2021-05-nftx (NFTX)

Benchmark Security Review Performed by Bug Hunter

26.06.2025

Table of Contents

About Bug Hunter

Bug Hunter is an automated code-review tool for Solidity smart contracts, developed by Truscova GmbH. It uses proprietary algorithms and advanced code-analysis techniques to identify potential vulnerabilities efficiently.

Bug Hunter is designed to support developers in the Ethereum ecosystem by delivering consistent, scalable assessments that align with best practices in Web 3.0 security. Reports are generated via the Bug Hunter tool and are accessible to clients through a secure dashboard web interface or relevant Git repositories.

Bug Hunter is a product of Truscova, a security R&D company headquartered in Bremen, Germany. Founded in late 2022, Truscova specializes in formal verification, fuzzing, dynamic and static analysis, and security tooling. Its founding team includes researchers with 35+ books, several patents, and over 20 000 citations in software testing, verification, security, and machine learning.

To explore more about Bug Hunter, visit:

- Bug Hunter Website
- Bug Hunter Documentation
- Example Reports

To explore more about Truscova, visit:

- Truscova Website
- X
- LinkedIn

Notices and Remarks

Copyright and Distribution

This security-review report was generated with Truscova's Bug Hunter tool to benchmark its performance on selected NFT projects. It is made available through the Bug Hunter dashboard and relevant Git repositories. Users may choose to share or publish this report at their own discretion.

Test Coverage Disclaimer

This security review was conducted exclusively by Bug Hunter, an automated code-review tool for Solidity codebases. Bug Hunter uses a combination of proprietary algorithms and programmatic techniques to identify potential vulnerabilities in Solidity smart contracts.

The findings in this report should not be considered a complete or exhaustive list of all possible security issues in the reviewed codebase and are dependent on Bug Hunter's coverage of detector rules. The full list of rules covered by Bug Hunter automated code review is available at https://docs.bughunter.live/.

Executive Summary

Overview and Scope

A codebase from project 2021-05-nftx was reviewed for security vulnerabilities by Bug Hunter with the following details:

- GitHub Repository: https://github.com/code-423n4/2021-05-nftx
  Commit hash: `f6d793c136d110774de259d9f3b25d003c4f8098`

Bug Hunter reviewed the following files:

Complete Project

Summary of Findings

The uncovered vulnerabilities identified during the security review are summarised in the table below:

Severity	Count
High	2
Medium	5
Low	4

Project Summary

Contact Information

For support, contact us at support@bughunter.live

Project Timeline

Date	Event
26.06.2025	Project submitted
26.06.2025	Security review concluded

Detailed Findings

1. Unsafe Arithmetic Operations Due to Lack of Overflow/Underflow Checks

Severity	High
Finding ID	BH-2021-05-nftx-01
Target	contracts/solidity/token/ERC20FlashMintUpgradeable.sol
Function name	flashLoan

Description

The function performs arithmetic operations (e.g., `amount + fee`, `currentAllowance - amount - fee`) without using SafeMath or similar protections. Since the Solidity version is 0.6.8, these operations can overflow or underflow without reverting, potentially leading to incorrect logic or exploitable conditions.

```
function flashLoan(
        IERC3156FlashBorrowerUpgradeable receiver,
        address token,
        uint256 amount,
        bytes memory data
    )
        public virtual override returns (bool)
    {
        uint256 fee = flashFee(token, amount);
        _mint(address(receiver), amount);
        require(receiver.onFlashLoan(msg.sender, token, amount, fee, data) ==
RETURN_VALUE, "ERC20FlashMint: invalid return value");
        uint256 currentAllowance = allowance(address(receiver), address(this));
        require(currentAllowance >= amount + fee, "ERC20FlashMint: allowance does
not allow refund");
        _approve(address(receiver), address(this), currentAllowance - amount - fee);
        _burn(address(receiver), amount + fee);
        return true;
    }
```

Recommendation

Use the SafeMath library for all arithmetic operations, or upgrade to Solidity 0.8.x or later, which has built-in overflow/underflow protection by default.

2. Denial of Service via Malicious or Faulty FeeReceiver Contract

Severity	High
Finding ID	BH-2021-05-nftx-02
Target	contracts/solidity/NFTXFeeDistributor.sol
Function name	_sendForReceiver

Description

The `_sendForReceiver` function performs a low-level `call` to the `receiver` contract's `receiveRewards` function and immediately decodes the return value using `abi.decode`. If the external contract reverts, returns malformed data, or is maliciously constructed, this will cause the entire `distribute` call to revert. Since `distribute` processes all fee receivers in a loop, a single misbehaving receiver can prevent the entire distribution process, effectively halting protocol revenue sharing. This is a classic denial-of-service vector.

```
function _sendForReceiver(FeeReceiver memory _receiver, uint256 _vaultId, address
_vault, uint256 _tokenBalance, uint256 _allocTotal) internal {
    uint256 amountToSend = _tokenBalance * _receiver.allocPoint / _allocTotal;
    uint256 balance = IERC20Upgradeable(_vault).balanceOf(address(this)) - 1000;
    amountToSend = amountToSend > balance ? balance : amountToSend;
    if (_receiver.isContract) {
      IERC20Upgradeable(_vault).approve(_receiver.receiver, amountToSend);
      bytes memory payload =
abi.encodeWithSelector(INFTXLPStaking.receiveRewards.selector, _vaultId,
amountToSend);
      (bool success, bytes memory returnData) =
address(_receiver.receiver).call(payload);
      bool tokensReceived = abi.decode(returnData, (bool));
      if (!success || !tokensReceived) {
        console.log("treasury fallback");
        IERC20Upgradeable(_vault).safeTransfer(treasury, amountToSend);
      }
    } else {
      IERC20Upgradeable(_vault).safeTransfer(_receiver.receiver, amountToSend);
    }
  }
```

Recommendation

Use `returnData.length > 0` to check for valid return data before decoding. Wrap external calls in a `try/catch` block or verify the target address is a contract. Alternatively, isolate each receiver payout in a separate transaction or batch to avoid full revert.

3. Predictable Randomization Enables Brute-Forcing of NFT Redemptions

Severity	Medium
Finding ID	BH-2021-05-nftx-03
Target	contracts/solidity/NFTXVaultUpgradeable.sol
Function name	withdrawNFTsTo

Description

The `withdrawNFTsTo` function uses `getRandomTokenIdFromFund`, which relies on insecure pseudo-randomness (`keccak256(blockhash(block.number - 1), randNonce)`). Since both `blockhash` and `randNonce` are publicly known or inferable, attackers can simulate the outcome of the random token ID selection off-chain. This allows them to repeatedly brute-force the `redeem` or `swap` logic until they identify a high-value NFT that will be returned, then submit a transaction to exploit this knowledge. This undermines the fairness of the vault and enables selective redemption of rare NFTs.

```
function withdrawNFTsTo(
        uint256 amount,
        uint256[] memory specificIds,
        address to
    ) internal virtual returns (uint256[] memory) {
        bool _is1155 = is1155;
        address _assetAddress = assetAddress;
        uint256[] memory redeemedIds = new uint256[](amount);

        for (uint256 i = 0; i < amount; i++) {
            uint256 tokenId = i < specificIds.length
                ? specificIds[i]
                : getRandomTokenIdFromFund();
            redeemedIds[i] = tokenId;

            if (_is1155) {
                IERC1155Upgradeable(_assetAddress).safeTransferFrom(
                    address(this),
                    to,
                    tokenId,
                    1,
```

```
                    ""
                );

                quantity1155[tokenId] = quantity1155[tokenId].sub(1);
                if (quantity1155[tokenId] == 0) {
                    holdings.remove(tokenId);
                }
            } else {
                IERC721Upgradeable(_assetAddress).safeTransferFrom(
                    address(this),
                    to,
                    tokenId
                );
                holdings.remove(tokenId);
            }
        }
        return redeemedIds;
    }
```

Recommendation

Replace the pseudo-random generator with a secure and verifiable randomness source such as Chainlink VRF. If verifiable randomness is not feasible, use additional sources of entropy (e.g., user address, timestamp, or block.difficulty) and obscure randNonce state. Consider delaying execution or enforcing time windows to limit brute-force opportunities.

4. Use of transfer Instead of safeTransfer for ERC20 Tokens

Severity	Medium
Finding ID	BH-2021-05-nftx-04
Target	contracts/solidity/NFTXLPStaking.sol
Function name	_withdraw

Description

The `_withdraw` function uses `IERC20Upgradeable(pool.stakingToken).transfer(account, amount)` to transfer tokens. However, the `transfer` function is not universally safe across all ERC20 implementations. Some tokens (e.g., USDT) do not return a boolean value or may behave non-standardly, which can lead to unexpected behavior or silent failures. The use of `safeTransfer` from OpenZeppelin's `SafeERC20` library ensures consistent handling of success/failure conditions and protects against these edge cases.

```
function _withdraw(StakingPool memory pool, uint256 amount, address account)
internal {
        _rewardDistributionTokenAddr(pool).burnFrom(account, amount);
        IERC20Upgradeable(pool.stakingToken).transfer(account, amount);
    }
```

Recommendation

Replace `transfer` with `safeTransfer` from OpenZeppelin's `SafeERC20Upgradeable` library to ensure compatibility with non-standard ERC20 tokens. Also ensure the `SafeERC20Upgradeable` library is properly imported and used throughout the contract for all token transfers.

5. Unbounded Loop Over Fee Receivers Can Cause Gas-Based Denial of Service

Severity	Medium
Finding ID	BH-2021-05-nftx-05
Target	contracts/solidity/NFTXFeeDistributor.sol
Function name	distribute

Description

The `distribute` function iterates over all entries in the `feeReceivers[vaultId]` array using a for-loop. If the number of fee receivers becomes large, the gas required to complete the loop may exceed the block gas limit. In such a case, it becomes impossible to call `distribute`, effectively halting fee distribution for the vault. This creates a denial-of-service risk, especially if the array is user-controlled or not strictly bounded.

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

    uint256 tokenBalance = IERC20Upgradeable(_vault).balanceOf(address(this));
    if (tokenBalance <= 10**9) {
      return;
    }
    tokenBalance -= 1000;

    uint256 _treasuryAlloc = specificTreasuryAlloc[vaultId];
    if (_treasuryAlloc == 0) {
      _treasuryAlloc = defaultTreasuryAlloc;
    }

    uint256 _allocTotal = allocTotal[vaultId] + _treasuryAlloc;
    uint256 amountToSend = tokenBalance * _treasuryAlloc / _allocTotal;
    amountToSend = amountToSend > tokenBalance ? tokenBalance : amountToSend;
    IERC20Upgradeable(_vault).safeTransfer(treasury, amountToSend);

    if (distributionPaused) {
      return;
    }
```

```
    FeeReceiver[] memory _feeReceivers = feeReceivers[vaultId];
    for (uint256 i = 0; i < _feeReceivers.length; i++) {
      _sendForReceiver(_feeReceivers[i], vaultId, _vault, tokenBalance,
_allocTotal);
    }
  }
```

Recommendation

Impose a maximum limit on the number of fee receivers per vault. Alternatively, break the distribution process into multiple smaller batched calls (e.g., using pagination or checkpoints). Ensure there are administrative tools to prune or update receiver lists safely. Consider emitting events for partial distributions to aid off-chain monitoring.

6. Potential Re-Entrancy via External Receiver Callback in distribute

Severity	Medium
Finding ID	BH-2021-05-nftx-06
Target	contracts/solidity/NFTXFeeDistributor.sol
Function name	distribute

Description

The `distribute` function performs low-level external calls to potentially untrusted contracts via `receiver.receiveRewards(...)` in the `_sendForReceiver` function. These calls occur before the loop completes and without using a reentrancy guard. This exposes the function to a re-entrancy attack where a malicious receiver contract could recursively call `distribute()` and trigger multiple reward distributions, double-claiming or corrupting allocation state. While the function deals with ERC20 tokens, certain token standards (e.g., ERC777) or poorly implemented custom tokens may allow callbacks, exacerbating this risk.

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

    uint256 tokenBalance = IERC20Upgradeable(_vault).balanceOf(address(this));
    if (tokenBalance <= 10**9) {
      return;
    }
    tokenBalance -= 1000;

    uint256 _treasuryAlloc = specificTreasuryAlloc[vaultId];
    if (_treasuryAlloc == 0) {
      _treasuryAlloc = defaultTreasuryAlloc;
    }

    uint256 _allocTotal = allocTotal[vaultId] + _treasuryAlloc;
    uint256 amountToSend = tokenBalance * _treasuryAlloc / _allocTotal;
    amountToSend = amountToSend > tokenBalance ? tokenBalance : amountToSend;
    IERC20Upgradeable(_vault).safeTransfer(treasury, amountToSend);

    if (distributionPaused) {
      return;
```

```
    }

    FeeReceiver[] memory _feeReceivers = feeReceivers[vaultId];
    for (uint256 i = 0; i < _feeReceivers.length; i++) {
      _sendForReceiver(_feeReceivers[i], vaultId, _vault, tokenBalance,
_allocTotal);
    }
  }
```

Recommendation

Apply OpenZeppelin's `ReentrancyGuard` to the contract and add the `nonReentrant` modifier to the `distribute` function. Additionally, follow the Checks-Effects-Interactions pattern by ensuring all state changes (e.g., allocations, balance deductions) occur before external calls. Consider validating receiver contracts or gating access to critical functions.

7. Tokens Can Get Stuck Due to Missing Mint Fulfillment or Cancellation Logic

Severity	Medium
Finding ID	BH-2021-05-nftx-07
Target	contracts/solidity/eligibility/NFTXMintRequestEligibility.sol
Function name	requestMint

Description

The `requestMint` function transfers NFTs from the user to the contract and records the request, but no corresponding logic for fulfilling, claiming, or cancelling the request is shown. As a result, user tokens can become permanently stuck in the contract if the request is never processed, is invalid, or if the vault is finalized. There are no timeout mechanisms or user-accessible withdrawal paths for reclaiming unfulfilled tokens, creating a lockup risk.

```
function requestMint(
        uint256[] calldata tokenIds,
        uint256[] calldata amounts
    ) external virtual {
        require(!finalized(), "Finalized");
        address _assetAddress = vault.assetAddress();
        for (uint256 i = 0; i < tokenIds.length; i++) {
            uint256 tokenId = tokenIds[i];
            uint256 amount = amounts[i];
            require(
                mintRequests[msg.sender][tokenId] == 0,
                "No existing request"
            );
            mintRequests[msg.sender][tokenId] = amount;
            if (is1155) {
                IERC1155Upgradeable(_assetAddress).safeTransferFrom(
                    msg.sender,
                    address(this),
                    tokenId,
                    amount,
                    ""
                );
            } else {
```

```
                IERC721Upgradeable(_assetAddress).safeTransferFrom(
                    msg.sender,
                    address(this),
                    tokenId
                );
            }
        }
        emit Request(msg.sender, tokenIds, amounts);
    }
```

Recommendation

Add explicit functions to fulfill or cancel mint requests, and allow users to reclaim their tokens if requests are not completed within a reasonable timeframe. Track request status (e.g., pending, completed, cancelled) and enforce expiration periods to prevent indefinite lockups.

8. Lack of Vault ID Validation in balanceOf Can Lead to Reverts or Incorrect Results

Severity	Low
Finding ID	BH-2021-05-nftx-08
Target	contracts/solidity/NFTXLPStaking.sol
Function name	balanceOf

Description

The `balanceOf` function copies the `StakingPool` struct from `vaultStakingInfo` into memory and proceeds to use it without validating the vaultId. If an invalid or uninitialized `vaultId` is passed, the derived reward distribution token may be zero or incorrect, leading to possible reverts or misleading return values from `dist.balanceOf(addr)`. Additionally, copying structs into memory when not needed can introduce unnecessary gas costs and semantic confusion in more complex functions.

```
function balanceOf(uint256 vaultId, address addr) public view returns (uint256) {
        StakingPool memory pool = vaultStakingInfo[vaultId];
        RewardDistributionTokenUpgradeable dist =
_rewardDistributionTokenAddr(pool);
        return dist.balanceOf(addr);
    }
```

Recommendation

Add a validation check to ensure the `vaultId` is valid (e.g., `require(pool.stakingToken != address(0), 'Invalid vault')`). Optionally, read the struct from storage directly to avoid confusion. Avoid memory copies unless mutability or struct decoupling is explicitly needed.

9. Lack of Input Sanitization in addModule Allows Invalid or Malicious Contracts

Severity	Low
Finding ID	BH-2021-05-nftx-09
Target	contracts/solidity/NFTXEligiblityManager.sol
Function name	addModule

Description

The `addModule` function does not validate the `implementation` address before registering it as an eligibility module. This allows the contract owner to add zero addresses, externally owned accounts (EOAs), or contracts that do not conform to the expected `EligibilityModule` interface. This may lead to runtime errors, unexpected behavior, or breakage in module-based eligibility checks. Additionally, there are no protections against duplicate module entries.

```
function addModule(address implementation) public onlyOwner {
    EligibilityModule memory module = EligibilityModule(implementation);
    modules.push(module);
  }
```

Recommendation

Add input validation to ensure the `implementation` is a deployed contract using `require(Address.isContract(implementation), 'Not a contract')`. Optionally, check that the contract supports the required interface using ERC165. Prevent duplicate entries by checking the module list or using a mapping.

10. Unsafe Arithmetic Operations Without SafeMath in distribute and _sendForReceiver

Severity	Low
Finding ID	BH-2021-05-nftx-10
Target	contracts/solidity/NFTXFeeDistributor.sol
Function name	distribute, _sendForReceiver

Description

The `distribute` and `_sendForReceiver` functions perform arithmetic operations (e.g., subtraction, multiplication, division) without using the SafeMath library in Solidity 0.6.x. This version of Solidity does not have built-in overflow or underflow protection, meaning that operations like `tokenBalance -= 1000` or `a * b / c` may silently wrap if inputs are not validated. This can lead to incorrect fee distribution, overflows, or underflows, especially if token balances or allocation values are manipulated or edge-case conditions are met.

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

    uint256 tokenBalance = IERC20Upgradeable(_vault).balanceOf(address(this));
    if (tokenBalance <= 10**9) {
      return;
    }
    tokenBalance -= 1000;

    uint256 _treasuryAlloc = specificTreasuryAlloc[vaultId];
    if (_treasuryAlloc == 0) {
      _treasuryAlloc = defaultTreasuryAlloc;
    }

    uint256 _allocTotal = allocTotal[vaultId] + _treasuryAlloc;
    uint256 amountToSend = tokenBalance * _treasuryAlloc / _allocTotal;
    amountToSend = amountToSend > tokenBalance ? tokenBalance : amountToSend;
    IERC20Upgradeable(_vault).safeTransfer(treasury, amountToSend);

    if (distributionPaused) {
      return;
```

```
    }

    FeeReceiver[] memory _feeReceivers = feeReceivers[vaultId];
    for (uint256 i = 0; i < _feeReceivers.length; i++) {
      _sendForReceiver(_feeReceivers[i], vaultId, _vault, tokenBalance,
_allocTotal);
    }
  }

function _sendForReceiver(FeeReceiver memory _receiver, uint256 _vaultId, address
_vault, uint256 _tokenBalance, uint256 _allocTotal) internal {
    uint256 amountToSend = _tokenBalance * _receiver.allocPoint / _allocTotal;
    uint256 balance = IERC20Upgradeable(_vault).balanceOf(address(this)) - 1000;
    amountToSend = amountToSend > balance ? balance : amountToSend;
    if (_receiver.isContract) {
      IERC20Upgradeable(_vault).approve(_receiver.receiver, amountToSend);
      bytes memory payload =
abi.encodeWithSelector(INFTXLPStaking.receiveRewards.selector, _vaultId,
amountToSend);
      (bool success, bytes memory returnData) =
address(_receiver.receiver).call(payload);
      bool tokensReceived = abi.decode(returnData, (bool));
      if (!success || !tokensReceived) {
        console.log("treasury fallback");
        IERC20Upgradeable(_vault).safeTransfer(treasury, amountToSend);
      }
    } else {
      IERC20Upgradeable(_vault).safeTransfer(_receiver.receiver, amountToSend);
    }
  }
```

Recommendation

Use OpenZeppelin's SafeMath library for all arithmetic operations in Solidity 0.6.x. Specifically, replace all `-`, `*`, `/` operations with their SafeMath equivalents (e.g., `tokenBalance = tokenBalance.sub(1000)`). Alternatively, upgrade to Solidity 0.8.x, which has built-in overflow/underflow protection by default.

11. Missing Input Validation in FeeDistributor Initializer

Severity	Low
Finding ID	BH-2021-05-nftx-11
Target	contracts/solidity/NFTXFeeDistributor.sol
Function name	__FeeDistributor__init__

Description

The `__FeeDistributor__init__` initializer function does not validate its input addresses `_lpStaking` and `_treasury`. If either is incorrectly set to the zero address or a non-contract address, the fee distribution logic may break or result in loss of funds. This risk is especially relevant in upgradeable contracts where improper initialization is a common vector for logic bugs or denial of service.

```
function __FeeDistributor__init__(address _lpStaking, address _treasury) public
override initializer {
    __Pausable_init();
    lpStaking = _lpStaking;
    treasury = _treasury;
    defaultTreasuryAlloc = 0.2 ether;
    defaultLPAlloc = 0.5 ether;
  }
```

Recommendation

Add explicit validation checks to ensure `_lpStaking` and `_treasury` are not zero addresses. Consider checking that `_lpStaking` is a contract using `AddressUpgradeable.isContract`. This ensures proper initialization and prevents accidental misconfiguration.