Security Audit Report

NFTX V2 Peripheral Smart Contracts

SECBIT

December 23, 2022

1. Introduction

NFTX is a platform for creating liquid markets for illiquid Non-Fungible Tokens (NFTs). Users deposit their NFT into an NFTX vault and mint a fungible ERC20 token (vToken) representing a claim on a random or specific asset from the vault. SECBIT Labs conducted an audit from November 14 to December 8, 2022, including an analysis of the smart contracts in 3 areas: **code bugs**, **logic flaws**, and **risk assessment**. The assessment shows that the peripheral smart contracts of NFTX V2 we audited have no critical security risks. The SECBIT team has some tips on logical implementation, potential risks, and code revising (see part 4 for details).

Type	Description	Level	Status
Design & Implementation	4.3.1 Improper initialization used for upgradable contracts.	Medium	Fixed
Design & Implementation	4.3.2 The CRYPTOPUNKS token cannot be transferred to the vault due to a lack of authorization.	Medium	Fixed
Design & Implementation	4.3.3 Add liquidity feature will not work due to a lack of authorization.	Medium	Fixed
Code Revise	4.3.4 Meaningless `require` code should be removed to clarify the code logic.	Info	Fixed
Design & Implementation	4.3.5 Add a check for the `quoteAmount` parameter to ensure users have enough vault tokens to redeem NFT tokens.	Info	Fixed
Code Revise	4.3.6 Revoke allowances to improve security and compatibility.	Info	Discussed
Design & Implementation	4.3.7 Spender variables are misused or misdescribed in multiple places.	Info	Fixed
Design & Implementation	4.3.8 Some issues about the `buyAndStakeLiquidity()` function.	Medium	Fixed
Gas Optimization	4.3.9 It is recommended to adjust the code structure to reduce user costs.	Info	Fixed
Code Revise	4.3.10 Unused code.	Info	Fixed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about the NFTX V2 peripheral contracts is shown below:

- Project website
  - https://nftx.io/
- Smart contract code
  - initial review commit *97aa8e3*
  - final review commit *a5da756*

2.2 Contract List

The following content shows the contracts included in the NFTX V2 peripheral contracts, which the SECBIT team audits:

Name	Lines	Description
NFTXMarketplace0xZap.sol	259	This contract creates a marketplace zap to interact with the 0x protocol.
NFTXYieldStakingZap.sol	156	This contract allows users to buy and stake tokens into an inventory or liquidity pool, handling the steps between buying and staking across 0x and Sushi protocol.
NFTXENSMerkleEligibility.sol	38	The contract allows vaults to be allow eligibility based ENS domains, allowing for minimum expiration times to be set.
VaultCreationZap.sol	198	An combination of vault creation steps, merged and optimized in a single contract call to reduce gas costs to the end-user.

3. Contract Analysis

This part describes code assessment details, including "role classification" and "functional analysis".

3.1 Role Classification

There are two key roles in the NFTX V2 peripheral contracts: Governance Account and Common Account.

- Governance Account
    - Description

      Contract administrator
    - Authority
        - Transfer ownership
        - Allows owner to withdraw any tokens in the contract
        - Allows the zap to be paused to prevent any processing
    - Method of Authorization

      The contract administrator is the contract's creator or authorized by transferring the governance account.
- Common Account
    - Description

      Participate in the NFTX V2 peripheral contracts
    - Authority
        - Buy and stake tokens against an inventory
        - Mint tokens from NFTX vault and sell them on 0x protocol
        - Purchase vault tokens from 0x protocol with WETH and then swap the tokens
        - Redeem the NFT tokens
    - Method of Authorization

      No authorization required

3.2 Functional Analysis

The NFTX V2 peripheral contracts combine the different core functions of the protocol to make it more user-friendly. The SECBIT team conducted a detailed audit of some of the contracts in the protocol. We can divide the essential functions of the contract into three parts:

NFTXMarketplace0xZap

This contract sets up a marketplace zap to interact with the 0x protocol. The 0x protocol contract will be called later and handles the token swap based on swap parameters.

The main functions in `NFTXMarketplace0xZap` are as below:

- `mintAndSell721()`

  This function allows users to mint vault tokens using ERC721 tokens and sells them on 0x protocol.

- `buyAndSwap721()`

  This function allows users to purchase vault tokens from 0x protocol with WETH and then swap ERC721 tokens for either random or specific ERC721 token IDs from the NFTX vault.

- `buyAndRedeem()`

  Users can call on this function to purchase vault tokens from 0x protocol with WETH, and then redeem ERC721 tokens for either random or specific token IDs from the vault.

- `mintAndSell1155()`

  Users can mint tokens using ERC1155 tokens from the NFTX vault and sell them on 0x protocol.

- `buyAndSwap1155()`

  Users can purchase vault tokens from 0x protocol with WETH, and then swap the ERC1155 tokens for either random or specific token IDs from the vault.

NFTXYieldStakingZap

This contract allows users to buy and stake tokens into an inventory or liquidity pool, handling the steps between buying and staking across 0x protocol and Sushi protocol.

The main functions in `NFTXYieldStakingZap` are as below:

- `buyAndStakeInventory()`

  This function allows the user to buy and stake tokens against an inventory. It will handle the purchase of the vault tokens against 0x protocol and then generate the xToken against the vault and time-lock them.

- `buyAndStakeLiquidity()`

  This function allows users to buy and stake tokens against a liquidity pool. It will handle the purchase of the vault tokens against 0x protocol, the liquidity pool supplying via Sushi, and then the time-locking against the LP token.

VaultCreationZap

A combination of vault creation steps merged and optimized in a single contract call to reduce gas costs to the end-user. The main function in `VaultCreationZap` is as below:

- `createVault()`

  Creates an NFTX vault, handling any desired settings and tokens.

4. Audit Detail

This part describes the process, and the detailed audit results also demonstrate the problems and potential risks.

4.1 Audit Process

The audit strictly followed the audit specification of SECBIT Lab. We analyzed the project from code bugs, logical implementation, and potential risks. The process consists of four steps:

- Fully analysis of contract code line by line.
- Evaluation of vulnerabilities and potential risks revealed in the contract code.
- Communication on assessment and confirmation.
- Audit report writing.

4.2 Audit Result

After scanning with adelaide, sf-checker, and badmsg.sender (internal version) developed by SECBIT Labs and open source tools, including Mythril, Slither, SmartCheck, and Securify, the auditing team performed a manual assessment. The team inspected the contract line by line, and the result could be categorized into the following types:

Number	Classification	Result
1	Normal functioning of features defined by the contract	✓
2	No obvious bug (e.g., overflow, underflow)	✓
3	Pass Solidity compiler check with no potential error	✓
4	Pass common tools check with no obvious vulnerability	✓
5	No obvious gas-consuming operation	✓
6	Meet with ERC20 standard	✓

7	No risk in low-level call (call, delegatecall, callcode) and in-line assembly	✓
8	No deprecated or outdated usage	✓
9	Explicit implementation, visibility, variable type, and Solidity version number	✓
10	No redundant code	✓
11	No potential risk manipulated by timestamp and network environment	✓
12	Explicit business logic	✓
13	Implementation consistent with annotation and other info	✓
14	No hidden code about any logic that is not mentioned in design	✓
15	No ambiguous logic	✓
16	No risk threatening the developing team	✓
17	No risk threatening exchanges, wallets, and DApps	✓
18	No risk threatening token holders	✓
19	No privilege on managing others' balances	✓
20	No non-essential minting method	✓
21	Correct managing hierarchy	✓

4.3 Issues

4.3.1 Improper initialization used for upgradable contracts.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Medium	Design logic	Fixed

Description

Due to a requirement of the proxy-based upgradeability system, no constructors can be used in upgradeable contracts. The code within an implementation contract's constructor will never be executed in the context of the proxy's state.

When writing an upgradable contract, you need to change its constructor into a regular function, typically named `initialize`, where you run all the setup logic. To prevent a contract from being *initialized* multiple times, you must add a check to ensure the `initialize` function is called only once. For more details, see the link: https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable.

The same issue exists in the `NFTXMarketplace0xZap.sol` and `NFTXYieldStakingZap.sol` contracts. If deployed as a proxy-based upgradable contract, their initialization would all be invalid, which in turn would cause other malfunctions.

```
constructor(
    address _vaultFactory,
    address _inventoryStaking,
    address _lpStaking,
    address _sushiRouter,
    address _sushiHelper,
    address _weth
  ) Ownable() ReentrancyGuard() {
    // Set our staking contracts
    inventoryStaking = INFTXInventoryStaking(_inventoryStaking);
    lpStaking = INFTXLPStaking(_lpStaking);

    // Set our NFTX factory contract
    vaultFactory = INFTXVaultFactory(_vaultFactory);

    // Set our Sushi Router used for liquidity
    sushiRouter = IUniswapV2Router01(_sushiRouter);
    sushiHelper = SushiHelper(_sushiHelper);

    // Set our chain's WETH contract
    WETH = IWETH(_weth);
  }
```

Status

The team has clarified that the upgradeable contract pattern is not used and has modified the corresponding codes in commit d2acdde.

4.3.2 The CRYPTOPUNKS token cannot be transferred to the vault due to a lack of authorization.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Medium	Design logic	Fixed

Description

The CRYPTOPUNKS token is a different NFT token that does not follow the ERC721 standard, and the zap contract has an exceptive handling strategy. The `_transferFromERC721()` function shows that the zap contract calls the `buyPunk()` function on the CRYPTOPUNKS token contract via the call instruction, which transfers the token from the user to the zap contract. Note that the state of the CRYPTOPUNKS token is reset by the `punkNoLongerForSale()` function when the `buyPunk()` function is called.

Next, the zap contract will call the `mintTo()` function, trying to transfer the CRYPTOPUNKS token to the vault contract, but the strategy will fail due to a lack of authorization. The `mintTo()` function indicates that it still calls the `buyPunk()` function to transfer the CRYPTOPUNKS token, but there is no authorization before that.

```
function createVault(
    vaultInfo calldata vaultData,
    uint vaultFeatures,
    vaultFeesConfig calldata vaultFees,
    vaultEligibilityStorage calldata eligibilityStorage,
    vaultTokens calldata assetTokens
  ) external nonReentrant payable returns (uint vaultId_) {
   ......

    // If we don't have any tokens to send, we can skip our transfers
    if (length > 0) {
      // Determine the token type to alternate our transfer logic
      if (!vaultData.is1155) {
        // Iterate over our 721 tokens to transfer them all to our vault
        for (uint i; i < length;) {
```

```
          _transferFromERC721(vaultData.assetAddress,
assetTokens.assetTokenIds[i], address(vault));
          unchecked { ++i; }
        }
      } else {
        ......
      }

      // We can now mint our asset tokens, giving the vault our tokens
and storing them
      // inside our zap, as we will shortly be staking them. Our zap is
excluded from fees,
      // so there should be no loss in the amount returned.
      vault.mintTo(assetTokens.assetTokenIds,
assetTokens.assetTokenAmounts, address(this));
      ......
  }


function _transferFromERC721(address assetAddr, uint256 tokenId, address
to) internal virtual {
    bytes memory data;

    if (assetAddr == 0xb47e3cd837dDF8e4c57F05d70Ab865de6e193BBB) {
      // Fix here for frontrun attack.
      bytes memory punkIndexToAddress =
abi.encodeWithSignature("punkIndexToAddress(uint256)", tokenId);
      (bool checkSuccess, bytes memory result) =
address(assetAddr).staticcall(punkIndexToAddress);
      (address nftOwner) = abi.decode(result, (address));
      require(checkSuccess && nftOwner == msg.sender, "Not the NFT
owner");
      data = abi.encodeWithSignature("buyPunk(uint256)", tokenId);
    } else {
      // We push to the vault to avoid an unneeded transfer.
      data =
abi.encodeWithSignature("safeTransferFrom(address,address,uint256)",
msg.sender, to, tokenId);
    }

    (bool success, bytes memory resultData) =
address(assetAddr).call(data);
    require(success, string(resultData));
}
```

```
//@notice Copy from https://github.com/NFTX-project/nftx-protocol-v2/blob/97aa8e30027d554592149f0f65aae06fbe86faf4/contracts/solidity/NFTXVaultUpgradeable.sol

function mintTo(
        uint256[] memory tokenIds,
        uint256[] memory amounts, /* ignored for ERC721 vaults */
        address to
    ) public override virtual nonReentrant returns (uint256) {
        onlyOwnerIfPaused(1);
        require(enableMint, "Minting not enabled");

        // Take the NFTs.
        uint256 count = receiveNFTs(tokenIds, amounts);
       ......
    }

function receiveNFTs(uint256[] memory tokenIds, uint256[] memory amounts)
        internal
        virtual
        returns (uint256)
    {
        require(allValidNFTs(tokenIds), "NFTXVault: not eligible");
        uint256 length = tokenIds.length;
        if (is1155) {
            ......
        } else {
            address _assetAddress = assetAddress;
            for (uint256 i; i < length; ++i) {
                uint256 tokenId = tokenIds[i];
                // We may already own the NFT here so we check in order:
                // Does the vault own it?
                //   - If so, check if its in holdings list
                //      - If so, we reject. This means the NFT has already been claimed for.
                //      - If not, it means we have not yet accounted for this NFT, so we continue.
                //   -If not, we "pull" it from the msg.sender and add to holdings.
                transferFromERC721(_assetAddress, tokenId);
                holdings.add(tokenId);
            }
            return length;
        }
    }
```

```
function transferFromERC721(address assetAddr, uint256 tokenId) internal
virtual {
        address kitties = 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d;
        address punks = 0xb47e3cd837dDF8e4c57F05d70Ab865de6e193BBB;
        bytes memory data;
        if (assetAddr == kitties) {
          ......
        } else if (assetAddr == punks) {
            // CryptoPunks.
            // Fix here for frontrun attack. Added in v1.0.2.
            bytes memory punkIndexToAddress =
abi.encodeWithSignature("punkIndexToAddress(uint256)", tokenId);
            (bool checkSuccess, bytes memory result) =
address(assetAddr).staticcall(punkIndexToAddress);
            (address nftOwner) = abi.decode(result, (address));
            require(checkSuccess && nftOwner == msg.sender, "Not the NFT
owner");
            data = abi.encodeWithSignature("buyPunk(uint256)", tokenId);
        } else {
            ......
        }
        (bool success, bytes memory resultData) =
address(assetAddr).call(data);
        require(success, string(resultData));
    }
```

Suggestion

One recommended modification is to call the `offerPunkForSaleToAddress()` function of the CRYPTOPUNKS token contract after transferring the CRYPTOPUNKS token from the user to the zap contract to authorize the vault contract.

```
function createVault(
    vaultInfo calldata vaultData,
    uint vaultFeatures,
    vaultFeesConfig calldata vaultFees,
    vaultEligibilityStorage calldata eligibilityStorage,
    vaultTokens calldata assetTokens
  ) external nonReentrant payable returns (uint vaultId_) {
   ......
    // If we don't have any tokens to send, we can skip our transfers
    if (length > 0) {
      // Determine the token type to alternate our transfer logic
```

```
      if (!vaultData.is1155) {
        // Iterate over our 721 tokens to transfer them all to our vault
        for (uint i; i < length;) {
          _transferFromERC721(vaultData.assetAddress,
assetTokens.assetTokenIds[i], address(vault));

          // @audit add the following codes for punks
          if(vaultData.assetAddress == CRYPTO_PUNKS) {
          bytes memory data = abi.encodeWithSignature(
              "offerPunkForSaleToAddress (uint256,uint256,address)",
      assetTokens.assetTokenIds[i], 0, address(vault));
          (bool success, bytes memory resultData) =
address(vaultData.assetAddress).call(data);
           require(success, string(resultData));
          }
          unchecked { ++i; }
        }
      } else {
        ......
      }

      // We can now mint our asset tokens, giving the vault our tokens
and storing them
      // inside our zap, as we will shortly be staking them. Our zap is
excluded from fees,
      // so there should be no loss in the amount returned.
      vault.mintTo(assetTokens.assetTokenIds,
assetTokens.assetTokenAmounts, address(this));
      ......
  }
```

Status

The team has adopted this suggestion and fixed this issue in commit 93ad784.

4.3.3 Add liquidity feature will not work due to a lack of authorization.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Medium	Design logic	Fixed

Description

After depositing NFT tokens into the vault, the user can mint asset tokens (named baseToken), which are sent to the zap contract. The user then uses the `baseToken` and `WETH` tokens to add liquidity to the Sushi protocol. At this point, we notice that only the `baseToken` under the zap contract is authorized/approved for the Sushi protocol, while the `WETH` token is not. In this case, the Sushi protocol cannot transfer the `WETH` tokens under the zap contract, so adding liquidity will fail.

```
function createVault(
    vaultInfo calldata vaultData,
    uint vaultFeatures,
    vaultFeesConfig calldata vaultFees,
    vaultEligibilityStorage calldata eligibilityStorage,
    vaultTokens calldata assetTokens
  ) external nonReentrant payable returns (uint vaultId_) {
    ......
        // Provide liquidity to sushiswap, using the vault tokens and
pairing it with the
        // liquidity amount specified in the call.
        IERC20Upgradeable(baseToken).safeApprove(address(sushiRouter),
assetTokens.minTokenIn);
        (,, uint256 liquidity) = sushiRouter.addLiquidity(
          baseToken,
          address(WETH),
          assetTokens.minTokenIn,
          assetTokens.wethIn,
          assetTokens.minTokenIn,
          assetTokens.minWethIn,
          address(this),
          block.timestamp
        );
       ......
  }
```

Suggestion

The zap contract should add authorization for WETH tokens. The recommended modification is as follows.

```
interface IWETH {
  function deposit() external payable;
  function transfer(address to, uint value) external returns (bool);
  function withdraw(uint) external;
```

```
  function balanceOf(address to) external view returns (uint256);

  // @audit add the following code
  function approve(address guy, uint wad) public returns (bool);
}

function createVault(
    vaultInfo calldata vaultData,
    uint vaultFeatures,
    vaultFeesConfig calldata vaultFees,
    vaultEligibilityStorage calldata eligibilityStorage,
    vaultTokens calldata assetTokens
  ) external nonReentrant payable returns (uint vaultId_) {
    ......
        // Provide liquidity to sushiswap, using the vault tokens and
pairing it with the
        // liquidity amount specified in the call.
        IERC20Upgradeable(baseToken).safeApprove(address(sushiRouter),
assetTokens.minTokenIn);

        // @audit add the following code
        WETH.approve(address(sushiRouter),assetTokens.wethIn);

        (,, uint256 liquidity) = sushiRouter.addLiquidity(
          baseToken,
          address(WETH),
          assetTokens.minTokenIn,
          assetTokens.wethIn,
          assetTokens.minTokenIn,
          assetTokens.minWethIn,
          address(this),
          block.timestamp
        );
        //@audit To prevent potential risks,
        //       it is recommended to revoke the allowance.
        IERC20Upgradeable(baseToken).safeApprove(address(sushiRouter),
0);
        WETH.approve(address(sushiRouter),0);
       ......
  }
```

Status

The team has adopted this suggestion and fixed this issue in commit 0e174bb.

4.3.4 Meaningless `require` code should be removed to clarify the code logic.

Risk Type	Risk Level	Impact	Status
Code Revise	Info	Design logic	Fixed

Description

The `buyAndRedeem()` function can be classified into two parts based on the content. One is to purchase vault tokens from 0x protocol with WETH tokens, and the other is to redeem the NFT tokens for either random or specific NFT token IDs from the vault with vault tokens. The number of ETH transferred in by the user is `msg.value`, while the parameter `amount` represents the number of NFT tokens the user wants to redeem. The two do not have the same meaning, and there is no point in comparing them. In extreme cases, it may even be problematic. Therefore, the `require` condition between them should be removed.

```
function buyAndRedeem(
    uint256 vaultId,
    uint256 amount,
    uint256[] calldata specificIds,
    address spender,
    bytes calldata swapCallData,
    address payable to
  ) external payable nonReentrant onlyOwnerIfPaused {

    require(to != address(0) && to != address(this), 'Invalid
recipient');

    require(amount > 0, 'Must send amount');

    //@audit the 'amount' represent the number of NFT tokens
    //@audit this code should be removed
    require(msg.value >= amount, 'Invalid amount');

    WETH.deposit{value: msg.value}();
    ......
  }
```

Status

The team has removed the related code in commit 44cc2fa.

4.3.5 Add a check for the `quoteAmount` parameter to ensure users have enough vault tokens to redeem NFT tokens.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Design logic	Fixed

Description

The `quoteAmount` parameter is the number of vault tokens purchased by the user. When the user redeems NFT tokens, the corresponding vault tokens will be burned. The burned value is straightforward to calculate. A check for the `quoteAmount` parameter is recommended to ensure that the user has enough vault tokens to redeem NFT tokens.

```
function buyAndRedeem(
    uint256 vaultId,
    uint256 amount,
    uint256[] calldata specificIds,
    address spender,
    bytes calldata swapCallData,
    address payable to
  ) external payable nonReentrant onlyOwnerIfPaused {
    ......
    // Get our vault address information
    address vault = _vaultAddress(vaultId);

    // Buy vault tokens that will cover our transaction
    uint256 quoteAmount = _fillQuote(address(WETH), vault, swapCallData);

    // Redeem token IDs from the vault
    _redeem(vaultId, amount, specificIds, to);
    emit Buy(amount, quoteAmount, to);

    // Transfer dust back to the spender
    _transferDust(spender, vault);
  }
```

Suggestion

A recommended modification is as follows.

```
function buyAndRedeem(
    uint256 vaultId,
    uint256 amount,
    uint256[] calldata specificIds,
    address spender,
    bytes calldata swapCallData,
    address payable to
  ) external payable nonReentrant onlyOwnerIfPaused {
    ......
    // Get our vault address information
    address vault = _vaultAddress(vaultId);

    // Buy vault tokens that will cover our transaction
    uint256 quoteAmount = _fillQuote(address(WETH), vault, swapCallData);

    // @audit add the following code
    require(quoteAmount >= amount * 1e18, 'Insufficient vault tokens');

    // Redeem token IDs from the vault
    _redeem(vaultId, amount, specificIds, to);
    emit Buy(amount, quoteAmount, to);

    // Transfer dust back to the spender
    _transferDust(spender, vault);
  }
```

Status

The team has adopted the suggestion and fixed this issue in commit f814d35.

4.3.6 Revoke allowances to improve security and compatibility.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Design logic	Discussed

Description

Unlimited allowances to external contracts can pose a high-security risk to funds. For security concerns, it is advisable to revoke allowances for external contracts as soon as after use. Besides, some tokens (e.g., USDT, KNC) will need allowance resets to zero before another update.

```
function _fillQuote(
    address sellToken,
    address buyToken,
    bytes calldata swapCallData
  ) internal returns (uint256) {
    // Track our balance of the buyToken to determine how much we've
bought.
    uint256 boughtAmount = IERC20(buyToken).balanceOf(address(this));

    // Give `swapTarget` an infinite allowance to spend this contract's
`sellToken`.
    // Note that for some tokens (e.g., USDT, KNC), you must first reset
any existing
    // allowance to 0 before being able to update it.
    require(IERC20(sellToken).approve(swapTarget, type(uint256).max),
'Unable to approve contract');

    // Call the encoded swap function call on the contract at
`swapTarget`
    (bool success,) = swapTarget.call(swapCallData);
    require(success, 'SWAP_CALL_FAILED');

    // Use our current buyToken balance to determine how much we've
bought.
    return IERC20(buyToken).balanceOf(address(this)) - boughtAmount;
  }
```

Suggestion

A recommended modification is as follows.

```
function _fillQuote(
    address sellToken,
    address buyToken,
    bytes calldata swapCallData
  ) internal returns (uint256) {
```

```
    // Track our balance of the buyToken to determine how much we've
bought.
    uint256 boughtAmount = IERC20(buyToken).balanceOf(address(this));

    // Give `swapTarget` an infinite allowance to spend this contract's
`sellToken`.
    // Note that for some tokens (e.g., USDT, KNC), you must first reset
any existing
    // allowance to 0 before being able to update it.
    require(IERC20(sellToken).approve(swapTarget, type(uint256).max),
'Unable to approve contract');

    // Call the encoded swap function call on the contract at
`swapTarget`
    (bool success,) = swapTarget.call(swapCallData);
    require(success, 'SWAP_CALL_FAILED');

     // @audit add the following code
     IERC20(sellToken).approve(swapTarget, 0);

    // Use our current buyToken balance to determine how much we've
bought.
    return IERC20(buyToken).balanceOf(address(this)) - boughtAmount;
  }
```

Status

This issue has been discussed. Currently, external contracts are trusted, so no particular actions are needed.

4.3.7 Spender variables are misused or misdescribed in multiple places.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Design logic	Fixed

Description

According to the code comment, the `spender` variable is the address defined in the 0x protocol interface for processing user token allowance. The address returned by the actual 0x protocol in production is the same as the swapTarget. However, in NFTXMarketplace0xZap, the `_transferDust` function does not make sense to transfer the

change to this `spender` address. The funds should be transferred to `msg.sender` or an address specified by the user.

```
/**
   * @notice Mints tokens from our NFTX vault and sells them on 0x.
   *
   * @param vaultId The ID of the NFTX vault
   * @param ids An array of token IDs to be minted
   * @param spender The `allowanceTarget` field from the API response
   * @param swapCallData The `data` field from the API response
   * @param to The recipient of the WETH from the tx
   */
  function mintAndSell721(
    uint256 vaultId,
    uint256[] calldata ids,
    address spender,
    bytes calldata swapCallData,
    address payable to
  ) external nonReentrant onlyOwnerIfPaused {
    ...
    // Transfer dust back to the spender
    _transferDust(spender, vault);
  }
```

Related functions: `mintAndSell721` `buyAndSwap721` `buyAndRedeem` `mintAndSell1155` `buyAndSwap1155`.

Suggestion

Confirm the implementation here and change the code comments.

Status

The team has adopted our suggestion and updated the recipient of the dust in the `_transferDust()` function from `spender` to `msg.sender` in commit 85b0aa2.

4.3.8 Some issues about the `buyAndStakeLiquidity()` function.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Medium	Design logic	Fixed

Description

The `buyAndStakeLiquidity()` function allows users to buy and stake tokens against a liquidity pool.

We have some opinions and suggestions about this function.

(1). It is recommended to determine if there are sufficient WETH tokens in the contract before adding liquidity to Sushi protocol.

(2). Adding liquidity will always fail because of not authorizing `WETH` tokens to Sushi protocol.

(3). Adjusting the `addLiquidity()` function parameters to prevent adding liquidity from failing.

When the parameter `amountADesired` is equal to the parameter `amountAMin`, adding liquidity will likely fail. Based on the current code, it seems that the `amountADesired` parameter of the addLiquidity function should be `vaultTokenAmount` rather than `minTokenIn`.

```
function buyAndStakeLiquidity(
    // Base data
    uint256 vaultId,

    // 0x integration
    bytes calldata swapCallData,

    // Sushiswap integration
    uint256 minTokenIn,
    uint256 minWethIn,
    uint256 wethIn

  ) external payable nonReentrant {
    ......
    // Convert WETH to vault token
    uint256 vaultTokenAmount = _fillQuote(baseToken, swapCallData);
    require(vaultTokenAmount > minTokenIn, 'Insufficient tokens
acquired');

    // Provide liquidity to sushiswap, using the vault token that we
acquired from 0x and
    // pairing it with the liquidity amount specified in the call.
    IERC20Upgradeable(baseToken).safeApprove(address(sushiRouter),
minTokenIn);
```

```
    (uint256 amountToken, , uint256 liquidity) =
sushiRouter.addLiquidity(
      baseToken,
      address(WETH),
      minTokenIn,
      wethIn,
      minTokenIn,
      minWethIn,
      address(this),
      block.timestamp
    );

    // Stake in LP rewards contract
    address lpToken = pairFor(baseToken, address(WETH));
    IERC20Upgradeable(lpToken).safeApprove(address(lpStaking),
liquidity);
    lpStaking.timelockDepositFor(vaultId, msg.sender, liquidity, 48
hours);
    ......
  }
```

Suggestion

One possible modification for reference is as follows.

```
interface IWETH {
  function deposit() external payable;
  function transfer(address to, uint value) external returns (bool);
  function withdraw(uint) external;
  function balanceOf(address to) external view returns (uint256);

  //@audit add the following code
  function approve(address guy, uint wad) public returns (bool);
}

function buyAndStakeLiquidity(
    // Base data
    uint256 vaultId,

    // 0x integration
    bytes calldata swapCallData,

    // Sushiswap integration
    uint256 minTokenIn,
    uint256 minWethIn,
```

```
    uint256 wethIn

  ) external payable nonReentrant {
     ......
    // Convert WETH to vault token
    uint256 vaultTokenAmount = _fillQuote(baseToken, swapCallData);
    require(vaultTokenAmount > minTokenIn, 'Insufficient tokens
acquired');

    // @audit Check WETH balance
    uint256 WETHAmount = WETH.balanceOf(address(this)) - wethBalance;
    require(WETHAmount >= wethIn, 'Insufficient WETH acquired');

    // Provide liquidity to sushiswap, using the vault token that we
acquired from 0x and
    // pairing it with the liquidity amount specified in the call.

    // @audit Adjust allowance from `minTokenIn` to `vaultTokenAmount`
    IERC20Upgradeable(baseToken).safeApprove(address(sushiRouter),
vaultTokenAmount);

    // @audit Approve WETH to sushiRouter
    WETH.approve(address(sushiRouter), wethIn);

    (uint256 amountToken, , uint256 liquidity) =
sushiRouter.addLiquidity(
      baseToken,
      address(WETH),
      vaultTokenAmount,  // @audit Adjust parameter
                         // from `minTokenIn` to `vaultTokenAmount`
      wethIn,
      minTokenIn,
      minWethIn,
      address(this),
      block.timestamp
    );


    // @audit Revoke allowance
    IERC20Upgradeable(baseToken).safeApprove(address(sushiRouter), 0);
    WETH.approve(address(sushiRouter), 0);

    // Stake in LP rewards contract
    address lpToken = pairFor(baseToken, address(WETH));
```

```
    IERC20Upgradeable(lpToken).safeApprove(address(lpStaking),
liquidity);
    lpStaking.timelockDepositFor(vaultId, msg.sender, liquidity, 48
hours);
    ......
  }
```

Status

The team has adopted the suggestion and fixed this issue in commits 11c11ed and 8209711.

4.3.9 It is recommended to adjust the code structure to reduce user costs.

Risk Type	Risk Level	Impact	Status
Gas Optimization	Info	More gas consumption	Fixed

Description

Notice that the user calls the `_fillQuote()` function to authorize the WETH tokens to the `swapTarget` address. Since the allowance can be considered an infinite value(2**256), this `approve` action only needs to be called once. We recommend moving this `approve` action to another function, such as an initialize function, which will reduce the gas cost when calling the `_fillQuote()` function each time.

```
function _fillQuote(
    address buyToken,
    bytes calldata swapCallData
  ) internal returns (uint256) {
      // Track our balance of the buyToken to determine how much we've
bought.
      uint256 boughtAmount =
IERC20Upgradeable(buyToken).balanceOf(address(this));

      // Give `swapTarget` an infinite allowance to spend this contract's
`sellToken`.
      // Note that for some tokens (e.g., USDT, KNC), you must first
reset any existing
      // allowance to 0 before being able to update it.
      require(IERC20Upgradeable(address(WETH)).approve(swapTarget,
type(uint256).max), 'Unable to approve contract');
```

```
        // Call the encoded swap function call on the contract at
`swapTarget`
        (bool success,) = swapTarget.call(swapCallData);
        require(success, 'SWAP_CALL_FAILED');

        // Use our current buyToken balance to determine how much we've
bought.
        return IERC20Upgradeable(buyToken).balanceOf(address(this)) -
boughtAmount;
    }
```

Status

The team has adopted our suggestion to adjust the authorization of the WETH token from the `_fillQuote()` function to the `constructor()` function in commit dde43e6.

4.3.10 Unused code

Risk Type	Risk Level	Impact	Status
Code Revise	Info	Design logic	Fixed

Description

The code below is never used. If it's not necessary, it is recommended to remove this code to reduce gas costs during contract deployment.

```
bool public paused = false;

function pause(bool _paused) external onlyOwner {
    paused = _paused;
}
```

Status

The team has added the related feature in commit 731a6bc.

5. Conclusion

After auditing and analyzing the NFTX V2 peripheral smart contracts, SECBIT Labs found some issues to optimize and proposed corresponding suggestions, which have been shown above.

Disclaimer

SECBIT smart contract audit service assesses the contract's correctness, security, and performability in code quality, logic design, and potential risks. The report is provided "as is", without any warranties about the code practicability, business model, management system's applicability, and anything related to the contract adaptation. This audit report is not to be taken as an endorsement of the platform, team, company, or investment.

APPENDIX

Vulnerability/Risk Level Classification

Level	Description
High	Severely damage the contract's integrity and allow attackers to steal ethers and tokens, or lock assets inside the contract.
Medium	Damage contract's security under given conditions and cause impairment of benefit for stakeholders.
Low	Cause no actual impairment to contract.
Info	Relevant to practice or rationality of the smart contract, could possibly bring risks.

SECBIT Lab is devoted to constructing a common-consensus, reliable, and ordered blockchain economic entity.

https://secbit.io

audit@secbit.io

@secbit_io