JPEG'd contest
Findings & Analysis Report

2022-06-10

Table of contents

- G-12 `FungibleAssetVaultForDAO.withdraw()` : L196 should be unchecked due to L194
- G-13 `FungibleAssetVaultForDAO.withdraw()` : `collateralAmount` should get cached
- G-14 `NFTVault._normalizeAggregatorAnswer()` : return statement should be unchecked
- G-15 `NFTVault._calculateAdditionalInterest()` : `totalDebtAmount` should get cached
- G-16 `NFTVault.sol` : `struct PositionPreview` can be tightly packed to save 1 storage slot
- G-17 `NFTVault.showPosition()` : L659 should be unchecked due to L649
- G-18 `NFTVault.showPosition()` : `positions[_nftIndex].liquidatedAt` should get cached
- G-19 `NFTVault.showPosition()` : Help the optimizer by saving a storage variable's reference instead of repeatedly fetching it (`positions[_nftIndex]`)
- G-20 `NFTVault.borrow()` : `totalDebtPortion` should get cached
- G-21 `NFTVault.repay()` : L781 should be unchecked due to ternary operator
- G-22 `NFTVault.repay()` : `totalDebtPortion` and `totalDebtAmount` should get cached
- G-23 `Controller.setStrategy()` : boolean comparison L87
- G-24 `StrategyPUSDConvex.balanceOfJPEG()` : `jpeg` should get cached
- G-25 `StrategyPUSDConvex.balanceOfJPEG()` : use a `return` statement instead of `break`
- G-26 `StrategyPUSDConvex.withdraw()` : L281 should be unchecked due to L279
- G-27 `StrategyPUSDConvex.harvest()` : L362 should be unchecked due to L359-L360 and how performanceFee is set L183
- G-28 `yVault.earn()` : `token` and `controller` should get cached

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 JPEG'd smart contract system written in Solidity. The audit contest took place between April 7—April 13 2022.

Wardens

66 Wardens contributed reports to the JPEG'd contest:

1. cmichel
2. hickuphh3
3. rayn
4. WatchPug
5. Dravee
6. TrungOre
7. berndartmueller
8. Kenshin
9. minhquanym
10. AuditsAreUS
11. 0xDjango
12. IllIllI
13. cccz
14. Jujic
15. hyh
16. pedroais
17. Kthere
18. Foundation (HardlyDifficult and batu)
19. 0x1f8b
20. PPrieditis
21. robee
22. Meta0xNull
23. rfa
24. 0xkatana
25. TerrierLover
26. Cr4ckM3

27. ilan
28. Funen
29. JMukesh
30. catchup
31. kenta
32. kebabsec (okkothejawa and FlameHorizon)
33. ellahi
34. Hawkeye (0xwags and 0xmint)
35. Cityscape
36. horsefacts
37. delfin454000
38. Picodes
39. smiling_heretic
40. Wayne
41. Ruhum
42. reassor
43. hubble (ksk2345 and shri4net)
44. saian
45. JC
46. dy
47. pauliax
48. samruna
49. jayjonah8
50. Tomio
51. 0xNazgul
52. 0v3rf10w
53. securerodd
54. FSchmoede
55. slywaters

56. rokinot
57. nahnah
58. sorrynotsorry
59. dirk_y
60. defsec
61. joshie
62. peritoflores

This contest was judged by LSDan.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 20 unique vulnerabilities. Of these vulnerabilities, 9 received a risk rating in the category of HIGH severity and 11 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 42 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 38 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 JPEG'd contest repository, and is composed of 14 smart contracts written in the Solidity programming language and includes 3,118 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 (9)

[H-01] yVault: First depositor can break minting of shares

Submitted by hickuphh3, also found by 0xDjango, berndartmueller, cmichel, hyh, and WatchPug

yVault.sol#L148-L153

The attack vector and impact is the same as TOB-YEARN-003, where users may not receive shares in exchange for their deposits if the total asset amount has been manipulated through a large “donation”.

Proof of Concept

- Attacker deposits 1 wei to mint 1 share
- Attacker transfers exorbitant amount to the `StrategyPUSDConvex` contract to greatly inflate the share’s price. Note that the strategy deposits its entire balance into Convex when its `deposit()` function is called.
- Subsequent depositors instead have to deposit an equivalent sum to avoid minting 0 shares. Otherwise, their deposits accrue to the attacker who holds the only share.

Insert this test into `yVault.ts`.

```
  it.only("will cause 0 share issuance", async () => {
    // mint 10k + 1 wei tokens to user1
    // mint 10k tokens to owner
    let depositAmount = units(10_000);
    await token.mint(user1.address, depositAmount.add(1));
    await token.mint(owner.address, depositAmount);
    // token approval to yVault
    await token.connect(user1).approve(yVault.address, 1);
    await token.connect(owner).approve(yVault.address, depositAmou

    // 1. user1 mints 1 wei = 1 share
    await yVault.connect(user1).deposit(1);

    // 2. do huge transfer of 10k to strategy
    // to greatly inflate share price (1 share = 10k + 1 wei)
    await token.connect(user1).transfer(strategy.address, depositA

    // 3. owner deposits 10k
    await yVault.connect(owner).deposit(depositAmount);
    // receives 0 shares in return
    expect(await yVault.balanceOf(owner.address)).to.equal(0);

    // user1 withdraws both his and owner's deposits
    // total amt: 20k + 1 wei
    await expect(() => yVault.connect(user1).withdrawAll())
      .to.changeTokenBalance(token, user1, depositAmount.mul(2).ac
  });
```

Recommended Mitigation Steps

- Uniswap V2 solved this problem by sending the first 1000 LP tokens to the zero address. The same can be done in this case i.e. when `totalSupply() == 0`, send the first min liquidity LP tokens to the zero address to enable share dilution.
- Ensure the number of shares to be minted is non-zero: `require(_shares != 0, "zero shares minted");`

spaghettieth (JPEG'd) confirmed, but disagreed with High severity

spaghettieth (JPEG'd) resolved and commented:

> Fixed in jpegd/core#16.

[H-02] Existing user's locked JPEG could be overwritten by new user, causing permanent loss of JPEG funds

Submitted by hickuphh3, also found by 0x1f8b, AuditsAreUS, Foundation, Kthere, Meta0xNull, rayn, and WatchPug

NFTVault.sol#L375
JPEGLock.sol#L54-L62

A user's JPEG lock schedule can be overwritten by another user's if he (the other user) submits and finalizes a proposal to change the same NFT index's value.

The existing user will be unable to withdraw his locked JPEGs, resulting in permanent lock up of JPEG in the locker contract.

Proof of Concept

1. `user` successfully proposes and finalizes a proposal to change his NFT's collateral value
2. Another user (`owner`) does the same for the same NFT index
3. `user` will be unable to withdraw his locked JPEG because schedule has been overwritten

Insert this test case into `NFTVault.ts`.

```
  it.only("will overwrite existing user's JPEG lock schedule", asy
    // 0. setup
    const index = 7000;
    await erc721.mint(user.address, index);
    await nftVault
      .connect(dao)
      .setPendingNFTValueETH(index, units(50));
    await jpeg.transfer(user.address, units(150000));
    await jpeg.connect(user).approve(locker.address, units(500000)
    await jpeg.connect(owner).approve(locker.address, units(50000(

    // 1. user has JPEG locked for finalization
    await nftVault.connect(user).finalizePendingNFTValueETH(index)
    // 2. owner submit proposal to further increase NFT value
    await nftVault
      .connect(dao)
      .setPendingNFTValueETH(index, units(100));

    // 3. owner finalizes, has JPEG locked
    await nftVault.connect(owner).finalizePendingNFTValueETH(index

    // user schedule has been overwritten
    let schedule = await locker.positions(index);
    expect(schedule.owner).to.equal(owner.address);

    // user tries to unstake
    // wont be able to because schedule was overwritten
    await timeTravel(days(366));
    await expect(locker.connect(user).unlock(index)).to.be.reverte
  });
```

Recommended Mitigation Steps

1. Release the tokens of the existing schedule. Simple and elegant.

```
  // in JPEGLock#lockFor()
  LockPosition memory existingPosition = positions[_nftIndex];
  if (existingPosition.owner != address(0)) {
    // release jpegs to existing owner
    jpeg.safeTransfer(existingPosition.owner, existingPosition.loc
  }
```

2. Revert in `finalizePendingNFTValueETH()` there is an existing lock schedule. This is less desirable IMO, as there is a use-case for increasing / decreasing the NFT value.

spaghettieth (JPEG'd) confirmed

spaghettieth (JPEG'd) resolved and commented:

> Fixed in jpegd/core#3.

[H-03] Update initializer modifier to prevent reentrancy during initialization

Submitted by Dravee

package.json#L18-L19

The solution uses:

```
    "@openzeppelin/contracts": "^4.0.0",
    "@openzeppelin/contracts-upgradeable": "^4.3.2",
```

These dependencies have a known high severity vulnerability:

- https://security.snyk.io/vuln/SNYK-JS-OPENZEPPELINCONTRACTSUPGRADEABLE-2320177
- https://snyk.io/test/npm/@openzeppelin/contracts-upgradeable/4.3.2#SNYK-JS-OPENZEPPELINCONTRACTSUPGRADEABLE-2320177
- https://snyk.io/test/npm/@openzeppelin/contracts/4.0.0#SNYK-JS-OPENZEPPELINCONTRACTS-2320176

Which makes these contracts vulnerable:

```
contracts/helpers/CryptoPunksHelper.sol:
  19:     function initialize(address punksAddress) external ini

contracts/helpers/EtherRocksHelper.sol:
  19:     function initialize(address rocksAddress) external ini

contracts/staking/JPEGStaking.sol:
  21:     function initialize(IERC20Upgradeable _jpeg) external

contracts/vaults/FungibleAssetVaultForDAO.sol:
  71:     ) external initializer {

contracts/vaults/NFTVault.sol:
  149:     ) external initializer {
```

Recommended Mitigation Steps

Upgrade `@openzeppelin/contracts` and `@openzeppelin/contracts-upgradeable` to version 4.4.1 or higher.

spaghettieth (JPEG'd) confirmed, but disagreed with High severity

spaghettieth (JPEG'd) resolved and commented:

> Fixed in jpegd/core#11.

[H-04] Reentrancy issue in `yVault.deposit`

Submitted by cmichel

yVault.sol#L144-L145

In `deposit`, the balance is cached and then a `token.transferFrom` is triggered which can lead to exploits if the `token` is a token that gives control to the sender, like ERC777 tokens.

Proof of Concept

Initial state: `balance() = 1000`, shares `supply = 1000`. Depositing 1000 amount should mint 1000 supply, but one can split the 1000 amounts into two 500 deposits and use re-entrancy to profit.

- Outer `deposit(500)`: `balanceBefore = 1000`. Control is given to attacker ...
- Inner `deposit(500)`: `balanceBefore = 1000`. `shares = (_amount * supply) / balanceBefore = 500 * 1000 / 1000 = 500` shares are minted ...
- Outer `deposit(500)` continues with the mint: `shares = (_amount * supply) / balanceBefore = 500 * 1500 / 1000 = 750` are minted.
- Withdrawing the `500 + 750 = 1250` shares via `withdraw(1250)`, the attacker receives `backingTokens = (balance() * _shares) / supply = 2000 * 1250 / 2250 = 1111.111111111`. The attacker makes a profit of `1111 - 1000 = 111` tokens.

- They repeat the attack until the vault is drained.

Recommended Mitigation Steps

The `safeTransferFrom` should be the last call in `deposit`.

spaghettieth (JPEG'd) confirmed

spaghettieth (JPEG'd) resolved and commented:

> Fixed in jpegd/core#19.

[H-05] `yVaultLPFarming`: No guarantee JPEG currentBalance > previousBalance

Submitted by hickuphh3

yVaultLPFarming.sol#L169-L170

yVault users participating in the farm have to trust that:

- `vault.balanceOfJPEG()` returns the correct claimable JPEG amount by its strategy / strategies
- the strategy / strategies will send all claimable JPEG to the farm

Should either of these assumptions break, then it could possibly be the case that `currentBalance` is less than `previousBalance`, causing deposits and crucially, withdrawals to fail due to subtraction overflow.

Proof of Concept

For instance,

- Farm migration occurs. A new farm is set in `yVault`, then `withdrawJPEG()` is called, which sends funds to the new farm. Users of the old farm would be unable to withdraw their deposits.

```
it.only("will revert old farms' deposits and withdrawals if yVau
  // 0. setup
  await token.mint(owner.address, units(1000));
  await token.approve(yVault.address, units(1000));
  await yVault.depositAll();
  await yVault.approve(lpFarming.address, units(1000));
  // send some JPEG to strategy prior to deposit
  await jpeg.mint(strategy.address, units(100));
  // deposit twice, so that the second deposit will invoke _upda
  await lpFarming.deposit(units(250));
  await lpFarming.deposit(units(250));

  // 1. change farm and call withdrawJPEG()
  await yVault.setFarmingPool(user1.address);
  await yVault.withdrawJPEG();

  // deposit and withdrawal will fail
  await expect(lpFarming.deposit(units(500))).to.be.revertedWith
  await expect(lpFarming.withdraw(units(500))).to.be.revertedWit
});
```

- Strategy migration occurs, but JPEG funds held by the old strategy were not claimed, causing `vault.balanceOfJPEG()` to report a smaller amount than previously recorded
- `jpeg` could be accidentally included in the StrategyConfig, resulting in JPEG being converted to other assets
- A future implementation takes a fee on the `jpeg` to be claimed

Recommended Mitigation Steps

A simple fix would be to `return` if `currentBalance ≤ previousBalance`. A full fix would properly handle potential shortfall.

```
if (currentBalance <= previousBalance) return;
```

spaghettieth (JPEG'd) confirmed, but disagreed with High severity and commented:

> The issue can be reproduced, but due to the extremely specific cases in which this happens the severity should be lowered to 2.

spaghettieth (JPEG'd) resolved and commented:

> Fixed in jpegd/core#7.

LSDan (judge) commented:

> I disagree with the sponsor. This is high risk.

[H-06] Setting new controller can break `YVaultLPFarming`

Submitted by cmichel

yVaultLPFarming.sol#L170
yVault.sol#L108

The accruals in `yVaultLPFarming` will fail if `currentBalance < previousBalance` in `_computeUpdate`.

```
currentBalance = vault.balanceOfJPEG() + jpeg.balanceOf(address
uint256 newRewards = currentBalance - previousBalance;
```

No funds can be withdrawn anymore as the `withdraw` functions first trigger an `_update`.

The `currentBalance < previousBalance` case can, for example, be triggerd by decreasing the `vault.balanceOfJPEG()` due to calling `yVault.setController`:

```
function setController(address _controller) public onlyOwner {
    // @audit can reduce balanceofJpeg which breaks other master
    require(_controller != address(0), "INVALID_CONTROLLER");
    controller = IController(_controller);
}
```

```
    function balanceOfJPEG() external view returns (uint256) {
        // @audit new controller could return a smaller balance
        return controller.balanceOfJPEG(address(token));
    }
```

Recommended Mitigation Steps

Setting a new controller on a vault must be done very carefully and requires a migration.

[LSDan (judge) commented](#):

> This is not a duplicate of H-05. Though both of them deal with issues related to balanceOfJPEG, they describe different causes.

[spaghettieth (JPEG'd) acknowledged](#)

[H-07] Controller: Strategy migration will fail

Submitted by hickuphh3, also found by rayn

[Controller.sol#L95](#)
[StrategyPUSDConvex.sol#L266](#)

The controller calls the `withdraw()` method to withdraw JPEGs from the contract, but the strategy might blacklist the JPEG asset, which is what the PUSDConvex strategy has done.

The migration would therefore revert.

Proof of Concept

Insert this test into [`StrategyPUSDConvex.ts`](#).

```
    it.only("will revert when attempting to migrate strategy", async
      await controller.setVault(want.address, yVault.address);
      await expect(controller.setStrategy(want.address, strategy.adc
    });
```

Recommended Mitigation Steps

Replace `_current.withdraw(address(jpeg));` with `_current.withdrawJPEG(vaults[_token])`.

[spaghettieth (JPEG'd) confirmed and commented]:

> The proposed migration steps would modify the intended behaviour, which is to withdraw JPEG to the controller and not the vault. A correct solution would be replacing `_current.withdraw(address(jpeg))` with `_current.withdrawJPEG(address(this))`.

[spaghettieth (JPEG'd) resolved and commented]:

> Fixed in [jpegd/core#6].

[H-08] `StrategyPUSDConvex.balanceOfJPEG` uses incorrect function signature while calling `extraReward.earned`, causing the function to unexpectedly revert everytime

Submitted by rayn

[StrategyPUSDConvex.sol#L234]

As specified in Convex [BaseRewardPool.sol] and [VirtualRewardPool.sol], the function signature of `earned` is `earned(address)`. However, `balanceOfJPEG` did not pass any arguments to `earned`, which would cause `balanceOfJPEG` to always revert.

This bug will propagate through `Controller` and `YVault` until finally reaching the source of the call in `YVaultLPFarming ._computeUpdate`, and render the entire farming contract unuseable.

Proof of Concept

Both `BaseRewardPool.earned` and `VirtualBalanceRewardPool.earned` takes an address as argument

```
    function earned(address account) public view returns (uint25
        return
            balanceOf(account)
                .mul(rewardPerToken().sub(userRewardPerTokenPaic
                .div(1e18)
                .add(rewards[account]);
    }

    function earned(address account) public view returns (uint25
        return
            balanceOf(account)
                .mul(rewardPerToken().sub(userRewardPerTokenPaic
                .div(1e18)
                .add(rewards[account]);
    }
```

But `balanceOfJPEG` does not pass any address to `extraReward.earned`, causing the entire function to revert when called

```
    function balanceOfJPEG() external view returns (uint256) {
        uint256 availableBalance = jpeg.balanceOf(address(this))

        IBaseRewardPool baseRewardPool = convexConfig.baseRewarc
        uint256 length = baseRewardPool.extraRewardsLength();
        for (uint256 i = 0; i < length; i++) {
            IBaseRewardPool extraReward = IBaseRewardPool(baseRe
            if (address(jpeg) == extraReward.rewardToken()) {
                availableBalance += extraReward.earned();
                //we found jpeg, no need to continue the loop
                break;
            }
        }

        return availableBalance;
    }
```

Tools Used

vim, ganache-cli

Recommended Mitigation Steps

Pass `address(this)` as argument of `earned`.

Notice how we modify the fetching of reward. This is reported in a separate bug report, but for completeness, the entire fix is shown in both report entries.

```
        function balanceOfJPEG() external view returns (uint256) {
            uint256 availableBalance = jpeg.balanceOf(address(this))

            IBaseRewardPool baseRewardPool = convexConfig.baseRewarc
            availableBalance += baseRewardPool.earned(address(this))
            uint256 length = baseRewardPool.extraRewardsLength();
            for (uint256 i = 0; i < length; i++) {
                IBaseRewardPool extraReward = IBaseRewardPool(baseRe
                if (address(jpeg) == extraReward.rewardToken()) {
                    availableBalance += extraReward.earned(address(t
                }
            }

            return availableBalance;
        }
```

[spaghettieth (JPEG'd) confirmed, but disagreed with High severity]()

[spaghettieth (JPEG'd) resolved and commented]():

> Fixed in [jpegd/core#15]().

[LSDan (judge) commented]():

> Leaving this as high risk. The issue would cause a loss of funds.

[H-09] Bad debts should not continue to accrue interest

Submitted by WatchPug

[NFTVault.sol#L844-L851]()

```
    uint256 debtAmount = _getDebtAmount(_nftIndex);
    require(
        debtAmount >= _getLiquidationLimit(_nftIndex),
        "position_not_liquidatable"
    );

    // burn all payment
    stablecoin.burnFrom(msg.sender, debtAmount);
```

In the current design/implementation, the liquidator must fully repay the user's outstanding debt in order to get the NFT.

When the market value of the NFT fell rapidly, the liquidators may not be able to successfully liquidate as they can not sell the NFT for more than the debt amount.

In that case, the protocol will have positions that are considered bad debts.

However, these loans, which may never be repaid, are still accruing interest. And every time the DAO collects interest, new `stablecoin` will be minted.

When the proportion of bad debts is large enough since the interest generated by these bad debts is not backed. It will damage the authenticity of the stablecoin.

Proof of Concept

Given:

- `NFT 1` worth 30,000 USD
- `creditLimitRate` = 60%
- `liquidationLimitRate` = 50%
- `debtInterestApr` = 10%
- Alice borrowed `10,000 USD` with `NFT #1`;
- After 1 year, `NFT 1`'s market value in USD has suddenly dropped to `10,000` USD, no liquidator is willing to repay 11,000 USD for `NFT #1`;
- The DAO `collect()` and minted `1,000` stablecoin;
- After 1 year, the DAO call `collect()` will mint `1,100` stablecoin. and so on...

Recommended Mitigation Steps

Consider adding a stored value to record the amount of bad debt, and add a public function that allows anyone to mark a bad debt to get some reward. and change `accrue` to:

```
    uint256 internal badDebtPortion;

    function accrue() public {
        uint256 additionalInterest = _calculateAdditionalInterest();

        totalDebtAccruedAt = block.timestamp;

        totalDebtAmount += additionalInterest;

        uint256 collectibleInterest = additionalInterest * (totalDeb
        totalFeeCollected += collectibleInterest;
    }
```

spaghettieth (JPEG'd) acknowledged, but disagreed with High severity

LSDan (judge) commented:

> I agree with the warden. Left unchecked, this issue is almost certain to occur and will cause substantial negative impacts on the protocol. The only way this would not occur is if the NFT market never crashes.

Medium Risk Findings (11)

[M-01] When _lpToken is jpeg, reward calculation is incorrect

Submitted by cccz, also found by minhquanym

In the LPFarming contract, a new staking pool can be added using the add() function. The staking token for the new pool is defined using the _lpToken variable. However, there is no additional checking whether the _lpToken is the same as the reward token (jpeg) or not.

```
    function add(uint256 _allocPoint, IERC20 _lpToken) external
        _massUpdatePools();

        uint256 lastRewardBlock = _blockNumber();
        totalAllocPoint = totalAllocPoint + _allocPoint;
        poolInfo.push(
            PoolInfo({
                lpToken: _lpToken,
                allocPoint: _allocPoint,
                lastRewardBlock: lastRewardBlock,
                accRewardPerShare: 0
            })
        );
    }
```

When the _lpToken is the same token as jpeg, reward calculation for that pool in the updatePool() function can be incorrect. This is because the current balance of the _lpToken in the contract is used in the calculation of the reward. Since the _lpToken is the same token as the reward, the reward minted to the contract will inflate the value of lpSupply, causing the reward of that pool to be less than what it should be.

```
    function _updatePool(uint256 _pid) internal {
        PoolInfo storage pool = poolInfo[_pid];
        if (pool.allocPoint == 0) {
            return;
        }

        uint256 blockNumber = _blockNumber();
        //normalizing the pool's `lastRewardBlock` ensures that
        uint256 lastRewardBlock = _normalizeBlockNumber(pool.las
        if (blockNumber <= lastRewardBlock) {
            return;
        }
        uint256 lpSupply = pool.lpToken.balanceOf(address(this))
        if (lpSupply == 0) {
            pool.lastRewardBlock = blockNumber;
            return;
        }
        uint256 reward = ((blockNumber - lastRewardBlock) *
            epoch.rewardPerBlock *
            1e36 *
            pool.allocPoint) / totalAllocPoint;
        pool.accRewardPerShare = pool.accRewardPerShare + rewarc
        pool.lastRewardBlock = blockNumber;
    }
```

Proof of Concept

[LPFarming.sol#L141-L154]
[LPFarming.sol#L288-L311]

Recommended Mitigation Steps

Add a check that _lpToken is not jpeg in the add function or mint the reward token to another contract to prevent the amount of the staked token from being mixed up with the reward token.

spaghettieth (JPEG'd) confirmed

spaghettieth (JPEG'd) resolved and commented:

> Fixed in jpegd/core#2.

[M-02] NFTHelper Contract Allows Owner to Burn NFTs

Submitted by Kenshin

CryptoPunksHelper.sol#L38
CryptoPunksHelper.sol#L52

In the NFT helper contract, there is no validation on that the receiver address must not be address zero. Therefore, it allows owner or an attacker who gain access to the owner address to burn NFTs forever through the functions by transferring the NFTs to address zero.

Proof of Concept

The PoC is originally conducted using foundry. However, it isn't that complicated so I rewrote it in TypeScipt as well, the team can easily proof this by including in the `CryptoPunksHelper.ts`.

TypeScript

```
// add `.only` to run only this test, not all.
it.only("allows the owner to burn nfts", async () => {
    // safeTransferFrom
    await cryptoPunks.getPunk(1);
    await cryptoPunks.transferPunk(helper.address, 1);
    await helper.safeTransferFrom(owner.address, ZERO_ADDRESS, 1
    expect(await cryptoPunks.punkIndexToAddress(1)).to.equal(ZEF
    expect(await helper.ownerOf(1)).to.equal(ZERO_ADDRESS);

    // transferFrom
    await cryptoPunks.getPunk(2);
    await cryptoPunks.transferPunk(helper.address, 2);
    await helper.transferFrom(owner.address, ZERO_ADDRESS, 2);
    expect(await cryptoPunks.punkIndexToAddress(2)).to.equal(ZEF
    expect(await helper.ownerOf(2)).to.equal(ZERO_ADDRESS);
  });
```

Foundry

```
pragma solidity ^0.8.0;

// for test
import "ds-test/test.sol";
import "forge-std/Vm.sol";

// contracts
import "../test/CryptoPunks.sol";
import "../helpers/CryptoPunksHelper.sol";

contract CryptoPunksHelperTest is DSTest {
    Vm constant vm = Vm(HEVM_ADDRESS);

    address owner = address(1);
    address user = address(2);

    CryptoPunks private cps;
    CryptoPunksHelper private helper;

    function setUp() public {
        vm.startPrank(owner);
        cps = new CryptoPunks();
        helper = new CryptoPunksHelper();
        helper.initialize(address(cps));
        vm.stopPrank();
    }

    function testOwnerTransferToZero() public {
        //make sure address zero hold no punks
        assertEq(cps.balanceOf(address(0)), 0);

        // safeTransferFrom PoC
        vm.startPrank(owner);
        cps.getPunk(1);
        cps.transferPunk(address(helper), 1);
        helper.safeTransferFrom(owner, address(0), 1);
        assertEq(cps.punkIndexToAddress(1), address(0));
        assertEq(helper.ownerOf(1), address(0));
        assertEq(cps.balanceOf(address(0)), 1);

        // transferFrom PoC
        cps.getPunk(2);
        cps.transferPunk(address(helper), 2);
        helper.transferFrom(owner, address(0), 2);
        assertEq(cps.punkIndexToAddress(2), address(0));
        assertEq(helper.ownerOf(2), address(0));
        assertEq(cps.balanceOf(address(0)), 2);
    }
}
```

foundry.toml

```
[default]
src = "contracts"
libs = ["lib/forge-std/lib", "lib/", "node_modules"]
solc_version = "0.8.0"
optimizer = false
fuzz_runs = 100000
test = "foundryTest"
```

Tools Used

- Foundry

- Hardhat

Recommended Mitigation Steps

Even the functions are restricted for only the owner, the zero address should not be allowed as the receiver address.

spaghettieth (JPEG'd) acknowledged and commented:

> The sole purpose of `CryptoPunksHelper.sol` and `EtherRocksHelper.sol` contracts is to allow compatibility of non ERC721 NFTs to be compatible with `NFTVault.sol` without having to modify the vault's code. They don't have to provide any additional security check outside of compatibility related ones, everything else is out of scope and should be handled by the underlying NFT.

[M-03] reward will be locked in the farm if no LP join the pool at epoch.startBlock

Submitted by TrungOre

LPFarming.sol#L214
LPFarming.sol#L107

A part of reward tokens will be locked in the farming pool if no user deposits lpToken at the epoch.startBlock.

Proof of Concept

```
    it("a part of reward should be locked in farm if no LP join the
        // manual mine new block
        await network.provider.send("evm_setAutomine", [false]);

        // prepare
        await lpTokens[0].transfer(alice.address, units(1000));
        await lpTokens[0].connect(alice).approve(farming.address,
        await mineBlocks(1);

        // create new pool
        await farming.add(10, lpTokens[0].address);
```

```
      await mineBlocks(1);
      expect(await farming.poolLength()).to.equal(1);

      let pool = await farming.poolInfo(0);
      expect(pool.lpToken).to.equal(lpTokens[0].address);
      expect(pool.allocPoint).to.equal(10);

      // create new epoch ==> balance of pool will be 1000
      let blockNumber = await ethers.provider.getBlockNumber();
      await farming.newEpoch(blockNumber + 1, blockNumber + 11,

      // skip the epoch.startBlock
      // it mean no one deposit lpToken to farm at this block
      await mineBlocks(1);
      expect(await jpeg.balanceOf(farming.address)).to.equal(10(

      // alice deposit
      await farming.connect(alice).deposit(0, units(100));
      await mineBlocks(1);

      // skip the blocks to the end of epoch
      await mineBlocks(13);

      await farming.connect(alice).claim(0);
      await mineBlocks(1);

      console.log("reward of alice: ", (await jpeg.balanceOf(ali
      console.log("reward remain: ", await jpeg.balanceOf(farmir

      // 100 jpeg will be locked in the pool forevers
      expect(await jpeg.balanceOf(alice.address)).to.equal(900);
      expect(await jpeg.balanceOf(farming.address)).to.equal(10(
    });
```

In the example above, I create an epoch from blockNumber + 1 to blockNumber + 11 with the reward for each block being 100JPEG. So, the total reward for this farm will be 1000JPEG. When I skip the epoch.startBlock and let Alice deposit 100 lpToken at the block right after, at the end of the farm (epoch.endBlock), the total reward of Alice is just 900JPEG, and 100JPEG still remains in the farming pool. Since there is no function for the admin (or users) to withdraw the remaining, 100JPEG will be stucked in the pool forever !!!

Tools Used

typescript

Recommended Mitigation Steps

Add a new function for the admin (or user) to claim all rewards which remained in the pool when epoch.endTime has passed

```
    function claimRemainRewardsForOwner() external onlyOwner {
            require(
                block.number > epoch.endBlock,
                'epoch has not ended'
            );
            uint256 remain = jpeg.balanceOf(address(this));
            jpeg.safeTransfer(msg.sender, remain);
        }
```

spaghettieth (JPEG'd) acknowledged, but disagreed with Medium severity and commented:

> This is a very minor issue, severity should be 0.

LSDan (judge) commented:

> I disagree with the sponsor. Funds are lost in this scenario and it is very easy to mitigate.

[M-04] `setDebtInterestApr` should accrue debt first

Submitted by cmichel, also found by pedroais

NFTVault.sol#L212

The `setDebtInterestApr` changes the debt interest rate without first accruing the debt.
This means that the new debt interest rate is applied retroactively to the unaccrued period on next `accrue()` call.

It should never be applied retroactively to a previous time window as this is unfair & wrong.
Borrowers can incur more debt than they should.

Recommended Mitigation Steps

Call `accrue()` first in `setDebtInterestApr` before setting the new `settings.debtInterestApr`.

spaghettieth (JPEG'd) confirmed

spaghettieth (JPEG'd) resolved and commented:

> Fixed in jpegd/core#4.

[M-05] Rewards will be locked if user transfer directly to pool without using deposit function

Submitted by TrungOre

LPFarming.sol#L190

Reward will be locked in the farming, when user execute a direct transfer with lpToken to farm without using deposit.

Proof of Concept

"pls add this test to LpFarming.ts to check"

```
    it("a part of rewards can't be distributed if user execute a dir
          // manual mine new block
          await network.provider.send("evm_setAutomine", [false]);

          // prepare
          const attacker = bob;
          await lpTokens[0].transfer(alice.address, units(1000));
          await lpTokens[0].transfer(attacker.address, units(1000));
          await lpTokens[0].connect(alice).approve(farming.address,
          await mineBlocks(1);
          // attacker direct deposit lp token to the pool
          await lpTokens[0].connect(attacker).transfer(farming.addre

          // create new pool
          await farming.add(10, lpTokens[0].address);
          await mineBlocks(1);
          expect(await farming.poolLength()).to.equal(1);

          let pool = await farming.poolInfo(0);
          expect(pool.lpToken).to.equal(lpTokens[0].address);
          expect(pool.allocPoint).to.equal(10);

          // create new epoch ==> balance of pool will be 1000
          let blockNumber = await ethers.provider.getBlockNumber();
          await farming.newEpoch(blockNumber + 1, blockNumber + 11,

          // alice deposit
          await farming.connect(alice).deposit(0, units(100));
          await mineBlocks(1);

          expect(await jpeg.balanceOf(farming.address)).to.equal(10(

          // when pool end, alice can just take 500 jpeg, and 500 jp
          await mineBlocks(13);
          console.log("reward of alice: ", (await   farming.pendingR
          expect(await farming.pendingReward(0, alice.address)).to.e
        });
```

In the test above, the attacker transfers 100 lpToken to the farm without using deposit function, and alice deposit 100 lpToken. Because the contract uses `pool.lpToken.balanceOf(address(this))` to get the total supply of lpToken in the pool, it will sum up 100 lpToken of attacker and 100 lpToken of alice. This will lead to the situation where Alice will only be able to claim 500 token (at epoch.endBlock), the rest will be locked in the pool forever. Not only with this pool, it also affects the following, a part of the reward will be locked in the pool when the farm end.

Tools Used

typescript

Recommended Mitigation Steps

Declare a new variable `totalLPSupply` to the struct `PoolInfo` , and use it instead of `pool.lpToken.balanceOf(address(this))` .

spaghettieth (JPEG'd) confirmed, but disagreed with High severity and commented:

> Very minor issue, severity should be 0.

spaghettieth (JPEG'd) resolved and commented:

> Fixed in jpegd/core#2.

LSDan (judge) decreased severity to Medium and commented:

> I'm going to downgrade this to a medium. There is a possibility for lost funds given real world external factors (user stupidity).

[M-06] Oracle data feed is insufficiently validated.

Submitted by Jujic, also found by hickuphh3

Price can be stale and can lead to wrong `answer` return value.

Proof of Concept

Oracle data feed is insufficiently validated. There is no check for stale price and round completeness. Price can be stale and can lead to wrong `answer` return value.

```
    function _collateralPriceUsd() internal view returns (uint256) {
            int256 answer = oracle.latestAnswer();
            uint8 decimals = oracle.decimals();

            require(answer > 0, "invalid_oracle_answer");

            ...
```

FungibleAssetVaultForDAO.sol#L105

Recommended Mitigation Steps

Validate data feed

```
    function _collateralPriceUsd() internal view returns (uint256) {

    (uint80 roundID, int256 answer, , uint256 timestamp, uint80 answ

        require(answer > 0, "invalid_oracle_answer");
        require(answeredInRound >= roundID, "ChainLink: Stale price'
        require(timestamp > 0, "ChainLink: Round not complete");

             ...
```

0xJPEG (JPEG'd) confirmed, but disagreed with High severity and commented:

> Can add validation for round not being complete yet and potentially for stale pricing.
> This should be medium risk, as shown in past contests [1] [2] [3]

spaghettieth (JPEG'd) resolved and commented:

> Fixed in jpegd/core#9.

LSDan (judge) decreased severity to Medium and commented:

> Agree with sponsor on the medium risk rating. An oracle with a bad value is by definition an external requirement.

[M-07] Wrong calculation for `yVault` price per share if decimals != 18

Submitted by berndartmueller

The yVault.getPricePerFullShare() function calculates the price per share by multiplying with `1e18` token decimals with the assumption that the underlying token always has 18 decimals. `yVault` has the same amount of decimals as it's underlying token see (yVault.decimals())

But tokens don't always have `1e18` decimals (e.g. USDC).

Impact

The price per share calculation does not return the correct price for underlying tokens that do not have 18 decimals. This could lead to paying out too little or too much and therefore to a loss for either the protocol or the user.

Proof of Concept

Following test will fail with the current implementation when the underlying vault token has 6 decimals:

NOTE: `units()` *helper function was adapted to accept the desired decimals.*

```
    it.only("should mint the correct amount of tokens for tokens wit
      const DECIMALS = 6;

      await token.setDecimals(DECIMALS);
      expect(await yVault.decimals()).to.equal(DECIMALS);

      expect(await yVault.getPricePerFullShare()).to.equal(0);
      await token.mint(user1.address, units(1000, DECIMALS));
      await token.connect(user1).approve(yVault.address, units(1000,

      await yVault.connect(user1).deposit(units(500, DECIMALS));
      expect(await yVault.balanceOf(user1.address)).to.equal(units(5

      await token.mint(strategy.address, units(500, DECIMALS));
      expect(await yVault.getPricePerFullShare()).to.equal(units(2,
    });
```

Fails with following error: `AssertionError: Expected "2000000000000000000" to be equal 2000000`

Recommended mitigation steps

Use vault `decimals()` instead of hardcoded `1e18` decimals.

```
    function getPricePerFullShare() external view returns (uint256)
        uint256 supply = totalSupply();
        if (supply == 0) return 0;
        return (balance() * (10**decimals())) / supply; // @audit-ir
    }
```

[spaghettieth (JPEG'd) disputed and commented]:

> The `yVault` contract has been designed to work with Curve LPs, which have 18 decimals

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

> I'm downgrading this to a medium risk but leaving it as valid. Any number of external factors could conspire to result in a non-18 decimal token being used in the future, at which point this code may have been forgotten. A better choice would be to do a decimal check.

[M-08] `_swapUniswapV2` may use an improper `path` which can cause a loss of the majority of the rewardTokens

Submitted by WatchPug

[StrategyPUSDConvex.sol#L311-L334]

```
    function harvest(uint256 minOutCurve) external onlyRole(STRATEGI
        convexConfig.baseRewardPool.getReward(address(this), true);

        //Prevent `Stack too deep` errors
        {
            DexConfig memory dex = dexConfig;
            IERC20[] memory rewardTokens = strategyConfig.rewardToke
            IERC20 _weth = weth;
            for (uint256 i = 0; i < rewardTokens.length; i++) {
                uint256 balance = rewardTokens[i].balanceOf(address(
                if (balance > 0)
                    //minOut is not needed here, we already have it
                    _swapUniswapV2(
                        dex.uniswapV2,
                        rewardTokens[i],
                        _weth,
                        balance,
                        0
                    );
            }

            uint256 wethBalance = _weth.balanceOf(address(this));
            require(wethBalance > 0, "NOOP");
            ...
```

StrategyPUSDConvex.sol#L410-L430

```
    function _swapUniswapV2(
        IUniswapV2Router router,
        IERC20 tokenIn,
        IERC20 tokenOut,
        uint256 amountIn,
        uint256 minOut
    ) internal {
        tokenIn.safeIncreaseAllowance(address(router), amountIn);

        address[] memory path = new address[](2);
        path[0] = address(tokenIn);
        path[1] = address(tokenOut);

        router.swapExactTokensForTokens(
            amountIn,
            minOut,
            path,
            address(this),
            block.timestamp
        );
    }
```

In the current implementation, `rewardTokens` from the underlying strategy will be swapped to `weth` first then `weth` -> `usdc`.

However, the `path` used for swapping from `rewardToken -> weth` is hardcoded as `[rewardToken, weth]`, which may not be the optimal route.

For example, the majority liquidity for a particular `rewardToken` may actually be in the `rewardToken/USDC` pool. Swapping through the `rewardToken/WETH` with low liquidity may end up getting only a dust amount of WETH.

Recommended Mitigation Steps

Consider allowing the admin to set a path for the rewardTokens.

spaghettieth (JPEG'd) disputed and commented:

> As of now, the one in the contract is the optimal routing path.

LSDan (judge) decreased severity to Medium and commented:

> I think the warden has made a reasonable find and recommendation. The sponsor used the phrase 'as of now' in disputing the report, but the idea that it may not always be the optimal path is actually specifically what the report and its mitigation addresses. That said, external factors are required so moving it to medium severity.

[M-09] The noContract modifier does not work as expected.

*Submitted by Wayne, also found by hyh, PPrieditis, rayn, smiling*heretic, and Cr4ckM3_

yVault.sol#L61
yVaultLPFarming.sol#L54

The expectation of the noContract modifier is to allow access only to accounts inside EOA or Whitelist, if access is controlled using ! access control with _account.isContract(), then because isContract() gets the size of the code length of the account in question by relying on extcodesize/address.code.length, this means that the restriction can be bypassed when deploying a smart contract through the smart contract's constructor call.

Recommended Mitigation Steps

Modify the code to `require(msg.sender == tx.origin);`

[spaghettieth (JPEG'd) confirmed, but disagreed with Medium severity and commented](#):

> The observations made in the issue are correct, but given how impractical it would be to make an autocompounder that bypasses the `noContract` modifier this issue should probably be severity 0. It's worth mentioning that this behaviour was already known as outlined in the modifier's documentation.

[spaghettieth (JPEG'd) resolved and commented](#):

> Fixed in [jpegd/core#17](#).

[LSDan (judge) commented](#):

> I'm going to let this stand. Impractical or not, this is very easily exploited.

[M-10] Chainlink pricer is using a deprecated API

Submitted by cccz, also found by 0xDjango, 0xkatana, berndartmueller, Cr4ckM3, defsec, horsefacts, hyh, JMukesh, joshie, Jujic, pedroais, peritoflores, rayn, reassor, Ruhum, and WatchPug

According to Chainlink's documentation, the latestAnswer function is deprecated. This function might suddenly stop working if Chainlink stops supporting deprecated APIs. And the old API can return stale data.

Proof of Concept

[FungibleAssetVaultForDAO.sol#L105](#)
[NFTVault.sol#L459](#)

Recommended Mitigation Steps

Use the latestRoundData function to get the price instead. Add checks on the return data with proper revert messages if the price is stale or the round is uncomplete.

https://docs.chain.link/docs/price-feeds-api-reference/

spaghettieth (JPEG'd) confirmed

spaghettieth (JPEG'd) resolved and commented:

> Fixed in jpegd/core#9.

[M-11] Division before Multiplication May Result In No Interest Being Accrued

Submitted by AuditsAreUS, also found by minhquanym

NFTVault.sol#L590-L595

There is a division before multiplication bug in `NFTVault._calculateAdditionalInterest()` which may result in no interesting being accrued and will have significant rounding issues for tokens with small decimal places.

This issue occurs since an intermediate calculation of `interestPerSec` may round to zero and therefore the multiplication by `elapsedTime` may remain zero.

Furthermore, even if `interestPerSec > 0` there will still be rounding errors as a result of doing division before multiplication and `_calculatedInterest()` will be understated.

This issue is significant as one divisor is 365 days = 30,758,400 (excluding the rate). Since many ERC20 tokens such as USDC and USDT only have 6 decimal places a numerator of less 30 * 10^6 will round to zero.

The rate also multiplies into the denominator. e.g. If the rate is 1% then the denominator will be equivalent to `1 / rate * 30 * 10^6 = 3,000 * 10^6`.

Proof of Concept

The order of operations for the interest calculations

- `totalDebtAmount`
- MUL `settings.debtInterestApr.numerator`
- DIV `settings.debtInterestApr.denominator`
- DIV `365 days`
- MUL `elapsedTime`

If the intermediate value of `interestPerSec = 0` then the multiplication by `elapsedTime` will still be zero and no interested will be accrued.

Excerpt from `NFTVault._calculateAdditionalInterest()`.

```
        uint256 interestPerYear = (totalDebtAmount *
            settings.debtInterestApr.numerator) /
            settings.debtInterestApr.denominator;
        uint256 interestPerSec = interestPerYear / 365 days;

        return elapsedTime * interestPerSec;
```

Recommended Mitigation Steps

This issue may be resolved by performing the multiplication by `elapsedTime` before the division by the denominator or `365 days`.

```
        uint256 interestAccrued = (elapsedTime *
            totalDebtAmount *
            settings.debtInterestApr.numerator) /
            settings.debtInterestApr.denominator /
            365 days;

        return  interestAccrued;
```

spaghettieth (JPEG'd) confirmed

spaghettieth (JPEG'd) resolved and commented:

> Fixed jpegd/core#8.

LSDan (judge) commented:

> This report makes sense as a medium to me because it involves a calculation error that can lead to the protocol functioning incorrectly.

Low Risk and Non-Critical Issues

For this contest, 42 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **Dravee** received the top score from the judge.

The following wardens also submitted reports: IllIllI, 0xDjango, robee, PPrieditis, Kthere, hickuphh3, 0xkatana, WatchPug, horsefacts, ilan, Ruhum, berndartmueller, hubble, hyh, kebabsec, kenta, Hawkeye, TerrierLover, rfa, catchup, rayn, Foundation, Funen, Picodes, TrungOre, delfin454000, ellahi, JMukesh, 0x1f8b, Cityscape, Jujic, JC, dy, AuditsAreUS, pauliax, cccz, cmichel, reassor, minhquanym, samruna, *and* jayjonah8.

Codebase Impressions & Summary

Overall, the code quality is high.

The findings here revolve around some commonly suggested practices.

[L-01] Add constructor initializers

As per OpenZeppelin's (OZ) recommendation, "The guidelines are now to make it impossible for *anyone* to run `initialize` on an implementation contract, by adding an empty constructor with the `initializer` modifier. So the implementation contract gets initialized automatically upon deployment."

Note that this behaviour is also incorporated the OZ Wizard since the UUPS vulnerability discovery: "Additionally, we modified the code generated by the Wizard 19 to include a constructor that automatically initializes the implementation when deployed."

Furthermore, this thwarts any attempts to frontrun the initialization tx of these contracts:

```
contracts/helpers/CryptoPunksHelper.sol:
  19:     function initialize(address punksAddress) external ini

contracts/helpers/EtherRocksHelper.sol:
  19:     function initialize(address rocksAddress) external ini

contracts/staking/JPEGStaking.sol:
  21:     function initialize(IERC20Upgradeable _jpeg) external

contracts/vaults/FungibleAssetVaultForDAO.sol:
  66:     function initialize(

contracts/vaults/NFTVault.sol:
  139:     function initialize(
```

[L-02] Immutable addresses should be 0-checked

Consider adding an `address(0)` check here (see `@audit`):

```
contracts/farming/LPFarming.sol:
  77:         jpeg = IERC20(_jpeg); //@audit low: should be addr

contracts/vaults/yVault/Controller.sol:
  28:         jpeg = IERC20(_jpeg);  //@audit low: should be add

contracts/vaults/yVault/yVault.sol:
  53:         token = ERC20(_token);  //@audit low: should be ad
```

[L-03] Unbounded loop on array can lead to DoS

As this array can grow quite large, the transaction's gas cost could exceed the block gas limit and make it impossible to call this function at all (see `@audit`):

```
File: LPFarming.sol
141:     function add(uint256 _allocPoint, IERC20 _lpToken) exte
...
```

```
146:            poolInfo.push( //@audit low: a push exist but there
...
154:        }
...
347:        function claimAll() external nonReentrant noContract(ms
348:            for (uint256 i = 0; i < poolInfo.length; i++) { //@
349:                _updatePool(i);
350:                _withdrawReward(i);
351:            }
...
360:        }
```

Consider introducing a reasonable upper limit based on block gas limits and/or adding a `remove` method to remove elements in the array.

[L-04] Add a timelock and event to critical functions

It is a good practice to give time for users to react and adjust to critical changes. A timelock provides more guarantees and reduces the level of trust required, thus decreasing risk for users. It also indicates that the project is legitimate.

Consider adding a timelock and event to:

```
vaults/yVault/strategies/StrategyPUSDConvex.sol:177:    functior
vaults/NFTVault.sol:290:    function setOrganizationFeeRate(Rate
```

[L-05] Fee in `StrategyPUSDConvex.setPerformanceFee()` should be upper-bounded

See `@audit`:

```
File: StrategyPUSDConvex.sol
177:        function setPerformanceFee(Rate memory _performanceFee)
178:            public
179:            onlyRole(DEFAULT_ADMIN_ROLE)
180:        {
181:            require(
182:                _performanceFee.denominator > 0 &&
183:                    _performanceFee.denominator >= _performance
```

```
                "INVALID_RATE"
            );
            performanceFee = _performanceFee; //@audit low: fee
        }
```

[L-06] Fee in `NFTVault.setOrganizationFeeRate()` should be upper-bounded

See `@audit`:

```
File: NFTVault.sol
        function setOrganizationFeeRate(Rate memory _organizati
            external
            onlyRole(DAO_ROLE)
        {
            _validateRate(_organizationFeeRate);
            settings.organizationFeeRate = _organizationFeeRate
        }
...
        function _validateRate(Rate memory rate) internal pure
            require(
                rate.denominator > 0 && rate.denominator >= rat
                "invalid_rate"
            );
        }
```

[L-07] A magical number should be documented and explained: `1e36`. Use a constant instead

```
farming/LPFarming.sol:196:                1e36 *
farming/LPFarming.sol:207:            1e36;
farming/LPFarming.sol:307:            1e36 *
farming/LPFarming.sol:319:            1e36;
farming/yVaultLPFarming.sol:94:            1e36;
farming/yVaultLPFarming.sol:172:        newAccRewardsPerShare =
farming/yVaultLPFarming.sol:179:            (accRewardPerShare -
```

I suggest using `constant` variables as this would make the code more maintainable and readable while costing nothing gas-wise.

[N-01] Avoid floating pragmas: the version should be locked (preferably at >= `0.8.4`)

The pragma declared across the solution is `^0.8.0`. As the compiler introduces a several interesting upgrades in Solidity `0.8.4`, consider locking at this version or a more recent one.

[N-02] Related data should be grouped in a struct

The following `maps` should be grouped in structs.

From:

```
contracts/farming/yVaultLPFarming.sol:
  31:     mapping(address => uint256) public balanceOf; //@audit
  32:     mapping(address => uint256) private userLastAccRewardP
  33:     mapping(address => uint256) private userPendingRewards

contracts/vaults/yVault/Controller.sol:
  20:     mapping(IERC20 => address) public vaults; //@audit NC:
  21:     mapping(IERC20 => IStrategy) public strategies; //@auc
  22:     mapping(IERC20 => mapping(IStrategy => bool)) public a
```

To

```
    struct UserInfo {
        uint256 balance;
        uint256 lastAccRewardPerShare;
        uint256 pendingReward;
    }

    mapping(address => UserInfo) public userInfo;
```

And

```
    struct TokenInfo {
        address vaults;
        IStrategy approvedStrategies;
        mapping(IStrategy => bool) pendingReward;
    }

    mapping(IERC20 => TokenInfo) public tokenInfo;
```

It would be less error-prone, more readable, and it would be possible to delete all related fields with a simple `delete userInfo[address]`.

However, the sponsor should notice that `pendingReward` won't be as easily deleted in `tokenInfo`, as it's a `mapping` field. It would still improve code quality nonetheless.

[N-03] Unused named returns

Using both named returns and a return statement isn't necessary. Removing one of those can improve code clarity (see `@audit`):

```
File: NFTEscrow.sol
    function precompute(address _owner, uint256 _idx)
        public
        view
        returns (bytes32 salt, address predictedAddress) //
    {
...
        salt = sha256(abi.encodePacked(_owner));
...
        predictedAddress = address(uint160(uint256(hash)));
        return (salt, predictedAddress); //@audit NC: unuse
    }
```

spaghettieth (JPEG'd) acknowledged

LSDan (judge) commented:

> I agree with the warden's security ratings for the issues described.

Gas Optimizations

For this contest, 38 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **Dravee** received the top score from the judge.

The following wardens also submitted reports: IllIllI, rfa, 0xkatana, TerrierLover, saian, Tomio, 0xNazgul, Funen, robee, Foundation, WatchPug, catchup, 0v3rf10w, berndartmueller, ellahi, securerodd, 0x1f8b, Cityscape, FSchmoede, hickuphh3, ilan, kenta, slywaters, rokinot, 0xDjango, JMukesh, nahnah, PPrieditis, sorrynotsorry, delfin454000, Hawkeye, Meta0xNull, dirk_y, rayn, Cr4ckM3, Picodes, *and* kebabsec.

Table of Contents

See original submission.

[G-01] `NFTEscrow._executeTransfer()` : Cheap Contract Deployment Through Clones

See `@audit` tag:

```
    function _executeTransfer(address _owner, uint256 _idx)
        (bytes32 salt, ) = precompute(_owner, _idx);
        new FlashEscrow{salt: salt}( //@audit gas: deploymer
            nftAddress,
            _encodeFlashEscrowPayload(_idx)
        );
    }
```

There's a way to save a significant amount of gas on deployment using Clones: https://www.youtube.com/watch?v=3Mw-pMmJ7TA .

This is a solution that was adopted, as an example, by Porter Finance. They realized that deploying using clones was 10x cheaper:

- https://github.com/porter-finance/v1-core/issues/15#issuecomment-1035639516
- https://github.com/porter-finance/v1-core/pull/34

I suggest applying a similar pattern.

[G-02] `LPFarming.newEpoch()` : L128 and L133 should be unchecked due to parent if/else condition

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

I suggest wrapping with an `unchecked` block here (see `@audit` tag):

```
107:     function newEpoch(
...
111:     ) external onlyOwner {
127:         if (remainingRewards > newRewards) {
128:             jpeg.safeTransfer(msg.sender, remainingRewards
129:         } else if (remainingRewards < newRewards) {
130:             jpeg.safeTransferFrom(
131:                 msg.sender,
132:                 address(this),
133:                 newRewards - remainingRewards  //@audit gas
134:             );
135:         }
136:     }
```

[G-03] `LPFarming.withdraw()` : L248 should be unchecked due to L243

See `@audit` tag:

```
235:     function withdraw(uint256 _pid, uint256 _amount)
236:         external
237:         noContract(msg.sender)
238:     {
239:         require(_amount > 0, "invalid_amount");
240:
```

```
241:         PoolInfo storage pool = poolInfo[_pid];
242:         UserInfo storage user = userInfo[_pid][msg.sender];
243:         require(user.amount >= _amount, "insufficient_amour
244: 
245:         _updatePool(_pid);
246:         _withdrawReward(_pid);
247: 
248:         user.amount -= _amount;  //@audit gas: should be ur
```

[G-04] `LPFarming._withdrawReward()` : `poolInfo[_pid].accRewardPerShare` should get cached

The code can be optimized by minimising the number of SLOADs. SLOADs are expensive (100 gas) compared to MLOADs/MSTOREs (3 gas). Here, the storage value should get cached in memory (see the `@audit` tags for further details):

```
315:     function _withdrawReward(uint256 _pid) internal returns
316:         UserInfo storage user = userInfo[_pid][msg.sender];
317:         uint256 pending = (user.amount *
318:             (poolInfo[_pid].accRewardPerShare - user.lastAc
319:             1e36;
320:         if (pending > 0) {
321:             userRewards[msg.sender] += pending;
322:         }
323: 
324:         user.lastAccRewardPerShare = poolInfo[_pid].accRewa
325: 
326:         return pending;
327:     }
```

[G-05] `yVaultLPFarming.withdraw()` : L124 should be unchecked due to L119

See `@audit` tag:

```
117:     function withdraw(uint256 _amount) external noContract
118:         require(_amount > 0, "invalid_amount");
119:         require(balanceOf[msg.sender] >= _amount, "insuffic
120: 
```

```
121:            _update();
122:            _withdrawReward(msg.sender);
123:
124:            balanceOf[msg.sender] -= _amount;  //@audit gas: sh
```

[G-06] `yVaultLPFarming._withdrawReward()` : `accRewardPerShare` should get cached

See `@audit` tags:

```
177:        function _withdrawReward(address account) internal retu
178:            uint256 pending = (balanceOf[account] *
179:                (accRewardPerShare - userLastAccRewardPerShare[
180:
181:            if (pending > 0) userPendingRewards[account] += pen
182:
183:            userLastAccRewardPerShare[account] = accRewardPerSh
184:
185:            return pending;
186:        }
```

[G-07] `JPEGLock.unlock()` : use `storage` instead of copying struct in memory L69

See `@audit` tag:

```
68:        function unlock(uint256 _nftIndex) external nonReentrant
69:            LockPosition memory position = positions[_nftIndex];
70:            require(position.owner == msg.sender, "unauthorized'
71:            require(position.unlockAt <= block.timestamp, "locke
72:
73:            delete positions[_nftIndex];
74:
75:            jpeg.safeTransfer(msg.sender, position.lockAmount);
76:
77:            emit Unlock(msg.sender, _nftIndex, position.lockAmou
78:        }
```

Here, a copy in memory is costing 3 SLOADs and 3 MSTORES. The, 2 variables are only read once through MLOAD (`position.owner` and `position.unlockAt`) and one is read twice (`position.lockAmount`). I suggest replacing the `memory` keyword with `storage` at L69 and only copying `position.lockAmount` in memory.

[G-08] `FungibleAssetVaultForDAO._collateralPriceUsd()` : `oracle` should get cached

See `@audit` tags:

```
    function _collateralPriceUsd() internal view returns (u
        int256 answer = oracle.latestAnswer();  //@audit ga
        uint8 decimals = oracle.decimals();  //@audit gas:

```

[G-09] `FungibleAssetVaultForDAO._collateralPriceUsd()` : return statement should be unchecked

See `@audit` tag:

```
    function _collateralPriceUsd() internal view returns (u
...
        return //@audit gas: whole return statement should
            decimals > 18
                ? uint256(answer) / 10**(decimals - 18)
                : uint256(answer) * 10**(18 - decimals);
    }
```

Due to the ternary condition and the fact that `int256 answer = oracle.latestAnswer();` , the return statement can't underflow and should be unchecked.

[G-10] `FungibleAssetVaultForDAO.deposit()` : `collateralAsset` should get cached

See `@audit` tags:

```
141:     function deposit(uint256 amount) external payable onlyR
142:         require(amount > 0, "invalid_amount");
143:
144:         if (collateralAsset == ETH) {  //@audit gas: SLOAD
145:             require(msg.value == amount, "invalid_msg_value
146:         } else {
147:             require(msg.value == 0, "non_zero_eth_value");
148:             IERC20Upgradeable(collateralAsset).safeTransfer
149:                 msg.sender,
150:                 address(this),
151:                 amount
152:             );
153:         }
```

[G-11] `FungibleAssetVaultForDAO.repay)` : L184 should be unchecked due to L182

See `@audit` tag:

```
179:     function repay(uint256 amount) external onlyRole(WHITEI
180:         require(amount > 0, "invalid_amount");
181:
182:         amount = amount > debtAmount ? debtAmount : amount;
183:
184:         debtAmount -= amount; //@audit gas: should be unche
```

[G-12] `FungibleAssetVaultForDAO.withdraw()` : L196 should be unchecked due to L194

See `@audit` tag:

```
193:     function withdraw(uint256 amount) external onlyRole(WHI
194:         require(amount > 0 && amount <= collateralAmount, '

        uint256 creditLimit = getCreditLimit(collateralAmou
```

[G-13] `FungibleAssetVaultForDAO.withdraw()`: `collateralAmount` should get cached

See `@audit` tags:

```
    function withdraw(uint256 amount) external onlyRole(WHI
        require(amount > 0 && amount <= collateralAmount, '

        uint256 creditLimit = getCreditLimit(collateralAmou
        require(creditLimit >= debtAmount, "insufficient_cr

        collateralAmount -= amount; //@audit gas: SLOAD 3 
```

[G-14] `NFTVault._normalizeAggregatorAnswer()`: return statement should be unchecked

See `@audit` tag:

```
    function _normalizeAggregatorAnswer(IAggregatorV3Interf
        internal
        view
        returns (uint256)
    {
...
        return //@audit gas: whole return statement should
            decimals > 18
                ? uint256(answer) / 10**(decimals - 18)
                : uint256(answer) * 10**(18 - decimals);
    }
```

[G-15] `NFTVault._calculateAdditionalInterest()`: `totalDebtAmount` should get cached

See `@audit` tags:

```
    function _calculateAdditionalInterest() internal view r
...
        if (totalDebtAmount == 0) {  //@audit gas: SLOAD 1
            return 0;
        }

        // Accrue interest
        uint256 interestPerYear = (totalDebtAmount *  //@au
```

[G-16] `NFTVault.sol : struct PositionPreview` can be tightly packed to save 1 storage slot

From (see `@audit` tags):

```
    struct PositionPreview { // @audit gas: can be tightly
        address owner;
        uint256 nftIndex;
        bytes32 nftType;
        uint256 nftValueUSD;
        VaultSettings vaultSettings;
        uint256 creditLimit;
        uint256 debtPrincipal;
        uint256 debtInterest; // @audit gas: 32 bytes
        BorrowType borrowType; // @audit gas: 1 byte (this
        bool liquidatable; // @audit gas: 1 byte
        uint256 liquidatedAt; // @audit gas: 32 bytes
        address liquidator; // @audit gas: 20 bytes
    }
```

To:

```
    struct PositionPreview {
        address owner;
        uint256 nftIndex;
        bytes32 nftType;
        uint256 nftValueUSD;
        VaultSettings vaultSettings;
        uint256 creditLimit;
        uint256 debtPrincipal;
        uint256 debtInterest; // @audit gas: 32 bytes
        uint256 liquidatedAt; // @audit gas: 32 bytes
        BorrowType borrowType; // @audit gas: 1 byte (this enum
        bool liquidatable; // @audit gas: 1 byte
        address liquidator; // @audit gas: 20 bytes
    }
```

[G-17] `NFTVault.showPosition()` : L659 should be unchecked due to L649

See `@audit` tag:

```
File: NFTVault.sol
628:     function showPosition(uint256 _nftIndex)
...
649:         if (debtPrincipal > debtAmount) debtAmount = debtPr
...
659:             debtInterest: debtAmount - debtPrincipal, //@au
```

[G-18] `NFTVault.showPosition()` : `positions[_nftIndex].liquidatedAt` should get cached

See `@audit` tags:

```
File: NFTVault.sol
628:     function showPosition(uint256 _nftIndex)
...
661:             liquidatable: positions[_nftIndex].liquidatedAt
662:                 debtAmount >= _getLiquidationLimit(_nftInde
663:             liquidatedAt: positions[_nftIndex].liquidatedAt
```

[G-19] `NFTVault.showPosition()` : Help the optimizer by saving a storage variable's reference instead of repeatedly fetching it (`positions[_nftIndex]`)

To help the optimizer, declare a `storage` type variable and use it instead of repeatedly fetching the reference in a map or an array.

The effect can be quite significant.

Here, instead of repeatedly calling `positions[_nftIndex]`, save its reference like this: `Position storage _position = positions[_nftIndex]` and use it.

Impacted lines (see `@audit` tags):

```
636:          uint256 debtPrincipal = positions[_nftIndex].debt
637:          uint256 debtAmount = positions[_nftIndex].liquida
638:              ? positions[_nftIndex].debtAmountForRepurchas
641:                  positions[_nftIndex].debtPortion, //@audi
660:              borrowType: positions[_nftIndex].borrowType,
661:              liquidatable: positions[_nftIndex].liquidatec
663:              liquidatedAt: positions[_nftIndex].liquidatec
664:              liquidator: positions[_nftIndex].liquidator /
```

This practice already exists in the solution, as seen in `NFTVault.borrow()`:

```
675:      function borrow(
...
697:          Position storage position = positions[_nftIndex];
```

[G-20] `NFTVault.borrow()` : `totalDebtPortion` should get cached

See `@audit` tags:

```
675:      function borrow(
...
735:          if (totalDebtPortion == 0) {  //@audit gas: SLOAD 1
...
738:          } else {
739:              uint256 plusPortion = (totalDebtPortion * _amou
740:                  totalDebtAmount;
741:              totalDebtPortion += plusPortion; //@audit gas:
```

[G-21] `NFTVault.repay()` : L781 should be unchecked due to ternary operator

See `@audit` tag:

```
756:     function repay(uint256 _nftIndex, uint256 _amount)
...
780:         uint256 paidPrincipal = _amount > debtInterest
781:             ? _amount - debtInterest //@audit gas: should
782:             : 0;
```

[G-22] `NFTVault.repay()` : `totalDebtPortion` and `totalDebtAmount` should get cached

See `@audit` tags:

```
756:     function repay(uint256 _nftIndex, uint256 _amount)
...
784:         uint256 minusPortion = paidPrincipal == debtPrincip
785:             ? position.debtPortion
786:             : (totalDebtPortion * _amount) / totalDebtAmou
787:
788:         totalDebtPortion -= minusPortion; //@audit gas: SL
...
791:         totalDebtAmount -= _amount; //@audit gas: SLOAD 2
```

[G-23] `Controller.setStrategy()` : boolean comparison L87

Comparing to a constant (`true` or `false`) is a bit more expensive than directly checking the returned boolean value. I suggest using `if(directValue)` instead of `if(directValue == true)` and `if(!directValue)` instead of `if(directValue == false)` here (see `@audit` tag):

```
82:     function setStrategy(IERC20 _token, IStrategy _strategy)
83:         external
84:         onlyRole(STRATEGIST_ROLE)
```

```
    {
        require(
            approvedStrategies[_token][_strategy] == true, /
```

[G-24] `StrategyPUSDConvex.balanceOfJPEG()`: `jpeg` should get cached

See `@audit` tags:

```
    function balanceOfJPEG() external view returns (uin
        uint256 availableBalance = jpeg.balanceOf(addre
...
        for (uint256 i = 0; i < length; i++) {
...
            if (address(jpeg) == extraReward.rewardToke
```

[G-25] `StrategyPUSDConvex.balanceOfJPEG()`: use a `return` statement instead of `break`

See `@audit` tag:

```
    function balanceOfJPEG() external view returns (uint256
...
        for (uint256 i = 0; i < length; i++) {
...
            if (address(jpeg) == extraReward.rewardToken())
                availableBalance += extraReward.earned();
                //we found jpeg, no need to continue the lo
                break; //@audit gas: instead of adding to a
            }
        }

        return availableBalance;
    }
```

Here, instead of making a memory operation with `availableBalance += extraReward.earned();` and then using `break;` before returning the memory variable `availableBalance`, it would’ve been more optimized to just return `availableBalance + extraReward.earned()`:

```
    function balanceOfJPEG() external view returns (uint256) {
...
        for (uint256 i = 0; i < length; i++) {
...
            if (address(jpeg) == extraReward.rewardToken()) {
              return availableBalance + extraReward.earned();
            }
        }
    }
```

[G-26] `StrategyPUSDConvex.withdraw()` : L281 should be unchecked due to L279

See `@audit` tag:

```
273:     function withdraw(uint256 _amount) external onlyControl
...
279:         if (balance < _amount)
280:             convexConfig.baseRewardPool.withdrawAndUnwrap(
281:                 _amount - balance, //@audit gas: should be
282:                 false
283:             );
```

[G-27] `StrategyPUSDConvex.harvest()` : L362 should be unchecked due to L359-L360 and how performanceFee is set L183

See `@audit` tags:

```
177:     function setPerformanceFee(Rate memory _performanceFee)
...
181:         require(
182:             _performanceFee.denominator > 0 &&
183:                 _performanceFee.denominator >= _performance
184:             "INVALID_RATE"
```

```
185:        );
...
311:    function harvest(uint256 minOutCurve) external onlyRole
...
359:        uint256 fee = (usdcBalance * performanceFee.numerat
360:            performanceFee.denominator;
361:        usdc.safeTransfer(strategy.controller.feeAddress(),
362:        usdcBalance -= fee; //@audit gas: should be uncheck
```

[G-28] `yVault.earn()` : `token` and `controller` should get cached

See `@audit` tags:

```
129:    function earn() external {
130:        uint256 _bal = available();
131:        token.safeTransfer(address(controller), _bal); //@a
132:        controller.earn(address(token), _bal); //@audit gas
133:    }
```

[G-29] `yVault.withdraw()` : L178 should be unchecked due to L177

See `@audit` tag:

```
166:    function withdraw(uint256 _shares) public noContract(ms
...
177:        if (vaultBalance < backingTokens) {
178:            uint256 toWithdraw = backingTokens - vaultBalar
```

[G-30] `yVault.withdraw()` : `token` should get cached

See `@audit` tags:

```
166:    function withdraw(uint256 _shares) public noContract(ms
...
176:        uint256 vaultBalance = token.balanceOf(address(this
        if (vaultBalance < backingTokens) {
            uint256 toWithdraw = backingTokens - vaultBalan
            controller.withdraw(address(token), toWithdraw)
        }

        token.safeTransfer(msg.sender, backingTokens);  //@
```

[G-31] `yVault.withdrawJPEG()` : `farm` should get cached

See `@audit` tags:

```
    function withdrawJPEG() external {
        require(farm != address(0), "NO_FARM");  //@audit g
        controller.withdrawJPEG(address(token), farm);  //@
    }
```

[G-32] Upgrade pragma to at least 0.8.4

Across the whole solution, the declared pragma is `^0.8.0`.

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

The advantages here are:

- **Low level inliner** (>= 0.8.2): Cheaper runtime gas (especially relevant when the contract has small functions).
- **Optimizer improvements in packed structs** (>= 0.8.3)
- **Custom errors** (>= 0.8.4): cheaper deployment cost and runtime cost. *Note*: the runtime cost is only relevant when the revert condition is met. In short, replace revert strings by custom errors.

Consider upgrading pragma to at least 0.8.4:

[G-33] No need to explicitly initialize variables with default values

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) {`

Instances include:

```
farming/LPFarming.sol:281:        for (uint256 pid = 0; pid < le
farming/LPFarming.sol:348:        for (uint256 i = 0; i < poolIr
vaults/yVault/strategies/StrategyPUSDConvex.sol:145:        for
vaults/yVault/strategies/StrategyPUSDConvex.sol:231:        for
vaults/yVault/strategies/StrategyPUSDConvex.sol:319:
vaults/FungibleAssetVaultForDAO.sol:45:    address internal cons
vaults/NFTVault.sol:181:        for (uint256 i = 0; i < _typeIni
vaults/NFTVault.sol:184:            for (uint256 j = 0; j < init
```

I suggest removing explicit initializations for default values.

[G-34] `> 0` is less efficient than `!= 0` for unsigned integers (with proof)

`!= 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 at 10k 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

I suggest changing `> 0` with `!= 0` here:

```
farming/LPFarming.sol:114:        require(_rewardPerBlock > 0, '
farming/LPFarming.sol:218:        require(_amount > 0, "invalid_
farming/LPFarming.sol:239:        require(_amount > 0, "invalid_
farming/LPFarming.sol:337:        require(rewards > 0, "no_rewar
farming/LPFarming.sol:354:        require(rewards > 0, "no_rewar
farming/yVaultLPFarming.sol:101:        require(_amount > 0, "ir
farming/yVaultLPFarming.sol:118:        require(_amount > 0, "ir
farming/yVaultLPFarming.sol:139:        require(rewards > 0, "nc
lock/JPEGLock.sol:40:        require(_newTime > 0, "Invalid lock
staking/JPEGStaking.sol:32:        require(_amount > 0, "invalic
vaults/yVault/strategies/StrategyPUSDConvex.sol:182:
vaults/yVault/strategies/StrategyPUSDConvex.sol:334:
vaults/yVault/yVault.sol<img class="emoji-icon" alt="emoji-100"
vaults/yVault/yVault.sol:143:        require(_amount > 0, "INVAI
vaults/yVault/yVault.sol:167:        require(_shares > 0, "INVAI
vaults/yVault/yVault.sol:170:        require(supply > 0, "NO_TOK
vaults/FungibleAssetVaultForDAO.sol:94:            _creditLimitF
vaults/FungibleAssetVaultForDAO.sol:108:        require(answer >
vaults/FungibleAssetVaultForDAO.sol:142:        require(amount >
vaults/FungibleAssetVaultForDAO.sol:164:        require(amount >
vaults/FungibleAssetVaultForDAO.sol:180:        require(amount >
vaults/FungibleAssetVaultForDAO.sol:194:        require(amount >
vaults/NFTVault.sol:278:        require(_newFloor > 0, "Invalid
vaults/NFTVault.sol:327:            _type == bytes32(0) || nftTy
vaults/NFTVault.sol:365:        require(pendingValue > 0, "no_pe
vaults/NFTVault.sol:402:            rate.denominator > 0 && rate
vaults/NFTVault.sol:462:        require(answer > 0, "invalid_ora
vaults/NFTVault.sol:687:        require(_amount > 0, "invalid_am
vaults/NFTVault.sol:764:        require(_amount > 0, "invalid_am
vaults/NFTVault.sol:770:        require(debtAmount > 0, "positic
vaults/NFTVault.sol:882:        require(position.liquidatedAt >
vaults/NFTVault.sol:926:        require(position.liquidatedAt >
```

Also, please enable the Optimizer.

[G-35] `>=` is cheaper than `>`

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

I suggest using `>=` instead of `>` to avoid some opcodes here:

```
vaults/NFTVault.sol:539:        return principal > calculatedDeb
vaults/NFTVault.sol:775:        _amount = _amount > debtAmount ?
```

[G-36] An array's length should be cached to save gas in for-loops

Reading array length at each iteration of the loop takes 6 gas (3 for mload and 3 to place memory_offset) in the stack.

Caching the array length in the stack saves around 3 gas per iteration.

Here, I suggest storing the array's length in a variable before the for-loop, and use it instead:

```
farming/LPFarming.sol:348:        for (uint256 i = 0; i < poolIr
vaults/yVault/strategies/StrategyPUSDConvex.sol:145:        for
vaults/yVault/strategies/StrategyPUSDConvex.sol:319:
vaults/NFTVault.sol:181:        for (uint256 i = 0; i < _typeIni
vaults/NFTVault.sol:184:            for (uint256 j = 0; j < init
```

This is already done here:

```
farming/LPFarming.sol:281:        for (uint256 pid = 0; pid < le
vaults/yVault/strategies/StrategyPUSDConvex.sol:231:        for
```

[G-37] `++i` costs less gas compared to `i++` or `i += 1`

`++i` costs less gas compared to `i++` or `i += 1` for unsigned integer, as pre-increment is cheaper (about 5 gas per iteration). This statement is true even with the optimizer enabled.

`i++` increments `i` and returns the initial value of `i`. Which means:

```
uint i = 1;
i++; // == 1 but i == 2
```

But `++i` returns the actual incremented value:

```
    uint i = 1;
    ++i; // == 2 and i == 2 too, so no need for a temporary variable
```

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

Instances include:

```
    farming/LPFarming.sol:348:        for (uint256 i = 0; i < poolIr
    vaults/yVault/strategies/StrategyPUSDConvex.sol:145:        for
    vaults/yVault/strategies/StrategyPUSDConvex.sol:231:        for
    vaults/yVault/strategies/StrategyPUSDConvex.sol:319:
    vaults/NFTVault.sol:181:        for (uint256 i = 0; i < _typeIni
    vaults/NFTVault.sol:184:            for (uint256 j = 0; j < init
```

I suggest using `++i` instead of `i++` to increment the value of an uint variable.

This is already done here:

```
    farming/LPFarming.sol:281:        for (uint256 pid = 0; pid < le
```

[G-38] Increments can be unchecked

In Solidity 0.8+, there's a default overflow check on unsigned integers. It's possible to uncheck this in for-loops and save some gas at each iteration, but at the cost of some code readability, as this uncheck cannot be made inline.

[ethereum/solidity#10695]()

Instances include:

```
    farming/LPFarming.sol:281:        for (uint256 pid = 0; pid < le
    farming/LPFarming.sol:348:        for (uint256 i = 0; i < poolIr
    vaults/yVault/strategies/StrategyPUSDConvex.sol:145:        for
    vaults/yVault/strategies/StrategyPUSDConvex.sol:231:        for
    vaults/yVault/strategies/StrategyPUSDConvex.sol:319:
```

```
vaults/NFTVault.sol:181:        for (uint256 i = 0; i < _typeIni
vaults/NFTVault.sol:184:            for (uint256 j = 0; j < init
```

The code would go from:

```
for (uint256 i; i < numIterations; i++) {
 // ...
}
```

to:

```
for (uint256 i; i < numIterations;) {
 // ...
 unchecked { ++i; }
}
```

The risk of overflow is inexistant for a `uint256` here.

[G-39] Use `calldata` instead of `memory`

When arguments are read-only on external functions, the data location should be `calldata`:

```
contracts/vaults/NFTVault.sol:
  212:     function setDebtInterestApr(Rate memory _debtInterest
  222:     function setValueIncreaseLockRate(Rate memory _valueI
  232:     function setCreditLimitRate(Rate memory _creditLimitR
  247:     function setLiquidationLimitRate(Rate memory _liquida
  290:     function setOrganizationFeeRate(Rate memory _organiza
  300:     function setInsurancePurchaseRate(Rate memory _insura
  311:         Rate memory _insuranceLiquidationPenaltyRate  //(
```

[G-40] Consider making some constants as non-public to save gas

Reducing from `public` to `private` or `internal` can save gas when a constant isn't used outside of its contract. I suggest changing the visibility from `public` to `internal` or `private` here:

```
tokens/JPEG.sol:10:    bytes32 public constant MINTER_ROLE = kec
tokens/StableCoin.sol:22:    bytes32 public constant MINTER_ROLE
tokens/StableCoin.sol:23:    bytes32 public constant PAUSER_ROLE
vaults/yVault/strategies/StrategyPUSDConvex.sol:66:    bytes32 p
vaults/yVault/Controller.sol:15:    bytes32 public constant STRA
vaults/FungibleAssetVaultForDAO.sol:41:    bytes32 public consta
vaults/NFTVault.sol:71:    bytes32 public constant DAO_ROLE = ke
vaults/NFTVault.sol:72:    bytes32 public constant LIQUIDATOR_RO
vaults/NFTVault.sol:74:    bytes32 public constant CUSTOM_NFT_HA
```

[G-41] Public functions to external

The following functions could be set external to save gas and improve code quality. External call cost is less expensive than of public functions.

```
withdraw(IERC20,uint256) should be declared external:
 - Controller.withdraw(IERC20,uint256) (contracts/vaults/yVault/
setFarmingPool(address) should be declared external:
 - YVault.setFarmingPool(address) (contracts/vaults/yVault/yVaul
```

[G-42] Reduce the size of error messages (Long revert Strings)

Shortening revert strings to fit in 32 bytes will decrease deployment time gas and will decrease runtime gas when the revert condition is met.

Revert strings that are longer than 32 bytes require at least one additional mstore, along with additional overhead for computing memory offset, etc.

Revert strings > 32 bytes:

```
tokens/JPEG.sol:23:            "JPEG: must have minter role to m
tokens/StableCoin.sol:41:            "StableCoin: must have mint
```

```
    tokens/StableCoin.sol:55:               "StableCoin: must have paus
    tokens/StableCoin.sol:69:               "StableCoin: must have paus
    vaults/NFTVault.sol:394:               "credit_rate_exceeds_or_equa
```

I suggest shortening the revert strings to fit in 32 bytes, or using custom errors as described next.

[G-43] Use Custom Errors instead of Revert Strings to save Gas

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).

See original submission for instances.

spaghettieth (JPEG'd) confirmed and commented:

> Very high quality report, may implement some of your suggestions. Thank you!

spaghettieth (JPEG'd) resolved and commented:

> Implemented most of your suggestions in jpegd/core#21. Custom errors have not been implemented as it would be too big of a change at this point and doing it properly may cause unexpected behaviour/bugs due to the codebase changing too much.

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