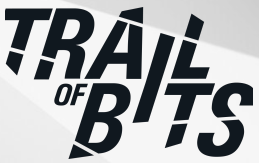

NFTX Protocol v2

Security Assessment

April 15, 2022

Prepared for:
Alex Gausman
NFTX

Prepared by: **Evan Sultanik, Jaime Iglesias**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and mutually agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

About Trail of Bits **1**

Notices and Remarks **2**

Table of Contents **3**

Executive Summary **5**

Project Summary **7**

Project Goals **8**

Project Targets **9**

Project Coverage **10**

Codebase Maturity Evaluation **12**

Summary of Findings **15**

Detailed Findings **16**

1. Reliance on third-party library for deployment 16

2. Missing validation of proxy admin indices 18

3. Random token withdrawals can be gamed 20

4. Duplicate receivers allowed by addReceiver() 22

5. OpenZeppelin vulnerability can break initialization 24

6. Potentially excessive gas fees imposed on users for protocol fee distribution 25

7. Risk of denial of service due to unbounded loop 28

8. A malicious fee receiver can cause a denial of service 31

9. Vault managers can grief users 33

10. Lack of zero address check in functions 35

A. Vulnerability Categories **36**

B. Code Maturity Categories **38**

C. Code Quality Recommendations **40**

D. Fee Distribution Redesign **41**

E. Token Integration Checklist **43**

Executive Summary

Engagement Overview

NFTX engaged Trail of Bits to review the security of version 2 of its protocol. From April 4 to April 15, 2022, a team of two consultants conducted a security review of the client-provided source code, with four person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the target system, including access to the source code and documentation for all of the on-chain components. Off-chain code was not included in the scope of this assessment. Also, we were instructed to ignore specific files in the codebase, enumerated in the Project Coverage section.

Summary of Findings

The audit uncovered two flaws that could impact system confidentiality, integrity, or availability. Details on these notable findings are provided below. This report also contains technical appendices. Appendix C lists code quality findings that do not necessarily have security implications. Appendix D provides guidance on redesigning the fee distribution mechanism. Finally, appendix E provides security guidance on interacting with third-party token contracts

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	1
Low	4
Informational	5
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Configuration	1
Data Exposure	1
Data Validation	5
Denial of Service	1
Patching	1

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-NFTX-9**
  The process of creating vaults in the NFTX protocol is trustless. This means that anyone can create a new vault and use any asset as the underlying vault NFT. The user calls the `NFTXVaultFactoryUpgradeable` contract to create a new vault. After deploying the new vault, the contract sets the user as the vault manager.

  Vault managers can change the vault fees and disable certain vault features. Therefore, users must verify that vaults that they interact with have reliable managers or have had their managers disabled through verification.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineers were associated with this project:

Evan Sultanik, Consultant
evan.sultanik@trailofbits.com

Jaime Iglesias, Consultant
jaime.iglesias@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
March 17, 2022	Pre-project architecture onboarding review
March 31, 2022	Pre-project kickoff call
April 8, 2022	Status update meeting #1
April 15, 2022	Delivery of report draft
April 15, 2022	Report readout meeting
May 10, 2022	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of version 2 of the NFTX protocol. Specifically, we sought to answer the following non-exhaustive list of questions:

- Is there a way for users to circumvent fees?
- Can the fee distributor be abused?
- Can a buildup of fees introduce attack vectors?
- Is there a way for a malicious vault manager to grief the vault's users?
- Is there a way for a malicious third-party token to compromise an NFTX vault?

Project Targets

The engagement involved a review and testing of the following target.

NFTX Protocol v2

Repository	https://github.com/NFTX-project/nftx-protocol-v2/
Version	`c8ddc72b4400ad1e12ed03f4369b765371564a00`
Type	Solidity
Platform	Ethereum

Note: Some files in the repository were explicitly out of scope for this assessment. These are enumerated in the following section.

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

NFTXVaultFactoryUpgradeable

This upgradeable contract is one of the centerpieces of the protocol. It is responsible for the deployment and management of vaults. As a vault factory, it is responsible for deploying new vault proxies. It also acts as a beacon proxy: vault proxies query the factory for the address of the vault reference implementation before executing `delegatecall` into it.

This beacon structure allows the NFTX team to upgrade all its vaults at the same time without much friction by simply updating the address of the reference implementation stored by the beacon.

NFTXVaultUpgradeable

This contract is the reference vault implementation and is the core of the NFTX protocol. Users can deposit NFTs (ERC721 and ERC1155) into vaults and receive ERC20 tokens representing claims to the deposited NFTs. This allows users to gain exposure to the floor price of the collection and to provide liquidity on protocols such as automated market makers (AMMs), like Uniswap and Sushiswap. Vaults also allow users to redeem NFTs by burning the corresponding ERC20 tokens and swapping their NFTs for those deposited in a vault.

NFTXInventoryStaking

This upgradeable contract acts as both a factory and a beacon proxy. As a factory, it enables the creation of xToken (also written “XToken” in the codebase) proxies; these ERC20 tokens are wrappers around vault tokens and enable depositors to earn rewards by staking their vault tokens in the inventory staking contract.

As a beacon proxy, similarly to the vault factory, it stores the reference to the xToken implementation contract, which xToken proxies can query before executing `delegatecall` into it.

The contract receives fees from the `NFTXSimpleFeeDistributor` contract.

NFTXLPStaking

This upgradeable contract acts as both a factory and an intermediary. As a factory, it deploys minimal proxies using `CREATE2`, which allows it to compute the addresses of the newly deployed proxies without having to store them.

As an intermediary, it manages the staking functionality, which allows users to deposit liquidity provider (LP) tokens to receive rewards, and is responsible for directing calls to the right proxy contracts by computing their addresses; to do so, it relies on the `StakingTokenProvider` contract, which maps vaults to paired tokens.

The contract receives fees from the `NFTXSimpleFeeDistributor` contract.

NFTXSimpleFeeDistributor

This upgradeable contract is responsible for managing the distribution of protocol fees. The owner (the NFTX DAO in this particular case) is responsible for adding and removing fee receivers. Anyone can perform fee distribution, although it is typically performed by users interacting with the vaults when they perform actions that charge them protocol fees.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. This engagement was scoped to assess only on-chain code. The web-based front end and the protocols for deploying and upgrading contracts warrant a separate review. The following files in the reviewed repository were also explicitly out of scope:

- `contracts/solidity/NFTXV1Buyout.sol`
- `contracts/solidity/tools/NFTXFlashSwipe.sol`
- `contracts/solidity/other/PalmNFTXStakingZap.sol`
- `/contracts/solidity/eligibility/*`
    - **Except for the following file, which was in scope:** `contracts/solidity/eligibility/NFTXRangeEligibility.sol`

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	Solidity 0.8 arithmetic operations are used throughout the system. However, the unit test coverage should be expanded to include all arithmetic edge cases, such as instances in which rewards are calculated or fees are distributed. Additionally, the codebase would benefit from automated testing such as fuzzing.	**Moderate**
Auditing	All critical state-changing operations emit events. However, the team did not provide an incident response plan. We recommend consulting NIST 800-61 and the CISA playbook for guidance on developing a plan.	**Moderate**
Authentication / Access Controls	Privileged access is controlled mainly through the NFTX DAO; other access-related areas of the system, like vault deployment, are trustless. In terms of vault management, the deployer of a vault automatically becomes the manager; however, the deployer can transfer the manager role to the NFTX DAO by setting the manager to the zero address.	**Satisfactory**
Complexity Management	Most of the protocol components are upgradeable. For this purpose, various proxy patterns and implementations are used: • Minimal proxies (EIP-1167) • Transparent proxies	**Weak**

	• Upgradeable beacons • Beacon proxies (CREATE and CREATE2) Most of the contracts have clear purposes, but the control flow is sometimes convoluted, a problem stemming from the lack of documentation. Additionally, old, unused, and undocumented code is still present in part of the codebase, which makes it hard to understand what is and is not relevant to the current version.	
Configuration	This assessment covered only NFTX's on-chain code.	**Not Considered**
Cryptography and Key Management	The NFTX DAO controls most privileged behavior, while some functionality, like the ability to pause certain functions, is controlled by six "guardian wallets." Because the NFTX DAO and the guardian wallets were out of scope for the review, we are not able to evaluate this aspect of the system.	**Not Applicable**
Decentralization	Privileged users control some of the system's functionality (e.g., protocol upgrades); however, in most cases, the NFTX DAO is the privileged user. Additionally, there are a number of "guardian wallets" that can pause the system in the event of an emergency. Because both the DAO and the "guardian wallets" were out of scope for the review, further investigation is required to evaluate this aspect of the system (e.g., voting thresholds, ownership distribution, and the use of multisignature schemes).	**Further Investigation Required**
Documentation	The user documentation is adequate; however, there is a general lack of both inline and developer documentation, which makes it hard to understand the intended behavior of each contract. Legacy storage variables from prior versions of the contracts that are no longer used (yet must remain in the codebase for consistency of the storage layout) are also undocumented (see appendix C).	**Weak**

Front-Running Resistance	Due to time constraints, we did not evaluate this aspect of the system.	**Further Investigation Required**
Low-Level Manipulation	The system uses low-level calls minimally and with the necessary safeguards; however, these instances are not thoroughly documented, making it hard to understand their intended behavior and purpose.	**Moderate**
Testing and Verification	The protocol uses mainnet forking tests for some scenarios; however, we recommend adding coverage support to indicate the parts of the code that have been tested and those that remain uncovered, documenting the current tests, and achieving 100% unit test coverage. Finally, we recommend integrating the unit tests into the CI pipeline and writing fuzzing tests.	**Weak**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Reliance on third-party library for deployment	Configuration	Informational
2	Missing validation of proxy admin indices	Data Validation	Informational
3	Random token withdrawals can be gamed	Data Exposure	Informational
4	Duplicate receivers allowed by addReceiver()	Data Validation	Low
5	OpenZeppelin vulnerability can break initialization	Patching	Informational
6	Potentially excessive gas fees imposed on users for protocol fee distribution	Data Validation	Low
7	Risk of denial of service due to unbounded loop	Data validation	Low
8	A malicious receiver can cause a denial of service	Denial of Service	Low
9	Vault managers can grief users	Access Controls	Medium
10	Lack of zero address check in functions	Data Validation	Informational

Detailed Findings

1. Reliance on third-party library for deployment	
Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-NFTX-1
Target: `NFTX-protocol`	

Description

Due to the use of the `delegatecall` proxy pattern, some NFTX contracts cannot be initialized with their own constructors; instead, they have initializer functions. These functions can be front-run, allowing an attacker to initialize contracts incorrectly.

```
    function __NFTXInventoryStaking_init(address _nftxVaultFactory) external virtual
override initializer {
        __Ownable_init();
        nftxVaultFactory = INFTXVaultFactory(_nftxVaultFactory);
        address xTokenImpl = address(new XTokenUpgradeable());
        __UpgradeableBeacon__init(xTokenImpl);
    }
```

Figure 1.1: The initializer function in `NFTXInventoryStaking.sol:37-42`

The following contracts have initializer functions that can be front-run:

- `NFTXInventoryStaking`
- `NFTXVaultFactoryUpgradeable`
- `NFTXEligibilityManager`
- `NFTXLPStaking`
- `NFTXSimpleFeeDistributor`

The NFTX team relies on `hardhat-upgrades`, a library that offers a series of safety checks for use with certain OpenZeppelin proxy reference implementations to aid in the proxy deployment process. It is important that the NFTX team become familiar with how the `hardhat-upgrades` library works internally and with the caveats it might have. For example, some proxy patterns like the beacon pattern are not yet supported by the library.

Exploit Scenario

Bob uses the library incorrectly when deploying a new contract: he calls `upgradeTo()` and then uses the `fallback` function to initialize the contract. Eve front-runs the call to the initialization function and initializes the contract with her own address, which results in an incorrect initialization and Eve's control over the contract.

Recommendations

Short term, document the protocol's use of the library and the proxy types it supports.

Long term, use a factory pattern instead of the initializer functions to prevent front-running of the initializer functions.

2. Missing validation of proxy admin indices	
Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-NFTX-2
Target: `ProxyController.sol`	

Description

Multiple functions of the `ProxyController` contract take an index as an input. The index determines which proxy (managed by the controller) is being targeted.

However, the index is never validated, which means that the function will be executed even if the index is out of bounds with respect to the number of proxies managed by the contract (in this case, five).

```
function changeProxyAdmin(uint256 index, address newAdmin)
    public
    onlyOwner
{
    if (index == 0) {
        vaultFactoryProxy.changeAdmin(newAdmin);
    } else if (index == 1) {
        eligManagerProxy.changeAdmin(newAdmin);
    } else if (index == 2) {
        stakingProviderProxy.changeAdmin(newAdmin);
    } else if (index == 3) {
        stakingProxy.changeAdmin(newAdmin);
    } else if (index == 4) {
        feeDistribProxy.changeAdmin(newAdmin);
    }
    emit ProxyAdminChanged(index, newAdmin);
}
```

Figure 2.1: The `changeProxyAdmin` function in `ProxyController.sol:79-95`

In the `changeProxyAdmin` function, a `ProxyAdminChanged` event is emitted even if the supplied index is out of bounds (figure 2.1).

Other `ProxyController` functions return the zero address if the index is out of bounds. For example, `getAdmin()` should return the address of the targeted proxy's admin. If `getAdmin()` returns the zero address, the caller cannot know whether she supplied the wrong index or whether the targeted proxy simply has no admin.

```
function getAdmin(uint256 index) public view returns (address admin) {
    if (index == 0) {
        return vaultFactoryProxy.admin();
    } else if (index == 1) {
        return eligManagerProxy.admin();
    } else if (index == 2) {
        return stakingProviderProxy.admin();
    } else if (index == 3) {
        return stakingProxy.admin();
    } else if (index == 4) {
        return feeDistribProxy.admin();
    }
}
```

Figure 2.2: The `getAdmin` function in `ProxyController.sol:38-50`

Exploit Scenario

A contract relying on the `ProxyController` contract calls one of the view functions, like `getAdmin()`, with the wrong index. The function is executed normally and implicitly returns zero, leading to unexpected behavior.

Recommendations

Short term, document this behavior so that clients are aware of it and are able to include safeguards to prevent unanticipated behavior.

Long term, consider adding an index check to the affected functions so that they revert if they receive an out-of-bounds index.

3. Random token withdrawals can be gamed

Severity: **Informational**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-NFTX-3
Target: `NFTXVaultUpgradeable.sol`	

Description

The algorithm used to randomly select a token for withdrawal from a vault is deterministic and predictable.

```
function getRandomTokenIdFromVault() internal virtual returns (uint256) {
    uint256 randomIndex = uint256(
        keccak256(
            abi.encodePacked(
                blockhash(block.number - 1),
                randNonce,
                block.coinbase,
                block.difficulty,
                block.timestamp
            )
        )
    ) % holdings.length();
    ++randNonce;
    return holdings.at(randomIndex);
}
```

Figure 3.1: The `getRandomTokenIdFromVault` function in `NFTXVaultUpgradable.sol:531-545`

All the elements used to calculate `randomIndex` are known to the caller (figure 3.1). Therefore, a contract calling this function can predict the resulting token before choosing to execute the withdrawal.

This finding is of high difficulty because NFTX's vault economics incentivizes users to deposit tokens of equal value. Moreover, the cost of deploying a custom exploit contract will likely outweigh the fee savings of choosing a token at random for withdrawal.

Exploit Scenario

Alice wishes to withdraw a specific token from a vault but wants to pay the lower "random redemption fee" rather than the higher "target redemption fee." She deploys a contract that checks whether the randomly chosen token is her target and, if so, automatically executes the random withdrawal.

Recommendations

Short term, document the risks described in this finding so that clients are aware of them.

Long term, consider removing all randomness from NFTX.

4. Duplicate receivers allowed by addReceiver()

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NFTX-4
Target: `NFTXSimpleFeeDistributor.sol`	

Description

The `NFTXSimpleFeeDistributor` contract is in charge of protocol fee distribution. To facilitate the fee distribution process, it allows the contract owner (the NFTX DAO) to manage a list of fee receivers.

To add a new fee receiver to the contract, the owner calls the `addReceiver()` function.

```
  function addReceiver(
    uint256 _allocPoint,
    address _receiver,
    bool _isContract
  ) external   override virtual onlyOwner  {
    _addReceiver(_allocPoint, _receiver, _isContract);
  }
```

Figure 4.1: The `addReceiver()` function in `NFTXSimpleFeeDistributor`

This function in turn executes the internal logic that pushes a new receiver to the receiver list.

```
  function _addReceiver(
    uint256 _allocPoint,
    address _receiver,
    bool _isContract
  ) internal virtual {
    FeeReceiver memory _feeReceiver = FeeReceiver(_allocPoint, _receiver,
_isContract);
    feeReceivers.push(_feeReceiver);
    allocTotal += _allocPoint;
    emit AddFeeReceiver(_receiver, _allocPoint);
  }
```

Figure 4.2: The `_addReceiver()` function in `NFTXSimpleFeeDistributor`

However, the function does not check whether the receiver is already in the list. Without this check, receivers can be accidentally added multiple times to the list, which would increase the amount of fees they receive.

The issue is of high difficulty because the `addReceiver()` function is owner-protected and, as indicated by the NFTX team, the owner is the NFTX DAO. Because the DAO itself was out of scope for this review, we do not know what the process to become a receiver looks like. We assume that a DAO proposal has to be created and a certain quorum has to be met for it to be executed.

Exploit Scenario

A proposal is created to add a new receiver to the fee distributor contract. The receiver address was already added, but the DAO members are not aware of this. The proposal passes, and the receiver is added. The receiver gains more fees than he is entitled to.

Recommendations

Short term, add a duplicate check to the `_addReceiver()` function.

Long term, document this behavior so that the NFTX DAO is aware of it and performs the adequate checks before adding a new receiver.

5. OpenZeppelin vulnerability can break initialization	
Severity: **Informational**	Difficulty: **Low**
Type: Patching	Finding ID: TOB-NFTX-5
Target: `package.json`	

Description
NFTX extensively uses OpenZeppelin v3.4.1. A bug was recently discovered in all OpenZeppelin versions prior to v4.4.1 that affects initializer functions invoked separately during contract creation: the bug causes the contract initialization modifier to fail to prevent reentrancy to the initializers (see CVE-2021-46320).

Currently, no external calls to untrusted code are made during contract initialization. However, if the NFTX team were to add a new feature that requires such calls to be made, it would have to add the necessary safeguards to prevent reentrancy.

Exploit Scenario
An NFTX contract initialization function makes a call to an external contract that calls back to the initializer with different arguments. The faulty OpenZeppelin `initializer` modifier fails to prevent this reentrancy.

Recommendations
Short term, upgrade OpenZeppelin to v4.4.1 or newer.

Long term, integrate a dependency checking tool like Dependabot into the NFTX CI process.

6. Potentially excessive gas fees imposed on users for protocol fee distribution

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-NFTX-6
Target: `NFTXSimpleFeeDistributor.sol`	

Description

Whenever a user executes a minting, redeeming, or swapping operation on a vault, a fee is charged to the user and is sent to the `NFXTSimpleFeeDistributor` contract for distribution.

```
    function _chargeAndDistributeFees(address user, uint256 amount) internal virtual
{
        // Do not charge fees if the zap contract is calling
        // Added in v1.0.3. Changed to mapping in v1.0.5.

        INFTXVaultFactory _vaultFactory = vaultFactory;

        if (_vaultFactory.excludedFromFees(msg.sender)) {
            return;
        }

        // Mint fees directly to the distributor and distribute.
        if (amount > 0) {
            address feeDistributor = _vaultFactory.feeDistributor();
            // Changed to a _transfer() in v1.0.3.
            _transfer(user, feeDistributor, amount);
            INFTXFeeDistributor(feeDistributor).distribute(vaultId);
        }
    }
```

Figure 6.1: The `_chargeAndDistributeFees()` function in `NFTXVaultUpgradeable.sol`

After the fee is sent to the `NFXTSimpleFeeDistributor` contract, the `distribute()` function is then called to distribute all accrued fees.

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

    uint256 tokenBalance = IERC20Upgradeable(_vault).balanceOf(address(this));
```

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

    uint256 length = feeReceivers.length;
    uint256 leftover;
    for (uint256 i; i < length; ++i) {
      FeeReceiver memory _feeReceiver = feeReceivers[i];
      uint256 amountToSend = leftover + ((tokenBalance * _feeReceiver.allocPoint) /
allocTotal);
      uint256 currentTokenBalance =
IERC20Upgradeable(_vault).balanceOf(address(this));
      amountToSend = amountToSend > currentTokenBalance ? currentTokenBalance :
amountToSend;
      bool complete = _sendForReceiver(_feeReceiver, vaultId, _vault, amountToSend);
      if (!complete) {
        uint256 remaining = IERC20Upgradeable(_vault).allowance(address(this),
_feeReceiver.receiver);
        IERC20Upgradeable(_vault).safeApprove(_feeReceiver.receiver, 0);
        leftover = remaining;
      } else {
        leftover = 0;
      }
    }

    if (leftover != 0) {
      uint256 currentTokenBalance =
IERC20Upgradeable(_vault).balanceOf(address(this));
      IERC20Upgradeable(_vault).safeTransfer(treasury, currentTokenBalance);
    }
  }
```

Figure 6.2: The `distribute()` function in `NFTXSimpleFeeDistributor.sol`

If the token balance of the contract is low enough (but not zero), the number of tokens distributed to each receiver (`amountToSend`) will be close to zero.

Ultimately, this can disincentivize the use of the protocol, regardless of the number of tokens distributed. Users have to pay the gas fee for the fee distribution operation, the gas fees for the token operations (e.g., redeeming, minting, or swapping), and the protocol fees themselves.

Exploit Scenario

Alice redeems a token from a vault, pays the necessary protocol fee, sends it to the `NFTXSimpleFeeDistributor` contract, and calls the `distribute()` function. Because

the balance of the distributor contract is very low (e.g., $0.50), Alice has to pay a substantial amount in gas to distribute a near-zero amount in fees between all fee receiver addresses.

Recommendations

Short term, add a requirement for a minimum balance that the `NFTXSimpleFeeDistributor` contract should have for the distribution operation to execute. Alternatively, implement a periodical distribution of fees (e.g., once a day or once every number of blocks).

Long term, consider redesigning the fee distribution mechanism to prevent the distribution of small fees. Also consider whether protocol users should pay for said distribution. See appendix D for guidance on redesigning this mechanism.

7. Risk of denial of service due to unbounded loop

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NFTX-7
Target: `NFTXSimpleFeeDistributor.sol`	

Description

When protocol fees are distributed, the system loops through the list of beneficiaries (known internally as receivers) to send them the protocol fees they are entitled to.

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

    uint256 tokenBalance = IERC20Upgradeable(_vault).balanceOf(address(this));

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

    uint256 length = feeReceivers.length;
    uint256 leftover;
    for (uint256 i; i < length; ++i) {
      FeeReceiver memory _feeReceiver = feeReceivers[i];
      uint256 amountToSend = leftover + ((tokenBalance * _feeReceiver.allocPoint) /
allocTotal);
      uint256 currentTokenBalance =
IERC20Upgradeable(_vault).balanceOf(address(this));
      amountToSend = amountToSend > currentTokenBalance ? currentTokenBalance :
amountToSend;
      bool complete = _sendForReceiver(_feeReceiver, vaultId, _vault, amountToSend);
      if (!complete) {
        uint256 remaining = IERC20Upgradeable(_vault).allowance(address(this),
_feeReceiver.receiver);
        IERC20Upgradeable(_vault).safeApprove(_feeReceiver.receiver, 0);
        leftover = remaining;
      } else {
        leftover = 0;
      }
    }

    if (leftover != 0) {
      uint256 currentTokenBalance =
IERC20Upgradeable(_vault).balanceOf(address(this));
      IERC20Upgradeable(_vault).safeTransfer(treasury, currentTokenBalance);
```

```
    }
  }
```

Figure 7.1: The `distribute()` function in `NFTXSimpleFeeDistributor.sol`

Because this loop is unbounded and the number of receivers can grow, the amount of gas consumed is also unbounded.

```
  function _sendForReceiver(FeeReceiver memory _receiver, uint256 _vaultId, address
_vault, uint256 amountToSend) internal virtual returns (bool) {
    if (_receiver.isContract) {
      IERC20Upgradeable(_vault).safeIncreaseAllowance(_receiver.receiver,
amountToSend);

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

      // If the allowance has not been spent, it means we can pass it forward to
next.
      return success && IERC20Upgradeable(_vault).allowance(address(this),
_receiver.receiver) == 0;
    } else {
      IERC20Upgradeable(_vault).safeTransfer(_receiver.receiver, amountToSend);
      return true;
    }
  }
```

Figure 7.2: The `_sendForReceiver()` function in `NFTXSimpleFeeDistributor.sol`

Additionally, if one of the receivers is a contract, code that significantly increases the gas cost of the fee distribution will execute (figure 7.2).

It is important to note that fees are usually distributed within the context of user transactions (redeeming, minting, etc.), so the total cost of the distribution operation depends on the logic outside of the `distribute()` function.

The issue is of low severity because there are currently only two fee receivers and there are no plans to add a significant number more. If there were more fee receivers or the ability to quickly add fee receivers, then the severity of this finding would be higher.

The issue is of high difficulty because the DAO has the ability to add new fee receivers, so an attacker with control of the DAO could exploit this issue.

Exploit Scenario

The NFTX team adds a new feature that allows NFTX token holders who stake their tokens to register as receivers and gain a portion of protocol fees; because of that, the number of receivers grows dramatically. Due to the large number of receivers, the `distribute()` function cannot execute because the cost of executing it has reached the block gas limit. As a result, users are unable to mint, redeem, or swap tokens.

Recommendations

Short term, examine the execution cost of the function to determine the safe bounds of the loop and, if possible, consider splitting the distribution operation into multiple calls.

Long term, consider redesigning the fee distribution mechanism to avoid unbounded loops and prevent denials of service. See appendix D for guidance on redesigning this mechanism.

8. A malicious fee receiver can cause a denial of service

Severity: **Low**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-NFTX-8
Target: `NFTXSimpleFeeDistributor.sol`	

Description

Whenever a user executes a minting, redeeming, or swapping operation on a vault, a fee is charged to the user and is sent to the `NFXTSimpleFeeDistributor` contract for distribution. The distribution function loops through all fee receivers and sends them the number of tokens they are entitled to (see figure 7.1).

If the fee receiver is a contract, a special logic is executed; instead of receiving the corresponding number of tokens, the receiver pulls all the tokens from the `NFXTSimpleFeeDistributor` contract.

```
  function _sendForReceiver(FeeReceiver memory _receiver, uint256 _vaultId, address
_vault, uint256 amountToSend) internal virtual returns (bool) {
    if (_receiver.isContract) {
      IERC20Upgradeable(_vault).safeIncreaseAllowance(_receiver.receiver,
amountToSend);

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

      // If the allowance has not been spent, it means we can pass it forward to
next.
      return success && IERC20Upgradeable(_vault).allowance(address(this),
_receiver.receiver) == 0;
    } else {
      IERC20Upgradeable(_vault).safeTransfer(_receiver.receiver, amountToSend);
      return true;
    }
  }
```

Figure 8.1: The `_sendForReceiver()` function in `NFTXSimpleFeeDistributor.sol`

In this case, because the receiver contract executes arbitrary logic and receives all of the gas, the receiver contract can spend all of it; as a result, only 1/64 of the original gas forwarded to the receiver contract would remain to continue executing the `distribute()`

function (see EIP-150), which may not be enough to complete the execution, leading to a denial of service.

The issue is of low severity because all of the current fee receivers are contracts created and owned by NFTX and there are no plans to add a significant number more. If there were external fee receivers or the ability to quickly add more, then the severity of this finding would be higher.

The issue is of high difficulty because the `addReceiver()` function is owner-protected and, as indicated by the NFTX team, the owner is the NFTX DAO. Because the DAO itself was out of scope for this review, we do not know what the process to become a receiver looks like. We assume that a proposal is created and a certain quorum has to be met for it to be executed.

Exploit Scenario

Eve, a malicious receiver, sets up a smart contract that consumes all the gas forwarded to it when `receiveRewards` is called. As a result, the `distribute()` function runs out of gas, causing a denial of service on the vaults calling the function.

Recommendations

Short term, change the fee distribution mechanism so that only a token transfer is executed even if the receiver is a contract.

Long term, consider redesigning the fee distribution mechanism to prevent malicious fee receivers from causing a denial of service on the protocol. See appendix D for guidance on redesigning this mechanism.

9. Vault managers can grief users

Severity: **Medium**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-NFTX-9
Target: `NFTXVaultUpgradeable.sol`	

Description

The process of creating vaults in the NFTX protocol is trustless. This means that anyone can create a new vault and use any asset as the underlying vault NFT.

The user calls the `NFTXVaultFactoryUpgradeable` contract to create a new vault. After deploying the new vault, the contract sets the user as the vault manager. Vault managers can disable certain vault features (figure 9.1) and change vault fees (figure 9.2).

```
    function setVaultFeatures(
        bool _enableMint,
        bool _enableRandomRedeem,
        bool _enableTargetRedeem,
        bool _enableRandomSwap,
        bool _enableTargetSwap
    ) public override virtual {
        onlyPrivileged();
        enableMint = _enableMint;
        enableRandomRedeem = _enableRandomRedeem;
        enableTargetRedeem = _enableTargetRedeem;
        enableRandomSwap = _enableRandomSwap;
        enableTargetSwap = _enableTargetSwap;

        emit EnableMintUpdated(_enableMint);
        emit EnableRandomRedeemUpdated(_enableRandomRedeem);
        emit EnableTargetRedeemUpdated(_enableTargetRedeem);
        emit EnableRandomSwapUpdated(_enableRandomSwap);
        emit EnableTargetSwapUpdated(_enableTargetSwap);
    }
```

Figure 9.1: The `setVaultFeatures()` function in `NFTXVaultUpgradeable.sol`

```
    function setFees(
        uint256 _mintFee,
        uint256 _randomRedeemFee,
        uint256 _targetRedeemFee,
```

```
        uint256 _randomSwapFee,
        uint256 _targetSwapFee
    ) public override virtual {
        onlyPrivileged();
        vaultFactory.setVaultFees(
          vaultId,
          _mintFee,
          _randomRedeemFee,
          _targetRedeemFee,
          _randomSwapFee,
          _targetSwapFee
        );
    }
```

Figure 9.2: The `setFees()` function in `NFTXVaultUpgradeable.sol`

The effects of these functions are instantaneous, which means users may not be able to react in time to these changes and exit the vaults. Additionally, disabling vault features with the `setVaultFeatures()` function can trap tokens in the contract.

Ultimately, this risk is related to the trustless nature of vault creation, but the NFTX team can take certain measures to minimize the effects. One such measure, which is already in place, is "vault verification," in which the vault manager calls the `finalizeVault()` function to pass her management rights to the zero address. This function then gives the "verified" status to the vault in the NFTX web application.

Exploit Scenario

Eve, a malicious manager, creates a new vault for a popular NFT collection. After it gains some user traction, she unilaterally changes the vault fees to the maximum (0.5 ether), which forces users to either pay the high fee or relinquish their tokens.

Recommendations

Short term, document the risks of interacting with vaults that have not been finalized (i.e., vaults that have managers).

Long term, consider adding delays to manager-only functionality (e.g., a certain number of blocks) so that users have time to react and exit the vault.

10. Lack of zero address check in functions	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NFTX-10
Target: Several contracts	

Description
Certain setter functions fail to validate incoming arguments, so callers can accidentally set important state variables to the zero address.

This issue affects the following contracts and functions:

- `NFTXInventoryStaking.sol`
  - `__NFTXInventoryStaking_init()`
- `NFTXSimpleFeeDistributor.sol`
  - `setInventoryStakingAddress()`
  - `addReceiver()`
  - `changeReceiverAddress()`
- `RewardDistributionToken`
  - `__RewardDistributionToken_init()`

Exploit Scenario
Alice deploys a new version of the `NFTXInventoryStaking` contract. When she initializes the proxy contract, she inputs the zero address as the address of the `_nftxVaultFactory` state variable, leading to an incorrect initialization.

Recommendations
Short term, add zero-value checks on all function arguments to ensure that users cannot accidentally set incorrect values, misconfiguring the system.

Long term, use Slither, which will catch functions that do not have zero checks.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.

Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Recommendations

- **Declare all the interfaces in the `interface` folder.** Some token interfaces are in the `token/` folder instead of the `interface/` folder:
    - `IERC1155ReceiverUpgradeable.sol`
    - `IERC1155Upgradeable.sol`
    - `IERC20Metadata.sol`
    - `IERC20Upgradeable.sol`
    - `IERC721Enumerable.sol`
    - `IERC721ReceiverUpgradeable.sol`
    - `IERC721Upgradeable.sol`
- **Document why `rewardDistTokenImpl` in `NFTXLPStaking.sol#24` is never used.** The unused variable is error-prone. If this is an artifact of a previous contract's version, it must be documented to prevent its misuse.
- **Use consistent nomenclature for vault addresses.** Throughout the codebase, the addresses of vaults are referred to as both `vaultAddrs` and `baseTokens` interchangeably. This overloading is confusing; we recommend using `vaultAddr` to refer to all vault addresses.

D. Fee Distribution Redesign

When users execute certain actions on a vault (e.g., minting, redeeming, and swapping), they have to pay a fee. This fee is sent to a distribution contract, which splits it between the receivers. If there are any tokens leftover after distribution, if there are no receivers, or if the fee is disabled for the vault, then the tokens are sent to the NFTX treasury contract.

Several findings that resulted from the audit relate to the fee distribution process; some of these findings can lead to users being disincentivized to use the system (TOB-NFTX-6) or trap the system (TOB-NFTX-7, TOB-NFTX-8). Because of these findings, we strongly recommend that the NFTX team redesign this portion of the protocol. For that purpose, we offer the following recommendations:

- **Avoid unbounded loops.**
  Because of gas limitations, using unbounded loops can be risky. If using a loop is unavoidable, then try to limit its size or consider splitting the function into multiple transactions so that the loop is bound within safe limits with each execution.

  In this case, the distribution contract would not loop through the receivers to send them their tokens; instead, the receivers themselves would call the contract to receive them. That way, the receivers would have to pay only for the receipt of the fees they are entitled to. This would remove the risk of griefing attacks by other receivers.

- **Avoid the execution of arbitrary bytecode.**
  As described in TOB-NFTX-8, when a receiver is a contract, a call is made to it and arbitrary code is executed. Because the distributor contact has no control over this code, unpredictable behavior may occur and additional safeguards have to be implemented.

  This risk is amplified in the current implementation because fees are distributed to all receivers at the same time, which could allow a malicious receiver contract to conduct a denial-of-service attack.

- **Rely on incentives.**
  Currently, users interacting with the vaults initiate the fee distribution process when they execute a redeeming, swapping, or minting operation. This means that users eat the cost of executing the distribution and may be disincentivized to use the protocol. On the other hand, the receivers naturally have an incentive to perform the distribution, unless the cost is higher than the value of the fees they will receive.

We recommend redesigning the fee distribution mechanism with this incentive in mind.

E. Token Integration Checklist

Since NFTX vaults integrate with external token contracts, it is incumbent upon vault deployers to ensure that those tokens are not malicious or faulty.

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. For an up-to-date version of the checklist, see `crytic/building-secure-contracts`.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken --erc erc20
slither-check-erc 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d KittyCore --erc erc721
```

To follow this checklist, use the below output from Slither for the token:

```
slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
slither [target] --print human-summary
slither [target] --print contract-summary
slither-prop . --contract ContractName # requires configuration, and use of Echidna
and Manticore
```

General Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on `blockchain-security-contacts`.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.
- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.

- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.

- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine whether the contract is upgradeable.

- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.

- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.

- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.

- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

ERC20 Tokens

ERC20 Conformity Checks

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.

- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.

- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.

- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from

stealing tokens.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

- ❑ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Risks of ERC20 Extensions

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❑ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.
- ❑ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❑ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❑ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❑ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❑ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❑ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❑ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

ERC721 Tokens

ERC721 Conformity Checks

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❑ **Transfers of tokens to the `0x0` address revert.** Several tokens allow transfers to `0x0` and consider tokens transferred to that address to have been burned; however, the ERC721 standard requires that such transfers revert.
- ❑ **`safeTransferFrom` functions are implemented with the correct signature.** Several token contracts do not implement these functions. A transfer of NFTs to one of those contracts can result in a loss of assets.
- ❑ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC721 standard and may not be present.
- ❑ **If it is used, `decimals` returns a `uint8(0)`.** Other values are invalid.
- ❑ **The `name` and `symbol` functions can return an empty string.** This behavior is allowed by the standard.
- ❑ **The `ownerOf` function reverts if the `tokenId` is invalid or is set to a token that has already been burned.** The function cannot return `0x0`. This behavior is required by the standard, but it is not always properly implemented.
- ❑ **A transfer of an NFT clears its approvals.** This is required by the standard.
- ❑ **The token ID of an NFT cannot be changed during its lifetime.** This is required by the standard.

Common Risks of ERC721 Tokens

To mitigate the risks associated with ERC721 contracts, conduct a manual review of the following conditions:

- ❑ **The `onERC721Received` callback is taken into account.** External calls in the transfer functions can lead to reentrancies, especially when the callback is not explicit (e.g., in `safeMint` calls).

- ❏ **When an NFT is minted, it is safely transferred to a smart contract.** If there is a minting function, it should behave similarly to `safeTransferFrom` and properly handle the minting of new tokens to a smart contract. This will prevent a loss of assets.
- ❏ **The burning of a token clears its approvals.** If there is a burning function, it should clear the token's previous approvals.
- ❏ **Token-specific logic is sound, and the contracts account for unexpected behavior.** Some non-standard ERC721 tokens allow users to own only a single NFT. This means that transfer functions may also burn tokens