NFTX contest
Findings & Analysis Report

2022-03-24

Table of contents

Overview

About C4

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

A C4 code contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the code contest outlined in this document, C4 conducted an analysis of the NFTX smart contract system written in Solidity. The code contest took place between December 16—December 22 2021.

Wardens

27 Wardens contributed reports to the NFTX contest:

1. cccz
2. hyh
3. leastwood
4. WatchPug (jtp and ming)
5. cmichel
6. robee
7. GreyArt (hickuphh3 and itsmeSTYJ)
8. pauliax
9. gzeon
10. Ruhum
11. ych18
12. csanuragjain
13. pedroais
14. jayjonah8
15. sirhashalot
16. 0x1f8b
17. PPrieditis
18. 0x0x0x
19. p4st13r4 (0x69e8 and 0xb4bb4)
20. defsec

21. Dravee
22. shenwilly
23. BouSalman
24. saian

This contest was judged by LSDan.

Final report assembled by liveactionllama and itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 58 unique vulnerabilities and 124 total findings. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 3 received a risk rating in the category of HIGH severity, 17 received a risk rating in the category of MEDIUM severity, and 38 received a risk rating in the category of LOW severity.

C4 analysis also identified 19 non-critical recommendations and 47 gas optimizations.

Scope

The code under review can be found within the C4 NFTX contest repository, and is composed of 11 smart contracts written in the Solidity programming language and includes 3072 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on [the C4 website](#).

High Risk Findings (3)

[H-01] buyAndSwap1155WETH() function may cause loss of user assets

Submitted by cccz

In the NFTXMarketplaceZap.sol contract, the buyAndSwap1155WETH function uses the WETH provided by the user to exchange VaultToken, but when executing the _buyVaultToken method, msg.value is used instead of maxWethIn. Since msg.value is 0, the call will fail.

```
function buyAndSwap1155WETH(
  uint256 vaultId,
  uint256[] memory idsIn,
  uint256[] memory amounts,
  uint256[] memory specificIds,
  uint256 maxWethIn,
  address[] calldata path,
  address to
) public payable nonReentrant {
  require(to != address(0));
  require(idsIn.length != 0);
  IERC20Upgradeable(address(WETH)).transferFrom(msg.sender, addr
  uint256 count;
  for (uint256 i = 0; i <idsIn.length; i++) {
      uint256 amount = amounts[i];
      require(amount> 0, "Transferring <1");
      count += amount;
  }
  INFTXVault vault = INFTXVault(nftxFactory.vault(vaultId));
```

```
    uint256 redeemFees = (vault.targetSwapFee() * specificIds.leng
        vault.randomSwapFee() * (count-specificIds.length)
    );
    uint256[] memory swapAmounts = _buyVaultToken(address(vault),
```

In extreme cases, when the user provides both ETH and WETH (the user approves the contract WETH in advance and calls the buyAndSwap1155WETH function instead of the buyAndSwap1155 function by mistake), the _buyVaultToken function will execute successfully, but because the buyAndSwap1155WETH function will not convert ETH to WETH, The user's ETH will be locked in the contract, causing loss of user assets.

```
function _buyVaultToken(
  address vault,
  uint256 minTokenOut,
  uint256 maxWethIn,
  address[] calldata path
) internal returns (uint256[] memory) {
  uint256[] memory amounts = sushiRouter.swapTokensForExactToker
    minTokenOut,
    maxWethIn,
    path,
    address(this),
    block.timestamp
  );

  return amounts;
}
```

Recommended Mitigation Steps

```
- uint256[] memory swapAmounts = _buyVaultToken(address(vault)
+ uint256[] memory swapAmounts = _buyVaultToken(address(vault)
```

[0xKiwi (NFTX) confirmed and resolved]

[H-02] The return value of the _sendForReceiver function is not set, causing the receiver to receive more fees

Submitted by cccz, also found by WatchPug

In the NFTXSimpleFeeDistributor.sol contract, the distribute function is used to distribute the fee, and the distribute function judges whether the fee is sent successfully according to the return value of the _sendForReceiver function.

```
function distribute(uint256 vaultId) external override virtual r
  require(nftxVaultFactory != address(0));
  address _vault = INFTXVaultFactory(nftxVaultFactory).vault(vau

  uint256 tokenBalance = IERC20Upgradeable(_vault).balanceOf(adc

  if (distributionPaused || allocTotal == 0) {
    IERC20Upgradeable(_vault).safeTransfer(treasury, tokenBalanc
    return;
  }

  uint256 length = feeReceivers.length;
  uint256 leftover;
  for (uint256 i = 0; i <length; i++) {
    FeeReceiver memory _feeReceiver = feeReceivers[i];
    uint256 amountToSend = leftover + ((tokenBalance * _feeRecei
    uint256 currentTokenBalance = IERC20Upgradeable(_vault).bala
    amountToSend = amountToSend> currentTokenBalance? currentTok
    bool complete = _sendForReceiver(_feeReceiver, vaultId, _vau
    if (!complete) {
      leftover = amountToSend;
    } else {
      leftover = 0;
    }
  }
```

In the _sendForReceiver function, when _receiver is not a contract, no value is returned. By default, this will return false. This will make the distribute function think that the fee sending has failed, and will send more fees next time.

```
function _sendForReceiver(FeeReceiver memory _receiver, uint256
  if (_receiver.isContract) {
```

```
      IERC20Upgradeable(_vault).approve(_receiver.receiver, amount
      // If the receive is not properly processed, send it to the

      bytes memory payload = abi.encodeWithSelector(INFTXLPStaking
      (bool success,) = address(_receiver.receiver).call(payload);

      // If the allowance has not been spent, it means we can pass
      return success && IERC20Upgradeable(_vault).allowance(addres
    } else {
      IERC20Upgradeable(_vault).safeTransfer(_receiver.receiver, a
    }
  }
```

Proof of Concept

https://github.com/code-423n4/2021-12-nftx/blob/main/nftx-protocol-v2/contracts/solidity/NFTXSimpleFeeDistributor.sol#L157-L168

https://github.com/code-423n4/2021-12-nftx/blob/main/nftx-protocol-v2/contracts/solidity/NFTXSimpleFeeDistributor.sol#L49-L67

Recommended Mitigation Steps

```
  function _sendForReceiver(FeeReceiver memory _receiver, uint256
    if (_receiver.isContract) {
      IERC20Upgradeable(_vault).approve(_receiver.receiver, amount
      // If the receive is not properly processed, send it to the

      bytes memory payload = abi.encodeWithSelector(INFTXLPStaking
      (bool success, ) = address(_receiver.receiver).call(payload)

      // If the allowance has not been spent, it means we can pass
      return success && IERC20Upgradeable(_vault).allowance(addres
    } else {
      - IERC20Upgradeable(_vault).safeTransfer(_receiver.receiver,
      + return IERC20Upgradeable(_vault).safeTransfer(_receiver.re
    }
  }
```

0xKiwi (NFTX) confirmed, but disagreed with high severity and commented:

> Good catch, thank you. Disagreeing with severity though since this is a permissioned contract, no user funds are at risk and this would most likely cause some failures.

> We aren't using any EOAs as receivers in production or testing, so this has not been caught. Thank you.

0xKiwi (NFTX) resolved

LSDan (judge) commented:

> I agree with the warden on this one. Funds are directly at risk and the likelihood of this occurring is 100%. I'm not sure if it matters if the funds are user funds or protocol funds. This would eventually have become a big problem that affected the protocol's ability to function.

[H-03] A vault can be locked from MarketplaceZap and StakingZap

Submitted by p4st13r4, also found by cmichel, GreyArt, hyh, jayjonah8, leastwood, pauliax, shenwilly, and WatchPug

Any user that owns a vToken of a particular vault can lock the functionalities of `NFTXMarketplaceZap.sol` and `NFTXStakingZap.sol` for everyone.

Every operation performed by the marketplace, that deals with vToken minting, performs this check:

```
require(balance == IERC20Upgradeable(vault).balanceOf(address(th
```

A malicious user could transfer any amount > 0 of a vault'vToken to the marketplace (or staking) zap contracts, thus making the vault functionality unavailable for every user on the marketplace

Proof of Concept

https://github.com/code-423n4/2021-12-nftx/blob/main/nftx-protocol-v2/contracts/solidity/NFTXMarketplaceZap.sol#L421

https://github.com/code-423n4/2021-12-nftx/blob/main/nftx-protocol-v2/contracts/solidity/NFTXMarketplaceZap.sol#L421

Recommended Mitigation Steps

Remove this logic from the marketplace and staking zap contracts, and add it to the vaults (if necessary)

0xKiwi (NFTX) confirmed, but disagreed with high severity and commented:

> Valid concern, confirmed. And disagreeing with severity.

0xKiwi (NFTX) resolved

LSDan (judge) commented:

> In this case I agree with the warden's severity. The attack would cause user funds to be locked and is incredibly easy to perform.

Medium Risk Findings (17)

[M-01] Missing non reentrancy modifier

Submitted by robee

The following functions are missing reentrancy modifier although some other pulbic/external functions does use reentrancy modifer. Even though I did not find a way to exploit it, it seems like those functions should have the nonReentrant modifier as the other functions have it as well..

```
NFTXMarketplaceZap.sol, receive is missing a reentrancy modifi
NFTXSimpleFeeDistributor.sol, __SimpleFeeDistributor__init__ i
NFTXSimpleFeeDistributor.sol, addReceiver is missing a reentra
NFTXSimpleFeeDistributor.sol, initializeVaultReceivers is miss
```

```
NFTXSimpleFeeDistributor.sol, changeReceiverAlloc is missing a
NFTXSimpleFeeDistributor.sol, changeReceiverAddress is missing
NFTXSimpleFeeDistributor.sol, removeReceiver is missing a reer
NFTXSimpleFeeDistributor.sol, setTreasuryAddress is missing a
NFTXSimpleFeeDistributor.sol, setLPStakingAddress is missing a
NFTXSimpleFeeDistributor.sol, setInventoryStakingAddress is mi
NFTXSimpleFeeDistributor.sol, setNFTXVaultFactory is missing a
NFTXSimpleFeeDistributor.sol, pauseFeeDistribution is missing
NFTXSimpleFeeDistributor.sol, rescueTokens is missing a reentr
NFTXStakingZap.sol, setLPLockTime is missing a reentrancy modi
NFTXStakingZap.sol, setInventoryLockTime is missing a reentrar
NFTXStakingZap.sol, provideInventory721 is missing a reentranc
NFTXStakingZap.sol, provideInventory1155 is missing a reentrar
NFTXStakingZap.sol, addLiquidity721ETH is missing a reentrancy
NFTXStakingZap.sol, addLiquidity1155ETH is missing a reentranc
NFTXStakingZap.sol, addLiquidity721 is missing a reentrancy mo
NFTXStakingZap.sol, addLiquidity1155 is missing a reentrancy m
NFTXStakingZap.sol, receive is missing a reentrancy modifier
NFTXStakingZap.sol, rescue is missing a reentrancy modifier
NFTXV1Buyout.sol, __NFTXV1Buyout_init is missing a reentrancy
NFTXV1Buyout.sol, emergencyWithdraw is missing a reentrancy mo
NFTXV1Buyout.sol, clearBuyout is missing a reentrancy modifier
NFTXV1Buyout.sol, addBuyout is missing a reentrancy modifier
NFTXV1Buyout.sol, removeBuyout is missing a reentrancy modifie
NFTXVaultUpgradeable.sol, __NFTXVault_init is missing a reentr
NFTXVaultUpgradeable.sol, finalizeVault is missing a reentranc
NFTXVaultUpgradeable.sol, setVaultMetadata is missing a reentr
NFTXVaultUpgradeable.sol, setVaultFeatures is missing a reentr
NFTXVaultUpgradeable.sol, assignDefaultFeatures is missing a r
NFTXVaultUpgradeable.sol, setFees is missing a reentrancy modi
NFTXVaultUpgradeable.sol, disableVaultFees is missing a reentr
NFTXVaultUpgradeable.sol, deployEligibilityStorage is missing
NFTXVaultUpgradeable.sol, setManager is missing a reentrancy m
NFTXVaultUpgradeable.sol, mint is missing a reentrancy modifie
NFTXVaultUpgradeable.sol, redeem is missing a reentrancy modif
NFTXVaultUpgradeable.sol, swap is missing a reentrancy modifie
NFTXVaultUpgradeable.sol, flashLoan is missing a reentrancy mo
PalmNFTXStakingZap.sol, setLockTime is missing a reentrancy mo
PalmNFTXStakingZap.sol, addLiquidity721 is missing a reentranc
PalmNFTXStakingZap.sol, addLiquidity1155 is missing a reentrar
PalmNFTXStakingZap.sol, receive is missing a reentrancy modifi
```

[0xKiwi (NFTX) disputed]

LSDan (judge) increased severity to medium and commented:

> I'm updating this [from a low] to a medium. Reentrancy represents a real and significant risk (as evident by ETC existing) and should be protected against regardless of if you can foresee the external event that causes lack of protection to be an issue.

> `2 — Med (M): vulns have a risk of 2 and are considered "Medium" severity when assets are 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.`

[M-02] NFTXSimpleFeeDistributor#addReceiver: Failure to check for existing receiver

Submitted by GreyArt

The `addReceiver()` function fails to check if the `_receiver` already exists. This could lead to the same receiver being added multiple times, which results in erroneous fee distributions.

The receiver would receive more than expected (until the duplicate entry has been removed).

Recommended Mitigation Steps

Have a mapping `address => bool isReceiver` that will update whenever receivers are added, modified to a new address or removed.

0xKiwi (NFTX) acknowledged, but disagreed with medium severity and commented:

> Valid concern, but this is a permissioned function.

LSDan (judge) commented:

> I think this one is much more likely. I would suggest adding a check because the problem is easy to create and much harder to notice.

[M-03] `NFTXMarketplaceZap.sol#buyAnd***()` should return unused weth/eth back to `msg.sender` instead of `to`

Submitted by WatchPug

https://github.com/code-423n4/2021-12-nftx/blob/194073f750b7e2c9a886ece34b6382b4f1355f36/nftx-protocol-v2/contracts/solidity/NFTXMarketplaceZap.sol#L226-L249

```
function buyAndSwap721WETH(
  uint256 vaultId,
  uint256[] memory idsIn,
  uint256[] memory specificIds,
  uint256 maxWethIn,
  address[] calldata path,
  address to
) public nonReentrant {
  require(to != address(0));
  require(idsIn.length != 0);
  IERC20Upgradeable(address(WETH)).transferFrom(msg.sender, addr
  INFTXVault vault = INFTXVault(nftxFactory.vault(vaultId));
  uint256 redeemFees = (vault.targetSwapFee() * specificIds.leng
      vault.randomSwapFee() * (idsIn.length - specificIds.length
  );
  uint256[] memory amounts = _buyVaultToken(address(vault), rede
  _swap721(vaultId, idsIn, specificIds, to);

  emit Swap(idsIn.length, amounts[0], to);

  // Return extras.
  uint256 remaining = WETH.balanceOf(address(this));
  WETH.transfer(to, remaining);
}
```

For example:

If Alice calls `buyAndSwap721WETH()` to buy some ERC721 and send to Bob, for slippage control, Alice put `1000 ETH` as `maxWethIn`, the actual cost should be lower.

Let's say the actual cost is `900 ETH`.

Expected Results: Alice spend only for the amount of the actual cost (`900 ETH`).

Actual Results: Alice spent `1000 ETH`.

0xKiwi (NFTX) acknowledged, but disagreed with medium severity and commented:

> I think the idea in this is that if a contract is buying for someone else, the zap handles the refund instead of the contract originating the purchase. But this is a valid concern, thank you

LSDan (judge) commented:

> This does result in a loss of funds if the user sends the wrong amount. I agree with the warden's severity rating.

[M-04] NFTXStakingZap and NFTXMarketplaceZap's transferFromERC721 transfer Cryptokitties to the wrong address

Submitted by hyh

`transferFromERC721(address assetAddr, uint256 tokenId, address to)` should transfer from `msg.sender` to `to`. It transfers to `address(this)` instead when ERC721 is Cryptokitties. As there is no additional logic for this case it seems to be a mistake that leads to wrong NFT accounting after such a transfer as NFT will be missed in the vault (which is `to`).

Proof of Concept

NFTXStakingZap: transferFromERC721 https://github.com/code-423n4/2021-12-nftx/blob/main/nftx-protocol-v2/contracts/solidity/NFTXStakingZap.sol#L416

NFTXMarketplaceZap: transferFromERC721 https://github.com/code-423n4/2021-12-nftx/blob/main/nftx-protocol-v2/contracts/solidity/NFTXMarketplaceZap.sol#L556

Both functions are called by user facing Marketplace buy/sell and Staking addLiquidity/provideInventory functions.

Recommended Mitigation Steps

Fix the address:

Now:

```
// Cryptokitties.
data = abi.encodeWithSignature("transferFrom(address,address,u
```

To be:

```
// Cryptokitties.
data = abi.encodeWithSignature("transferFrom(address,address,u
```

0xKiwi (NFTX) confirmed, but disagreed with medium severity and commented:

> This was intentional, as I thought it was needed for the contract to require custody, but it should work fine to send directly to the vault.

0xKiwi (NFTX) resolved

[M-05] Pool Manager can frontrun fees to 100% and use it to steal the value from users

Submitted by pedroais

Pool Manager can front-run entry fee to 100% and users could lose all their deposits.

Proof of Concept

Considering:
The pool manager is the creator of the pool.
Anyone can create a pool.
Manager is not a trusted actor.

Anyone can create a pool and get people to join. If there is a big deposit admin could front-run the transaction and set the fee to max which is uint(1 ether) = 10**18 (100% as this is a per token fee).

Function that set fees : https://github.com/code-423n4/2021-12-nftx/blob/194073f750b7e2c9a886ece34b6382b4f1355f36/nftx-protocol-v2/contracts/solidity/NFTXVaultUpgradeable.sol#L119 Max fees are 1 ether : https://github.com/code-423n4/2021-12-nftx/blob/194073f750b7e2c9a886ece34b6382b4f1355f36/nftx-protocol-v2/contracts/solidity/NFTXVaultFactoryUpgradeable.sol#L122

The manager could benefit from this by having other pool assets deposited in staking so he would receive fees in Vtokens and could use them to withdraw the nfts.

Recommended Mitigation Steps

Add a timelock to change fees. In that way, frontrunning wouldn't be possible and users would know the fees they are agreeing with.

0xKiwi (NFTX) acknowledged, but disagreed with high severity and commented:

> Most users aren't on vaults that aren't finalized. We warn users for any vaults that arent finalized and we don't present them on our website. Acknowledging and disagreeing with severity.

LSDan (judge) decreased severity to medium and commented:

> In my view, this is a medium risk. While user funds are at direct risk, the likelihood of this happening or being worth the effort is low. As the sponsor states, it's very

> rare for a user to interact with an un-finalized vault. The user would have to be directly linked to the vault and then ignore the giant warning presented front and center in the UI. If that warning were to be removed, however, the risk would increase. This external requirement is the only reason I'm going with medium and not low.

[M-06] `xToken` Approvals Allow Spenders To Spend More Tokens

Submitted by leastwood

The `approve` function has not been overridden and therefore uses `xToken` shares instead of the equivalent rebalanced amount, i.e. the underlying vault token amount.

Proof of Concept

The approved spender may spend more tokens than desired. In fact, the approved amount that can be transferred keeps growing as rewards continue to be distributed to the `XTokenUpgradeable` contract.

Many contracts also use the same amount for the `approve` call as for the amount they want to have transferred in a subsequent `transferFrom` call, and in this case, they approve an amount that is too large (as the approved `shares` amount yields a higher rebalanced amount).

Recommended Mitigation Steps

The `_allowances` field should track the rebalanced amounts (i.e. the equivalent vault token amount) such that the approval value does not grow.

The `transferFrom` needs to be overridden and approvals should then be subtracted by the transferred vault token `amount`, not `shares`.

0xKiwi (NFTX) acknowledged, but disagreed with high severity and commented:

> Not sure if I agree with this severity. If I approve for xTokens, I'm using xTokens, not the underlying token.

LSDan (judge) decreased severity to medium and commented:

> This is a medium risk, not high. External assumptions (malicious contracts) are requires for any attack regarding the approval being too high.

[M-07] Rewards can be stolen

Submitted by cmichel

The `NFTXInventoryStaking` contract distributes new rewards to all previous stakers when the owner calls the `receiveRewards` function. This allows an attacker to frontrun this `receiveRewards` transaction when they see it in the mem pool with a `deposit` function. The attacker will receive the rewards pro-rata to their deposits. The deposit will be locked for 2 seconds only (`DEFAULT_LOCKTIME`) after which the depositor can withdraw their initial deposit & the rewards again for a profit.

The rewards can be gamed this way and one does not actually have to *stake*, only be in the staking contract at the time of reward distribution for 2 seconds. The rest of the time they can be used for other purposes.

Recommended Mitigation Steps

Distribute the rewards equally over time to the stakers instead of in a single chunk on each `receiveRewards` call. This is more of a "streaming rewards" approach.

0xKiwi (NFTX) confirmed and commented:

> Thanks for the report.

> This is unfortunately unavoidable but streaming isn't a bad idea. Will consider. Thank you.

> Confirming.

[M-08] Low-level call return value not checked

Submitted by cmichel

The `NFTXStakingZap.addLiquidity721ETHTo` function performs a low-level `.call` in `payable(to).call{value: msg.value-amountEth}` but does not check the return value if the call succeeded.

Impact

If the call fails, the refunds did not succeed and the caller will lose all refunds of `msg.value - amountEth`.

Recommended Mitigation Steps

Revert the entire transaction if the refund call fails by checking that the `success` return value of the `payable(to).call(...)` returns `true`.

0xKiwi (NFTX) confirmed and commented:

> Nice catch, thank you. Confirmed.

0xKiwi (NFTX) resolved

[M-09] Bypass zap timelock

Submitted by gzeon

The default value of `inventoryLockTime` in `NFTXStakingZap` is `7 days` while `DEFAULT_LOCKTIME` in `NFTXInventoryStaking` is 2 ms. These timelock value are used in `NFTXInventoryStaking` to eventually call `_timelockMint` in `XTokenUpgradeable`.

https://github.com/code-423n4/2021-12-nftx/blob/194073f750b7e2c9a886ece34b6382b4f1355f36/nftx-protocol-v2/contracts/solidity/token/XTokenUpgradeable.sol#L74

```
    function _timelockMint(address account, uint256 amount, uint256
      uint256 timelockFinish = block.timestamp + timelockLength;
      timelock[account] = timelockFinish;
      emit Timelocked(account, timelockFinish);
      _mint(account, amount);
```

```
}
```

The applicable timelock is calculated by `block.timestamp + timelockLength`, even when the existing timelock is further in the future. Therefore, one can reduce their long (e.g. 7 days) timelock to 2 ms calling `deposit` in `NFTXInventoryStaking`

Proof of Concept

https://github.com/code-423n4/2021-12-nftx/blob/194073f750b7e2c9a886ece34b6382b4f1355f36/nftx-protocol-v2/contracts/solidity/NFTXStakingZap.sol#L160 https://github.com/code-423n4/2021-12-nftx/blob/194073f750b7e2c9a886ece34b6382b4f1355f36/nftx-protocol-v2/contracts/solidity/NFTXInventoryStaking.sol#L30

Recommended Mitigation Steps

```
function _timelockMint(address account, uint256 amount, uint256
  uint256 timelockFinish = block.timestamp + timelockLength;
  if(timelockFinish > timelock[account]){
    timelock[account] = timelockFinish;
    emit Timelocked(account, timelockFinish);
  }
  _mint(account, amount);
}
```

0xKiwi (NFTX) disputed

0xKiwi (NFTX) confirmed and commented:

> After taking another look, this is definitely accurate. Thank you!

0xKiwi (NFTX) resolved

[M-10] NFTXSimpleFeeDistributor._sendForReceiver doesn't return success if receiver is not a contract

Submitted by hyh

Double spending of fees being distributed will happen in favor of the first fee receivers in the `feeReceivers` list at the expense of the last ones. As `_sendForReceiver` doesn't return success for completed transfer when receiver isn't a contract, the corresponding fee amount is sent out twice, to the current and to the next fee receiver in the list. This will lead to double payments for those receivers who happen to be next in the line right after EOAs, and missed payments for the receivers positioned closer to the end of the list as the funds available are going to be already depleted when their turn comes.

Proof of Concept

`distribute` use `_sendForReceiver` to transfer current vault balance across `feeReceivers` : https://github.com/code-423n4/2021-12-nftx/blob/main/nftx-protocol-v2/contracts/solidity/NFTXSimpleFeeDistributor.sol#L67

`_sendForReceiver` returns a boolean that is used to move current distribution amount to the next receiver when last transfer failed. When `_receiver.isContract` is `false` nothing is returned, while `safeTransfer` is done: https://github.com/code-423n4/2021-12-nftx/blob/main/nftx-protocol-v2/contracts/solidity/NFTXSimpleFeeDistributor.sol#L168

This way `_sendForReceiver` will indicate that transfer is failed and leftover amount to be added to the next transfer, i.e. the `amountToSend` will be spent twice: https://github.com/code-423n4/2021-12-nftx/blob/main/nftx-protocol-v2/contracts/solidity/NFTXSimpleFeeDistributor.sol#L64

Recommended Mitigation Steps

Now:

```
    function _sendForReceiver(FeeReceiver memory _receiver, uint256
      if (_receiver.isContract) {
      ...
      } else {
        IERC20Upgradeable(_vault).safeTransfer(_receiver.receiver, a
      }
    }
```

To be:

```
function _sendForReceiver(FeeReceiver memory _receiver, uint256
  if (_receiver.isContract) {
  ...
  } else {
    IERC20Upgradeable(_vault).safeTransfer(_receiver.receiver, a
    return true;
  }
}
```

0xKiwi (NFTX) confirmed and resolved

[M-11] NFTXVaultFactoryUpgradeable implementation can be replaced in production breaking the system

Submitted by hyh

`NFTXVaultFactory` contract holds information regarding vaults, assets and permissions (vaults, _vaultsForAsset and excludedFromFees mappings). As there is no mechanics present that transfers this information to another implementation, the switch of nftxVaultFactory to another address performed while in production will break the system.

Proof of Concept

`setNFTXVaultFactory` function allows an owner to reset `nftxVaultFactory` without restrictions in the following contracts:

NFTXLPStaking https://github.com/code-423n4/2021-12-nftx/blob/main/nftx-protocol-v2/contracts/solidity/NFTXLPStaking.sol#L59

NFTXInventoryStaking https://github.com/code-423n4/2021-12-nftx/blob/main/nftx-protocol-v2/contracts/solidity/NFTXInventoryStaking.sol#L51

NFTXSimpleFeeDistributor https://github.com/code-423n4/2021-12-nftx/blob/main/nftx-protocol-v2/contracts/solidity/NFTXSimpleFeeDistributor.sol#L135

Recommended Mitigation Steps

Either restrict the ability to change the factory implementation to pre-production stages or make `nftxVaultFactory` immutable by allowing changing it only once:

Now:

```
function setNFTXVaultFactory(address newFactory) external virtua
  require(newFactory != address(0));
  nftxVaultFactory = INFTXVaultFactory(newFactory);
}
```

To be:

```
function setNFTXVaultFactory(address newFactory) external virtua
  require(nftxVaultFactory == address(0), "nftxVaultFactory is i
  nftxVaultFactory = INFTXVaultFactory(newFactory);
}
```

If the implementation upgrades in production is desired, the factory data migration logic should be implemented and then used atomically together with the implementation switch in all affected contracts.

0xKiwi (NFTX) confirmed and commented:

> This is not a contract that is designed to be replaced, but upgraded. But it is a valid concern that these assistant contracts can have their factory be changed and rendered broken. (even if it were permissioned)
> Confirming.

0xKiwi (NFTX) resolved

[M-12] `buyAndSwap1155WETH` Does Not Work As Intended

Submitted by leastwood

The `buyAndSwap1155WETH` function in `NFTXMarketplaceZap` aims to facilitate buying and swapping `ERC1155` tokens within a single transaction. The function expects to transfer `WETH` tokens from the `msg.sender` account and use these tokens in purchasing vault tokens. However, the `_buyVaultToken` call in `buyAndSwap1155WETH` actually uses `msg.value` and not `maxWethIn`. As a result, the function will not work unless the user supplies both `WETH` and native `ETH` amounts, equivalent to the `maxWethIn` amount.

Proof of Concept

https://github.com/code-423n4/2021-12-nftx/blob/main/nftx-protocol-v2/contracts/solidity/NFTXMarketplaceZap.sol#L284-L314

```
    function buyAndSwap1155WETH(
      uint256 vaultId,
      uint256[] memory idsIn,
      uint256[] memory amounts,
      uint256[] memory specificIds,
      uint256 maxWethIn,
      address[] calldata path,
      address to
    ) public payable nonReentrant {
      require(to != address(0));
      require(idsIn.length != 0);
      IERC20Upgradeable(address(WETH)).transferFrom(msg.sender, addr
      uint256 count;
      for (uint256 i = 0; i < idsIn.length; i++) {
          uint256 amount = amounts[i];
          require(amount > 0, "Transferring < 1");
          count += amount;
      }
      INFTXVault vault = INFTXVault(nftxFactory.vault(vaultId));
      uint256 redeemFees = (vault.targetSwapFee() * specificIds.lenc
          vault.randomSwapFee() * (count - specificIds.length)
      );
      uint256[] memory swapAmounts = _buyVaultToken(address(vault),
      _swap1155(vaultId, idsIn, amounts, specificIds, to);

      emit Swap(count, swapAmounts[0], to);

      // Return extras.
      uint256 remaining = WETH.balanceOf(address(this));
```

```
        WETH.transfer(to, remaining);
    }
```

Tools Used

Manual code review. Discussions with Kiwi.

Recommended Mitigation Steps

Consider updating the `buyAndSwap1155WETH` function such that the following line of code is used instead of **this**.

```
    uint256[] memory swapAmounts = _buyVaultToken(address(vault), re
```

0xKiwi (NFTX) confirmed and resolved

[M-13] Dishonest Stakers Can Siphon Rewards From `xToken` Holders Through The `deposit` Function In `NFTXInventoryStaking`

Submitted by leastwood

`xTokens` is intended to be a representation of staked vault tokens. As the protocol's vaults accrue fees from users, these fees are intended to be distributed to users in an inconsistent fashion. `NFTXInventoryStaking` is one of the ways users can stake vault tokens. Deposits are timelocked for `2` seconds by default, essentially rendering flash loan attacks redundant. However, it is more than likely that the same user could withdraw their `xToken` deposit in the next block (assuming an average block time of just over 13 seconds).

Hence, if a well-funded attacker sees a transaction to distribute rewards to `xToken` holders, they could deposit a large sum of vault tokens and receive a majority share of the rewards before withdrawing their tokens in the following block. Additionally, the attacker can force distribute rewards in `NFTXSimpleFeeDistributor` as there is no access control on the `distribute` function.

This issue allows users to siphon user's rewards from the protocol, intended to be distributed to honest vault token stakers.

Proof of Concept

Consider the following exploit scenario:

- Currently there are 1000 `shares` and 1000 `base tokens` in the `XTokenUpgradeable` contract.
- Honest actor, Alice, calls `distribute` in `NFTXSimpleFeeDistributor` which attempts to send 200 `base tokens` as rewards for `xToken` holders accrued via protocol usage.
- Bob sees a transaction to reward `xToken` holders and frontruns this transaction by staking vault tokens, minting 1000 `shares` and 1000 `base tokens`.
- Rewards are distributed such that `XTokenUpgradeable` has 2000 `shares` and 2200 `base tokens`.
- Bob unstakes his tokens and exits the pool, redeeming his 1000 `shares` for 1100 `base tokens`.
- As a result, Bob was able to siphon off 100 `base tokens` without having to stake their tokens for the same time period that Alice had staked her tokens for.
- This unfair distribution can be abused again and again to essentially reward dishonest actors over honest staking participants such as Alice.

Tools Used

Manual code review. Discussions with Kiwi.

Recommended Mitigation Steps

Consider adding a delay to users token deposits into the `XTokenUpgradeable` such that miners cannot feasibly censor a transaction for the specified time interval and users cannot frontrun a transaction to distribute rewards. The interval should be chosen such that enough time is provided for the transaction to be included in a block, given poor network conditions.

I.e. If the chosen interval is 20 blocks. Miners must be able to censor the rewards distribution for 20 blocks. This is unlikely as there would need to be sufficient miner

collusion for value to be extracted from the protocol. Additionally, an interval of 20 blocks means that stakers who attempt to enter the pool upon seeing the transaction in the mempool won't be rewarded for such behaviour.

It is also essential that the `distribute` function in `NFTXSimpleFeeDistributor` is restricted to a given role, ensuring malicious users cannot control at what point rewards are distributed.

Alternatively, PoolTogether has a Time-Weighted-Average-Balance (TWAB) implementation which can be used as **reference**. This would ensure the fairest distribution of rewards to stakers, however, there are additional gas costs associated with this implementation. Hence, unless the protocol intends to be used primarily on L2 protocols, this solution should be avoided.

0xKiwi (NFTX) acknowledged, but disagreed with high severity and commented:

> While this attack is possible, without available flash liquidity, this attack vector requires a lot of (possibly difficult to acquire) capital to execute. Disagreeing with severity and acknowledging.

LSDan (judge) decreased severity to medium and commented:

> I agree with the sponsor that the risk of this happening is almost zero. Yes it's technically possible but the funds lost are going to be minimal and the attacker will almost definitely pay more in slippage and gas fees than they make. That said, this is a direct attack which results in a loss of user funds so making it less than medium risk seems disingenuous.

[M-14] Return variable can remain unassigned in _sendForReceiver

Submitted by sirhashalot, also found by pauliax

The `_sendForReceiver()` function only sets a return function in the "if" code block, not the "else" case. If the "else" case is true, no value is returned. The result of this oversight is that the `_sendForReceiver()` function called from the `distribute()` function could sucessfully enter its `else` block if a receiver has

`isContract` set to False and successfully transfer the `amountToSend` value. The `ditribute()` function will then have `leftover > 0` and send `currentTokenBalance` to the treasury. This issue is partially due to Solidity using implicit returns, so if no bool value is explicitly returned, the default bool value of False will be returned.

This problem currently occurs for any receiver with `isContract` set to False. The `_addReceiver` function allows for `isContract` to be set to False, so such a condition should not result in tokens being sent to the treasury as though it was an emergency scenario.

Proof of Concept

The `else` block is missing a return value https://github.com/code-423n4/2021-12-nftx/blob/194073f750b7e2c9a886ece34b6382b4f1355f36/nftx-protocol-v2/contracts/solidity/NFTXSimpleFeeDistributor.sol#L167-L169

Tools Used

VS Code “Solidity Visual Developer” extension

Recommended Mitigation Steps

Verify that functions with a return value do actually return a value in all cases. Adding the line `return true;` can be added to the end of the `else` block as one way to resolve this.

Alternatively, if `isContract` should never be set to False, the code should be designed to prevent a receiver from being added with this value.

0xKiwi (NFTX) confirmed and resolved

[M-15] No access control on assignFees() function in NFTXVaultFactoryUpgradeable contract

Submitted by ych18

If the Vault owner decides to set factoryMintFee and factoryRandomRedeemFee to zero, any user could call the function NFTXVaultFactoryUpgradeable.assignFees() and hence all the fees are updated.

0xKiwi (NFTX) confirmed and commented:

> This function is left over from some upgrades. It will be removed. Thank you.

0xKiwi (NFTX) resolved

[M-16] Malicious receiver can make distribute function denial of service

Submitted by cccz

In the NFTXSimpleFeeDistributor.sol contract, the distribute function calls the _sendForReceiver function to distribute the fee

```
function distribute(uint256 vaultId) external override virtual r
  require(nftxVaultFactory != address(0));
  address _vault = INFTXVaultFactory(nftxVaultFactory).vault(vau

  uint256 tokenBalance = IERC20Upgradeable(_vault).balanceOf(adc

  if (distributionPaused || allocTotal == 0) {
    IERC20Upgradeable(_vault).safeTransfer(treasury, tokenBalanc
    return;
  }

  uint256 length = feeReceivers.length;
  uint256 leftover;
  for (uint256 i = 0; i <length; i++) {
    FeeReceiver memory _feeReceiver = feeReceivers[i];
    uint256 amountToSend = leftover + ((tokenBalance * _feeRecei
    uint256 currentTokenBalance = IERC20Upgradeable(_vault).bala
    amountToSend = amountToSend> currentTokenBalance? currentTok
    bool complete = _sendForReceiver(_feeReceiver, vaultId, _vau
    if (!complete) {
      leftover = amountToSend;
    } else {
      leftover = 0;
```

```
    }
  }
```

In the _sendForReceiver function, when the _receiver is a contract, the receiver's receiveRewards function will be called. If the receiver is malicious, it can execute revert() in the receiveRewards function, resulting in DOS.

```
function _sendForReceiver(FeeReceiver memory _receiver, uint256
  if (_receiver.isContract) {
    IERC20Upgradeable(_vault).approve(_receiver.receiver, amount
    // If the receive is not properly processed, send it to the

    bytes memory payload = abi.encodeWithSelector(INFTXLPStaking
    (bool success,) = address(_receiver.receiver).call(payload);

    // If the allowance has not been spent, it means we can pass
    return success && IERC20Upgradeable(_vault).allowance(addres
  } else {
    IERC20Upgradeable(_vault).safeTransfer(_receiver.receiver, a
  }
}
```

Proof of Concept

https://github.com/code-423n4/2021-12-nftx/blob/main/nftx-protocol-v2/contracts/solidity/NFTXSimpleFeeDistributor.sol#L157-L166

Recommended Mitigation Steps

The contract can store the fee sent to the receiver in a state variable, and then the receiver can take it out by calling a function.

0xKiwi (NFTX) confirmed, but disagreed with high severity and commented:

> This is a permissioned entity, but this is a valid concern. User funds not at risk and the dao can remove the malciious receiver (if it ever gets there in the first place). Good thinking.

0xKiwi (NFTX) resolved

LSDan (judge) decreased severity to medium and commented:

> This is a medium risk, not high. The attack has external requirements and is relatively easy for the DAO to mitigate.

[M-17] transfer return value is ignored

Submitted by robee, also found by 0x1f8b, cmichel, csanuragjain, defsec, hyh, leastwood, sirhashalot, and WatchPug

Need to use safeTransfer instead of transfer. As there are popular tokens, such as USDT that transfer/transferFrom method doesn't return anything. The transfer return value has to be checked (as there are some other tokens that returns false instead revert), that means you must

1. Check the transfer return value

Another popular possibility is to add a whiteList. Those are the appearances (solidity file, line number, actual line):

```
NFTXStakingZap.sol, 401, IERC20Upgradeable(vault).transfer(to, n
NFTXStakingZap.sol, 474, IERC20Upgradeable(token).transfer(msg.s
PalmNFTXStakingZap.sol, 190, pairedToken.transferFrom(msg.sender
PalmNFTXStakingZap.sol, 195, pairedToken.transfer(to, wethIn-amo
PalmNFTXStakingZap.sol, 219, pairedToken.transferFrom(msg.sender
PalmNFTXStakingZap.sol, 224, pairedToken.transfer(to, wethIn-amo
PalmNFTXStakingZap.sol, 316, IERC20Upgradeable(vault).transfer(t
XTokenUpgradeable.sol, 54, baseToken.transfer(who, what);
NFTXFlashSwipe.sol, 51, IERC20Upgradeable(vault).transferFrom(ms
```

0xKiwi (NFTX) confirmed, but disagreed with high severity and commented:

> Disagreeing with the severity, but will make sure I stick to safeTransferFrom, thank you.

0xKiwi (NFTX) resolved

LSDan (judge) decreased severity to medium and commented:

> This is medium risk, not high. Loss of funds requires external factors.

Low Risk Findings (38)

- [L-01] safeApprove of openZeppelin is deprecated *Submitted by robee*
- [L-02] Validations *Submitted by pauliax*
- [L-03] NFTXVaultFactoryUpgradeable.sol function assignFees() does not have onlyOwner modifier *Submitted by PPrieditis*
- [L-04] Users can create vaults with a malicious _assetAddress *Submitted by jayjonah8*
- [L-05] Timelock functionality for `xToken` is applied on all existing balance *Submitted by 0x0x0x*
- [L-06] Unsafe approve in NFTXSimpleFeeDistributor *Submitted by 0x1f8b*
- [L-07] Outdated comment in `TimelockRewardDistributionTokenImpl.burnFrom` *Submitted by cmichel*
- [L-08] PausableUpgradeable: Document lockId code 10 = deposit *Submitted by GreyArt*
- [L-09] Cached lpStaking and inventoryStaking in Zap contracts *Submitted by pauliax*
- [L-10] NFTXMarketplaceZap: Restrict native ETH transfers to WETH contract *Submitted by GreyArt, also found by defsec and pauliax*
- [L-11] NFTXSimpleFeeDistributor: Inconsistency between implementation and comment *Submitted by GreyArt, also found by hyh*
- [L-12] NFTXMarketplaceZap: Add rescue() function *Submitted by GreyArt*
- [L-13] NFTXStakingZap: Sanity checks on “to” (dest) address *Submitted by GreyArt*
- [L-14] Marketplace allows functions made for ERC721 vaults to interact with ERC1155 vaults *Submitted by Ruhum*
- [L-15] InventoryStaking `deposit()` and `withdraw()` don’t validate passed vaultId *Submitted by Ruhum*
- [L-16] `NFTXSimpleFeeDistributor#__SimpleFeeDistributor__init__()` Missing `__ReentrancyGuard_init()` *Submitted by WatchPug*

- [L-17] Race condition in approve() 收件箱 *Submitted by cccz, also found by WatchPug*
- [L-18] Same module can be added several times *Submitted by cmichel, also found by csanuragjain*
- [L-19] Zaps should verify paths *Submitted by cmichel, also found by leastwood*
- [L-20] Init frontrun *Submitted by robee*
- [L-21] Unbounded iteration in `NFTXVaultUpgradeable.allHoldings` over all holdings *Submitted by cmichel*
- [L-22] Inconsistency in fee distribution *Submitted by csanuragjain*
- [L-23] Unfair fee distribution *Submitted by p4st13r4, also found by cccz and cmichel*
- [L-24] Missing OOB check in `changeReceiverAlloc` *Submitted by gzeon*
- [L-25] `NFTXLPStaking.rewardDistTokenImpl` is never initialized *Submitted by Ruhum*
- [L-26] NFTXVaultUpgradeable.mintTo and swapTo do not check for user supplied arrays length *Submitted by hyh*
- [L-27] DOS on withdrawal *Submitted by csanuragjain*
- [L-28] NFTXInventoryStaking._deployXToken create2 deploy result isn't zero checked *Submitted by hyh*
- [L-29] NFTXStakingZap, NFTXMarketplaceZap and NFTXVaultUpgradeable use hard coded Cryptokitties and CryptoPunks addresses *Submitted by hyh*
- [L-30] onlyOwnerIfPaused(0) argument should not be hard coded *Submitted by jayjonah8*
- [L-31] Missing address(0) checks *Submitted by sirhashalot, also found by BouSalman and robee*
- [L-32] `timelockMint` In `TimelockRewardDistributionTokenImpl` Does Not Ensure Mint Is Greater Than Zero *Submitted by leastwood*
- [L-33] `assignDefaultFeatures` Does Nothing *Submitted by leastwood*
- [L-34] Rewards Cannot Be Claimed If LP Tokens Are Unstaked *Submitted by leastwood*
- [L-35] max timelockLength *Submitted by pauliax*

- [L-36] Require with empty message *Submitted by robee, also found by hyh and WatchPug*
- [L-37] Require with not comprehensive message *Submitted by robee*
- [L-38] Two Steps Verification before Transferring Ownership *Submitted by robee, also found by Dravee and leastwood*

Non-Critical Findings (19)

- [N-01] Wrong code style *Submitted by 0x1f8b*
- [N-02] Local variables shadowing *Submitted by sirhashalot*
- [N-03] Weak nonce usage *Submitted by sirhashalot*
- [N-04] NFTXMarketplaceZap: incorrect parameter name *Submitted by GreyArt*
- [N-05] NFTXInventoryStaking: Index vaultId in events *Submitted by GreyArt*
- [N-06] NFTXLPStaking: Implementation Upgrade Storage Layout Caution *Submitted by GreyArt*
- [N-07] Staking Zap approves wrong LP token *Submitted by cmichel*
- [N-08] isContract() duplication and Address.sol library usage *Submitted by PPrieditis*
- [N-09] TimelockRewardDistributionTokenImpl.sol function withdrawableRewardOf() visibility can be changed from internal to public *Submitted by PPrieditis*
- [N-10] Unused function input argument “vault” *Submitted by PPrieditis*
- [N-11] Internal functions names should start with underscore *Submitted by sirhashalot*
- [N-12] Constants are not explicitly declared *Submitted by WatchPug*
- [N-13] Incorrect contract referenced in test *Submitted by sirhashalot*
- [N-14] Unchecked return value for `ERC20.approve` call *Submitted by WatchPug*
- [N-15] Outdated compiler version *Submitted by WatchPug*
- [N-16] `transfer()` is not recommended for sending ETH *Submitted by WatchPug*

- [N-17] Use of floating pragma *Submitted by saian*
- [N-18] Sell event amounts[1] *Submitted by pauliax*
- [N-19] Misleading comments *Submitted by p4st13r4*

Gas Optimizations (47)

- [G-01] Upgrade pragma to at least 0.8.4 *Submitted by Dravee, also found by defsec*
- [G-02] Using 10**X for constants isn't gas efficient *Submitted by Dravee, also found by WatchPug*
- [G-03] Unnecessary checked arithmetic in for-loops *Submitted by Dravee, also found by 0x0x0x, defsec, PPrieditis, robee, and WatchPug*
- [G-04] `++i` costs less gass compared to `i++` *Submitted by Dravee, also found by 0x0x0x, robee, and WatchPug*
- [G-05] Unused local variables *Submitted by WatchPug, also found by gzeon and sirhashalot*
- [G-06] Explicit initialization with zero not required *Submitted by Dravee, also found by robee*
- [G-07] Constants can be made internal / private *Submitted by Dravee*
- [G-08] Avoid unnecessary external call can save gas *Submitted by WatchPug*
- [G-09] NFTXStakingZap: Unused xTokenMinted variable *Submitted by GreyArt*
- [G-10] Use immutable variables can save gas *Submitted by WatchPug*
- [G-11] Remove unnecessary variables can make the code simpler and save some gas *Submitted by WatchPug*
- [G-12] Public functions to external *Submitted by robee, also found by cccz, csanuragjain, and WatchPug*
- [G-13] `NFTXMarketplaceZap.sol#buyAndSwap1155WETH()` Implementation can be simpler and save some gas *Submitted by WatchPug*
- [G-14] Move kitties/punk addresses to constants *Submitted by p4st13r4*
- [G-15] Inline unnecessary function can make the code simpler and save some gas *Submitted by WatchPug*
- [G-16] Unused events *Submitted by WatchPug*

- [G-17] Unused function parameters *Submitted by WatchPug*
- [G-18] Check if amount > 0 before token transfer can save gas *Submitted by WatchPug*
- [G-19] Cache storage variables in the stack can save gas *Submitted by WatchPug*
- [G-20] Unused imports *Submitted by robee, also found by PPrieditis*
- [G-21] Using constants instead of local variables can save some gas *Submitted by WatchPug*
- [G-22] Named return issue *Submitted by robee*
- [G-23] Gas savings *Submitted by csanuragjain*
- [G-24] Unnecessary assignment of 0 to an uninitialized variable *Submitted by Ruhum*
- [G-25] Gas saving by storing modulesCopy.length in local variable *Submitted by csanuragjain*
- [G-26] Store totalSupply() in variable *Submitted by sirhashalot*
- [G-27] Gas saving by using mapping instead of computeAddress *Submitted by csanuragjain*
- [G-28] Unnecessary check for a condition *Submitted by ych18*
- [G-29] After Solidity 0.8.1, The Inline Assembly Contract Check Can Be replaced with the new function *Submitted by defsec*
- [G-30] Use a constant instead of block.timestamp for the deadline *Submitted by defsec*
- [G-31] Use `calldata` instead of `memory` for function parameters *Submitted by defsec, also found by Dravee*
- [G-32] `> 0` can be replaced with `!= 0` for gas optimization *Submitted by defsec, also found by 0x0x0x, Dravee, and pedroais*
- [G-33] Gas Optimization: Use uint232 for `allocPoint` *Submitted by gzeon*
- [G-34] Gas Optimization: Use immutable to cache beaconhash *Submitted by gzeon*
- [G-35] isContract() code is duplicated in multiple files *Submitted by jayjonah8*
- [G-36] Cache duplicate external calls *Submitted by pauliax*
- [G-37] Unused state variables *Submitted by pauliax, also found by GreyArt*

- [G-38] uint64 state variable is less efficient than uint256 *Submitted by pauliax*
- [G-39] Use cached version of sushiRouter.WETH() *Submitted by pauliax, also found by cmichel*
- [G-40] Use unchecked math and cache values *Submitted by pauliax, also found by WatchPug*
- [G-41] Refactor `distribute()` logic *Submitted by sirhashalot*
- [G-42] Use bytes32 instead of string to save gas whenever possible *Submitted by robee*
- [G-43] Short the following require messages *Submitted by robee, also found by pauliax, sirhashalot, and WatchPug*
- [G-44] Storage double reading. Could save SLOAD *Submitted by robee, also found by WatchPug*
- [G-45] Unnecessary array boundaries check when loading an array element twice *Submitted by robee*
- [G-46] Importing unused contract *Submitted by saian*
- [G-47] provideInventory1155 assumes tokenIds.length == amounts.length *Submitted by sirhashalot*

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