls skip contract existence checks if the external call has a return value

Consider upgrading here :

```
BatchRequests.sol:2:pragma solidity 0.8.9;
LiquidityReserve.sol:2:pragma solidity 0.8.9;
LiquidityReserveStorage.sol:2:pragma solidity 0.8.9;
```

```
Migration.sol:2:pragma solidity 0.8.9;
Staking.sol:2:pragma solidity 0.8.9;
StakingStorage.sol:2:pragma solidity 0.8.9;
Yieldy.sol:2:pragma solidity 0.8.9;
YieldyStorage.sol:2:pragma solidity 0.8.9;
```

[G-19] Use Custom Errors instead of Revert Strings to save Gas

Solidity 0.8.4 introduced custom errors. They are more gas efficient than revert strings, when it comes to deploy cost as well as runtime cost when the revert condition is met. Use custom errors instead of revert strings for gas savings.

Custom errors from Solidity 0.8.4 are cheaper than revert strings (cheaper deployment cost and runtime cost when the revert condition is met)

Source: https://blog.soliditylang.org/2021/04/21/custom-errors/:

> Starting from Solidity v0.8.4, there is a convenient and gas-efficient way to explain to users why an operation failed through the use of custom errors. Until now, you could already use strings to give more information about failures (e.g., `revert("Insufficient funds.");`), but they are rather expensive, especially when it comes to deploy cost, and it is difficult to use dynamic information in them.

Custom errors are defined using the `error` statement, which can be used inside and outside of contracts (including interfaces and libraries).

Consider replacing all revert strings with custom errors in the solution.

```
LiquidityReserve.sol:25:        require(msg.sender == stakingCor
LiquidityReserve.sol:44:        require(
LiquidityReserve.sol:61:        require(!isReserveEnabled, "Alre
LiquidityReserve.sol:62:        require(_stakingContract != addr
LiquidityReserve.sol:68:        require(
LiquidityReserve.sol:94:        require(_fee <= BASIS_POINTS, "C
LiquidityReserve.sol:105:        require(isReserveEnabled, "Not
LiquidityReserve.sol:163:        require(_amount <= balanceOf(ms
LiquidityReserve.sol:170:        require(
```

```
LiquidityReserve.sol:192:        require(isReserveEnabled, "Not
LiquidityReserve.sol:215:        require(isReserveEnabled, "Not
Migration.sol:20:        require(
Staking.sol:54:        require(
Staking.sol:118:        require(_recipient.amount > 0, "Must ent
Staking.sol:143:        require(_claimAddress != address(0), "Ir
Staking.sol:408:        require(!isStakingPaused, "Staking is pa
Staking.sol:410:        require(_amount > 0, "Must have valid am
Staking.sol:527:        require(
Staking.sol:572:        require(_amount > 0, "Invalid amount");
Staking.sol:574:        require(
Staking.sol:586:        require(reserveBalance >= _amount, "Not
Staking.sol:604:        require(_amount > 0, "Invalid amount");
Staking.sol:605:        require(
Staking.sol:611:        require(
Staking.sol:644:            require(from == 1 || to == 1, "Inval
Staking.sol:676:        require(!isUnstakingPaused, "Unstaking i
Yieldy.sol:58:        require(stakingContract == address(0), "Al
Yieldy.sol:59:        require(_stakingContract != address(0), "I
Yieldy.sol:83:        require(_totalSupply > 0, "Can't rebase if
Yieldy.sol:96:            require(rebasingCreditsPerToken > 0, '
Yieldy.sol:187:        require(_to != address(0), "Invalid addre
Yieldy.sol:190:        require(creditAmount <= creditBalances[ms
Yieldy.sol:210:        require(_allowances[_from][msg.sender] >=
Yieldy.sol:249:        require(_address != address(0), "Mint to
Yieldy.sol:257:        require(_totalSupply < MAX_SUPPLY, "Max s
Yieldy.sol:279:        require(_address != address(0), "Burn fro
Yieldy.sol:286:        require(currentCredits >= creditAmount, '
```

[G-20] Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided.

```
BatchRequests.sol:81:    function addAddress(address _address) e
BatchRequests.sol:89:    function removeAddress(address _address
LiquidityReserve.sol:92:    function setFee(uint256 _fee) exter
Staking.sol:141:    function transferToke(address _claimAddress)
Staking.sol:157:    function setCurvePool(address _curvePool) ex
```

```
Staking.sol:167:    function setAffiliateFee(uint256 _affiliateF
Staking.sol:177:    function setAffiliateAddress(address _affili
Staking.sol:187:    function shouldPauseStaking(bool _shouldPaus
Staking.sol:197:    function shouldPauseUnstaking(bool _shouldPa
Staking.sol:207:    function shouldPauseInstantUnstaking(bool _s
Staking.sol:217:    function setEpochDuration(uint256 duration)
Staking.sol:226:    function setWarmUpPeriod(uint256 _vestingPer
Staking.sol:235:    function setCoolDownPeriod(uint256 _vestingP
Staking.sol:370:    function unstakeAllFromTokemak() public only
Staking.sol:769:    function preSign(bytes calldata orderUid) ex
```

[G-21] Use `1000` rather than exponentiation `10**3`

`1000` is readable enough and the cost of the exponentiation operation would be saved here:

```
+ LiquidityReserveStorage.sol:9:    uint256 public constant MINI
- LiquidityReserveStorage.sol:9:    uint256 public constant MINI
```

moose-code (judge) commented:

> Excellent.

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