NFTX Report No. 1

Prepared by Level K

Nov, 2020

Chapter 1

Introduction

1.1 Scope of Work

This code review was prepared by Level K at the request of Alex Gausman on behalf of `NFTX`, a project aiming to create derivative crypto-assets collateralized by "Non-Fungible Tokens" (or "NFTs"). The code covered by this review (see section 1.2) is a subset of the DAO's governance logic. This review was conducted over nine engineer-days.

1.2 Source Files

This audit covers code from a private `git` repository. The commit hash of the reviewed code is as follows:

`f80a9c1c65f75b433346196c1bcd12a5d1bbea9c`

Within that revision, only the following files received line-by-line review:

- contracts/XStore.sol
- contracts/NFTX.sol

This review was conducted under the optimistic assumption that all of the supporting software infrastructure necessary for the deployment and operation of the reviewed code works as intended. There may be critical defects in code outside of the scope of this review that could render deployed smart contracts inoperable or exploitable.

1.3 License and Disclaimer of Warranty

This source code review is not an endorsement of the code or its suitability for any legal/regulatory regime, and it is not intended as a definitive or exhaustive list of defects. This document is provided expressly for the benefit of NFTX developers and only under the following terms:

Chapter 2

Critical Defects

Issues discussed in this section are defects that lead to the code to misbehave in ways that are directly exploitable. Critical issues can have severe consequences, such as loss of funds.

2.1 Attacker-controlled Token Address

`NFTX.createVault()` accepts an arbitrary address as the `XToken` that the vault will use for pro-rata ownership shares of the NFT pool. The only validation that the code attempts on the attacker-controlled address is to call `owner()` and check that it returns the address of the `NFTX` contract, which is a behavior that is trivially spoof-able. Consequently, an attacker can provide the address of a contract that implements the `XToken` interface in a malicious manner. For example, the attacker-controlled `XToken` could allow the attacker to mint themselves enough tokens to drain the associated NFT `Vault`. When combined with some social engineering, it is plausible that this defect could be used to steal NFTs from contributors to the `Vault`.

`NFTX.createVault()` should probably deploy a fresh `XToken` contract when it is called.

Note that the address of the implementation of the non-fungible token is also attacker-controlled in `createVault()`, but it may be reasonable to assume that users will not interact with `Vault`s that manage unknown NFTs.

Chapter 3

Moderate Defects

Issues discussed in this section are code defects that may lead to unintended deviations in behavior. It may be possible to chain multiple moderate defects into a working exploit.

3.1 Incorrect Index in `_redeem()`

The expression `nftIds[i]` on line 273 of `NFTX.sol` uses the wrong index into the `nftIds[]` array.

```
uint256[] memory nftIds = new uint256[](1);
if (store.holdingsLength(vaultId) > 0) {
    uint256 rand = _getPseudoRand(store.holdingsLength(vaultId));
    nftIds[0] = store.holdingsAt(vaultId, rand);
} else {
    uint256 rand = _getPseudoRand(store.reservesLength(vaultId));
    nftIds[i] = store.reservesAt(vaultId, rand); // <-- incorrect in-
dex
}
```

The correct expression is `nftIds[0]`, as the length of `nftIds` is always one.

The impact of this bug is that redemptions of more than one NFT when there are no more non-reserved NFTs in a `Vault` will erroneously revert due to an out-of-bounds array access.

3.2 Vault Manager Fee Theft and Frontrunning

Before a `Vault` is "finalized," the original creator of a `Vault` (called a "manager") has unlimited lattitude over the fees and bounties associated with a `Vault`. Of particular concern is the fact that the `Vault` manager can set fees relatively high (and bounties relatively low) under ordinary conditions, and then within a single transaction radically increase the bounty and eliminate fees and then **claim the entire balance** of the `Vault` by depositing and then removing a single NFT. The "manager" can also grief other users of the `Vault` by front-running their transactions with calls to `setIsEligible()` that will invalidate particular exchanges. This sort of administrative latitude may discourage users in interacting with a `Vault` that has not yet been "finalized."

See also issues section 5.4 and section 3.5.

3.3 Bounty Calculation: Integer Truncation

In `NFTX.sol` on line 255, the expression that calculates the bounty amount performs division before multiplication, which means that the low-order bits of `ethMax` may disappear due to roundoff error.

```
(uint256 ethMax, uint256 length) = store.supplierBounty(vaultId);
if (_vaultSize >= length) return 0;
uint256 depth = length.sub(_vaultSize);
return ethMax.div(length).mul(depth); // <-- truncation here
```

In the specific case that `length` is greater than `ethMax`, the result of `_calcBountyHelper()` will always be zero.

The expression `ethMax.mul(depth).div(length)` produces the desired result without losing precision.

3.4 Unsafe ERC20 Transfers

The code on line 253 of `NFTX.sol` calls `ERC20.transferFrom()` without checking for a return value.

```
store.d2Asset(vaultId).transferFrom(
    _msgSender(),
    address(this),
    amount
);
```

Similarly, line 281 of `NFTX.sol` calls `ERC20.transfer()` without checking for a return value:

```
store.d2Asset(vaultId).transfer(_msgSender(), amount);
```

Early implementations of ERC20 tokens **do not revert** on invalid transfers, and instead return `false`. For safety, callers of `ERC20.transfer()` and `ERC20.transferFrom()` need to check return values.

Consider using OpenZeppelin's `SafeERC20` library for calling methods on ERC20 tokens.

3.5 Accumulation of Stuck Ether

Ether that is sent to the `NFTX` contract in order to pay for fees may be in excess of the actual fee amount. If multiple transactions to call `NFTX.mint()` are sent by different addresses around the same time, the ordering of those transactions is indeterminate, and therefore the fees charged by the `NFTX` contract are also somewhat unpredictable. Participants that over-pay Ether for fees are not refunded their over-payment, and furthermore there is no way to withdraw Ether from the `NFTX` contract that isn't assigned to a `Vault` explicitly via `_receiveEthToVault()`. (Note that `_receiveEthToVault()` only assigns exactly the fee amount to the vault; it does not handle any overpayment.) Consequently, it is likely that Ether will get "stuck" in the `NFTX` contract.

Chapter 4

Minor Defects

Issues discussed in this section are subjective code defects that affect readability, reliability, or performance.

4.1 Use of Functions as Modifiers

For the sake of adhering to idiomatic solidity code style, the following functions should be modifiers instead of functions:

- `onlyExtension()`
- `onlyManager()`
- `onlyPrivileged()`
- `onlyOwnerIfPaused()`

4.2 Unnecessary Use of `virtual` Keyword

Since the `NFTX` contract is not inherited by any other contracts, it is unnecessary to adorn public functions with the `virtual` keyword. The `virtual` keyword is only relevant to code that expects its methods to be overridden.

4.3 Unnecessary Arithmetic Checks in Loops

For an induction variable `i`, which is a 256-bit integer that starts at zero and is incremented once per loop iteration, cannot overflow within the gas block limit. In other words, the code can safely substitute `i++` for `i.add(1)` where `i` is a 256-bit wide integer induction variable.

4.4 `mintAndRedeem()` Can Return Provided NFTs

`NFTX.mintAndRedeem()` can end up returning the same NFTs provided to the contract during the `_redeem()` step if the result from `_getPsuedoRand()` collides. The likelihood of this collision will depend on the number of NFTs present in the `Vault` at the time the call to `mintAndRedeem()` is made.

4.5 Unnecessary State Indirection

Using `XStore` as storage for the `NFTX` contract adds a lot of additional lines of code and significant gas overhead. The `UpgradeabilityProxy` contract already gives you state that can have its code upgraded; there is no need to re-implement this functionality explicitly. The OpenZeppelin proxy code uses `delegatecall` to run arbitrary code that modifies the proxy contract's state, so it is already possible for the code to upgrade its business logic without modifying the internal contract storage. For more information, see the OpenZeppelin proxy documentation.

Refactoring the code to avoid this indirection would likely eliminate more than one third of the lines of code covered in this review. It would also eliminate a substantial portion of the gas overhead of each interaction with the `NFTX` contract.

4.6 Consider Splitting Vault Structure

The code in `NFTX.sol` and `XStore.sol` uses the same `Vault` data structure to represent two distinct concepts: a `Vault` collateralized by one Non-Fungible Token, and a `Vault` collateralized by a (fungible) ERC20 token. Consequently, the code has to carefully evaluate whether each and every operation performed on the `Vault` is contextually correct depending on its mode of operation.

The code would likely be significantly simpler if the `Vault` structure was replaced with two distinct fit-for-purpose data structures.

4.7 Extensions Can Violate Balance Invariants

Code that the owner of the `NFTX` contract designates as "extensions" (by calling `setExtension()`) can call `directRedeem()`. Consequently, any "extension" code can remove NFTs without a guarantee that it burns the correct number of `XToken`s. Any code granted these privileges will need to be audited separately to determine whether or not it obeys the correct invariants.

See also section 4.10 for further discussion about guaranteeing invariants.

4.8 Accumulation of Stuck Tokens

The implementation of `onERC721Received()` that `NFTX` inherits from `ERC721Holder` accepts all ERC721 transfers unconditionally. If a user accidentally transfers an ERC721 token to the NFTX contract, the token will be permanently stuck.

Consider adding functionality for the owner of the NFTX contract to transfer ERC721 tokens owned by the contract that are not members of any Vault. Alternatively, consider changing the implementation of onERC721Received() so that it refuses transfers that are not initiated as part of a mint() operation.

4.9 Suboptimal State-change Ordering

In general, it is safest to perform calls to foreign contracts (those that live at an arbitrary address) **after** internal state changes in order to avoid the impact of a reentrancy bug.

Consider the following code on lines 253 throug 259 of NFTX.sol:

```
store.d2Asset(vaultId).transferFrom(
    _msgSender(),
    address(this),
    amount
);
store.xToken(vaultId).mint(_msgSender(), amount);
store.setD2Holdings(vaultId, store.d2Holdings(vaultId).add(amount));
```

Since store.d2Asset(vaultId) may return the address of a contract that isn't controlled directly by the Vault manager or the NFTX DAO, the implementation of transferFrom() cannot be relied upon not to make additional external calls itself. Consequently, these three lines of code could conceivably be attacked with a reentrancy exploit. Currently, it does not appear that this code can be reached from an external entry point without passing through a reentrancy guard, but it would still be safest to adhere to the "checks, effects, interactions" convention in case a future refactoring introduces code that touches this code path without a reentrancy guard. In this particular case, the external call on lines 253 through 257 can simply be inserted beneath the calls to the XStore contract. There are a number of other places in the NFTX contract where external code is called *before* internal state changes, and out of an abundance of caution those places should be fixed as well.

4.10 Missing Invariants Checks

One of the explicit Vault invariants in the NFTX documentation provided to Level K indicates that the total supply of the XToken for a vault should always be equivalent to the total supply of the Vault NFT. To ensure that this invariant is preserved across external calls, the NFTX contract should employ a modifier for external calls that asserts the appropriate condition. Something similar to the following code could be applied to functions that deal only with ordinary (non-"D2") vaults:

```
// the balanced(vault) modifier ensures that
// the XToken for 'vault' has exactly as many
// circulating tokens as the number of NFTs
// in the vault
modifier balanced(uint256 vault) {
    _;
    uint256 xt = store.xToken(vault).totalSupply();
    uint256 nt = store.nft(vault).balanceOf(address(this));
    require(xt == nt.mul(10**store.xToken(vault).decimals()));
}

// a simplified 'mint()' that deals with regular Vaults:
function mint(uint256 vault, uint256[] nfts)
    public
    payable
    nonReentrant
    balanced(vault)
{
    // ... body here
}
```

Note that the code above does not apply as-is, nor would it apply to code that addresses issues from section 5.1, section 4.5, and so forth. Additionally, the code above won't work correctly without remediating the issue described in section 4.8, as stuck tokens will inflate the result of the call to `ERC721.balanceOf()`. The example code above is intended only to be illustrative of how to employ a modifier to preserve invariants.

Chapter 5

Cryptoeconomic Defects

Cryptoeconomic defects are those that present a risk of economic loss to the users or administrators during ordinary operation of the smart contracts covered by this review.

5.1 Redemption Arbitrage

The fact that an `XToken` can, in principle, be redeemed for any NFT that constitutes a part of the token's `Vault` means that there will be an economic incentive to redeem high-value NFTs in exchange for low-value NFTs that constitute members of the same `Vault`. When the total value of NFTs in a `Vault` falls, we expect to see a corresponding reduction in the market price of the `XToken`. There are a variety of ways in which an attacker can perform this arbitrage reliably and without capital risk; some of those scenarios are as follows:

- An attacker can deploy a large number of trivial proxy contracts and compute within a transaction which value of `msg.sender` leads to a desirable collision. Given that block times are approximately 15 seconds apart, and therefore there are fewer than 4 bits of entropy in `block.timestamp` when an attacker knows the timeframe in which a transaction is posted, an attacker needs only to evaluate sixteen candidate proxy addresses on average in order to select one that yields a desirable output for `_getPseudoRand()`.

- Even if `_getPseudoRand()` were actually non-deterministic, an attacker could use a flash loan to segregate high-value and low-value NFTs from a redemption. For example, if an attacker uses a flash loan of an `XToken` to redeem 100 NFTs, sells the highest-value 5 NFTs of that group, buys 5 low-value NFTs, and then returns those plus the remaining 95 to the `Vault`, the attacker can return a profit determined by the variance in market value of the NFTs that constitute a `Vault`. Anecdotally, many NFTs have seen a large intra-NFT variance in market-clearing price. Note that a flash loan is not necessary for this attack to work; it simply reduces the capital requirements and associated cost-of-carry to make this attack efficient.

- The `_getPseudoRand()` function uses inputs that are deterministically controlled by miners, so miners and anyone that can collude with them can ensure that they can withdraw precisely the NFT they intend to receive with a call to `redeem()`. An attacker can also facilitate collusion with a miner by creating a contract that rewards the transaction coinbase (the miner) of particular calls if the miner-controlled bits of determinism (like the timestamp) yield a large profit from trading. No informal or "off-chain" collusion is necessary.
- Attackers can use a flash loan to temporarily redeem every non-reserved NFT in a `Vault`, which means they can launch this attack against a (presumably) smaller list of higher-value NFTs.

The fact that a `Vault` is comprised of a collection of indivisible assets with different prices means that it is **not economically sound** to allow 1:1 redemption of fungible tokens for non-fungible assets. In order to remediate this issue, the `Vault` would need to recognize that the market price of its assets is heterogeneous. Otherwise, the proposition of a forseeable and deterministic economic loss may discourage owners of non-fungible tokens from participating in the `NFTX` ecosystem.

5.2 Bounty Balance Draining

In the event that minting, bounties, or burning fees are asymmetric, the Ether balance of the `Vault` contract can be drained risklessly. In other words, if minting tokens produces more in bounties than redeeming tokens costs in fees, then anyone can risklessly pocket the difference until the `Vault` balance is exhausted by minting and redeeming repeatedly within a single transaction. An attacker that borrows a significant portion of the `XToken` supply for the duration of a flash loan can carry out this attack instantaneously and with zero capital (beyond transaction fees).

There isn't a clear economic rationale for assessing fees upon the exchange of non-fungibles for fungible assets, or vice-versa. Anyone can independently encourage NFTs to be locked in collateral pools by bidding up the price of synthetic non-fungible-collateralized assets on an exchange, so `NFTX` does not need to invent a new mechanism to create an economic incentive to deposit NFTs as collateral. Moreover, eliminating any management of Ether in the NFTX contracts eliminates a substantial amount of attack surface. In summary, NFTX should use on-chain exchanges like Uniswap to send a price signal about NFTX tokens, rather than inventing another price signal mechanism.

5.3 Nonlinear Fees

The fees calculated by `_calcFee()` are not linear in the number of tokens exchanged. For example, one call to `mint()` that mints two tokens is not assessed the same fees as two calls to `mint()` that mint one token apiece. In the former case, `_calcFee(2, ethBase, ethStep, false)` will yield just `ethBase+ethStep`, while in the latter case, `2*_calcFee(1, ethBase, ethStep, false)` will yield `2*ethBase`. Unless `ethBase` is always equal to `ethStep`, callers will have an incentive to structure their calls such that they trigger whichever case leads to the largest bounty and/or the lowest fees. Since minting two tokens in one call versus two is economically equivalent, it makes sense to assess fees that are exactly the same in both circumstances.

Note that this issue may make the issue described in section 5.2 easier to exploit.

5.4 Vault Emptiness Check Bypass

The check on lines 429 through 437 of `NFTX.sol` check that a `Vault` is empty before `setNegateEligibility()` is called on it. However, the vault "manager" can use a flash loan to temporarily redeem all of the vaults assets to make it empty for a portion of a transaction, thereby bypassing the check. See also issue section 3.2.