SFT Security Review

Pashov Audit Group

Conducted by: ether_sky, juancito, ubermensch, Peakbolt, 0xbepresent

May 13th 2024 - May 19th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **CypherLabz/SFT418V2** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About SFT418

SFT is a semi-fungible token design with a wrapper standard that combines ERC20 and ERC721 into a contract-pair that operates atomically and allows creation of a new semi-fungible tokens, as well as wrapping existing ERC721s (NFTs) into SFTs, unlocking DeFi liquidity for NFTs.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

the first review commit hash - 805714c57a7370baff9cf57b8662abad531846c6

the second review commit hash - 7be0d9b0cb08e4c78414e4b8fc3bfbfdeaf0e155

fixes review commit hash - 07329b7c13ae80c833b657aee31544da4fb75ff7

Scope

The following smart contracts were in scope of the audit:

- `SFT418`
- `SFT418Pair`
- `SFT418Wrapper`
- `SFT418WrapperPair`
- `ChunkProcessable`
- `ChunkProcessor`
- `ChunkProcessorWithSelector`
- `Interfaces`
- `SFT20`
- `SFT418Core`
- `SFT418CoreSequential`
- `SFT418PairCore`
- `SFT721`
- `SFTInterfaceHandler`
- `SFTMetadata`

7. Executive Summary

Over the course of the security review, ether_sky, juancito, ubermensch, Peakbolt, 0xbepresent engaged with SFT to review SFT418. In this period of time a total of **6** issues were uncovered.

Protocol Summary

Protocol Name	SFT418
Repository	https://github.com/CypherLabz/SFT418V2
Date	May 13th 2024 - May 19th 2024
Protocol Type	Token Standard

Findings Count

Severity	**Amount**
Critical	2
Medium	2
Low	2
Total Findings	**6**

Summary of Findings

ID	Title	Severity	Status
[C-01]	NFT stolen due to wrong approval validation (Project self-injected exploit)	Critical	Resolved
[C-02]	An adversary can claim any NFT from any user (Project self-injected exploit)	Critical	Resolved
[M-01]	Duplicate Chunk IDs	Medium	Resolved
[M-02]	Reroll becomes unfunctional after all tokens have been minted	Medium	Resolved
[L-01]	Incorrect Check in _reorderChunk	Low	Resolved
[L-02]	Missing token id existence check	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] NFT stolen due to wrong approval validation (Project self-injected exploit)

Severity

Impact: High

Likelihood: High

Description

When calling `approve()`, some checks are performed under the hood via `_FB721Approve()` to validate approval permissions.

Note that the check regarding `_isApprovedForAll` is flawed. It can be bypassed for any `id` if the caller approved a spender previously.

```
function _FB721Approve
      (address spender_, uint256 id_, address msgSender_) internal virtual {
        address _owner = _chunkToOwners[id_].owner;

        require(
            _owner == msgSender_ || // the owner of the token is the msg.sender
@>          _isApprovedForAll[msgSender_][spender_], // the msg.sender is an
// approved operator
            "NOT_AUTHORIZED"
        );

        _getApproved[id_] = spender_;
        require(NFT.emitApproval(_owner, spender_, id_));
    }
```

Proof of Concept

This test shows how anyone can steal any token from any user.

- Add the test to `test/SFT418V2_ERC721.t.sol`
- Run `forge test --mt "testApproveVulnerability"`

```
function testApproveVulnerability() public {
    address victim = address(0xABCD);
    address attacker = address(666);

    pToken.mint(victim, 1337);
    assertEq(pToken.ownerOf(1337), victim);

    hevm.startPrank(attacker);
    pToken.setApprovalForAll(attacker, true);
    pToken.approve(attacker, 1337); // @audit vulnerable function
    pToken.transferFrom(victim, attacker, 1337);
    hevm.stopPrank();

    assertEq(pToken.ownerOf(1337), attacker);
}
```

Recommendations

Make the following changes to the `_isApprovedForAll` check. Note how it now checks that the `owner` approved the caller as an operator.

```
function _FB721Approve
      (address spender_, uint256 id_, address msgSender_) internal virtual {
        address _owner = _chunkToOwners[id_].owner;

        require(
            _owner == msgSender_ || // the owner of the token is the msg.sender
-
-            _isApprovedForAll[msgSender_][spender_], // the msg.sender is an approved
+
+            _isApprovedForAll[_owner][msgSender_], // the msg.sender is an approved o
            "NOT_AUTHORIZED"
        );

        _getApproved[id_] = spender_;
        require(NFT.emitApproval(_owner, spender_, id_));
    }
```

[C-02] An adversary can claim any NFT from any user (Project self-injected exploit)

Severity

Impact: High

Likelihood: High

Description

The `_claim()` function in `SFT418Core` allows users to claim any NFT, despite the intention to only allow it from the token pool:

```
/**
@>
     * @notice claims an existing chunk in the token pool if the target has enough eli
     * @param target_ - the claimer
     * @param id_ - the ID to claim
     */
    function _claim(address target_, uint256 id_) internal virtual {
        uint256 _eligibleChunks = _getEligibleChunks(target_);
        require(_eligibleChunks > 0, "NOT_ENOUGH_ELIGIBLE");

        if (_chunkToOwners[id_].owner == TOKEN_POOL) {
            _ERC721Transfer(TOKEN_POOL, target_, id_);
        }

        else {
@>          _ERC721Transfer(_chunkToOwners[id_].owner, target_, id_);
        }
    }
```

So, a user can wrap an NFT, and without any further interaction from them, an adversary can "claim" it, unwrap it, and steal it from the user.

The victim might call `_claim()` after that, getting another NFT, but not the same.

Note how another similar function like `_swap()` prevents this attack with a `require(TOKEN_POOL == _chunkToOwners[targetId_].owner)` check.

Proof of Concept

- Add a claiming function to `test/demo/SFT418WPairDemoV2.sol`:

```
function claim(uint256 id_) public virtual {
    _claim(id_);
}
```

- Add the test to `test/SFT418WV2.t.sol`
- Run `forge test --mt "testClaimVulnerability"`

```
function testClaimVulnerability() public {
    // Set-up

    address victim = address(0x71C714);
    address attacker1 = address(0x666);
    address attacker2 = address(0x667);

    iToken.mint(victim, 1337);
    iToken.mint(attacker1, 666);
    iToken.mint(attacker2, 667);

    hevm.startPrank(victim);
    iToken.approve(address(token), 1337);
    pToken.wrap(1337);
    hevm.stopPrank();

    hevm.startPrank(attacker1);
    iToken.approve(address(token), 666);
    pToken.wrap(666);
    hevm.stopPrank();

    hevm.startPrank(attacker2);
    iToken.approve(address(token), 667);
    pToken.wrap(667);
    hevm.stopPrank();

    // Attack

    hevm.prank(attacker1);
    token.transfer(attacker2, 1);

    hevm.prank(attacker2);
    token.transfer(attacker1, 1);

    assertEq(pToken.ownerOf(1337), victim);

    hevm.prank(attacker1);
    pToken.claim(1337);

    assertEq(pToken.ownerOf(1337), attacker1);

    hevm.prank(attacker1);
    pToken.unwrap(1337);

    assertEq(iToken.ownerOf(1337), attacker1);
}
```

Recommendations

Only allow claiming from the token pool:

```
function _claim(address target_, uint256 id_) internal virtual {
        uint256 _eligibleChunks = _getEligibleChunks(target_);
        require(_eligibleChunks > 0, "NOT_ENOUGH_ELIGIBLE");

-       if (_chunkToOwners[id_].owner == TOKEN_POOL) {
-           _ERC721Transfer(TOKEN_POOL, target_, id_);
-       }
-
-       else {
-           _ERC721Transfer(_chunkToOwners[id_].owner, target_, id_);
-       }

+       require(TOKEN_POOL == _chunkToOwners[id_].owner, "INVALID_ID");
+       _ERC721Transfer(TOKEN_POOL, target_, id_);
    }
```

8.2. Medium Findings

[M-01] Duplicate Chunk IDs

Severity

Impact: High

Likelihood: Low

Description

In `ChunkProcessor`, the chunk manipulation functions such as `pushChunk()` will perform a downcast of Chunk ID and Indexes from `uint256` to `uint32`. This effectively reduces the maximum number of chunks to `type(uint32).max`, As chunks represent the paired/wrapped ERC721, this means that NFT supply based on SFT418 will be similarly capped as well.

The issue is that capping of the NFT supply to 32-bit is not compliant with EIP721, which allows up to `type(uint256).max` NFTs. This is evident from the use of `uint256` to represent the account balance in `balanceOf()`.

The implication of this issue is that SFT418 will not function properly when it is used for cases where the number of NFTs exceeds `type(uint32).max`.

That will cause duplication of chunk ID as Chunk ID greater than `type(uint32).max` will be truncated to 32-bit, violating the non-fungible property.

```
function _pushChunk(address to_, uint256 id_) internal virtual {
        require(to_ != address(0), "INVALID_TO");

        require(_chunkToOwners[id_].owner == address(0), "CHUNK_EXISTS");

        uint256 _nextIndex = _ownerToChunkIndexes[to_].length;

        _chunkToOwners[id_] = ChunkInfo(
            to_,
            //@audit index is downcasted to uint32
            uint32(_nextIndex)
        );

         //@audit chunk ID is also downcasted to uint32
        _ownerToChunkIndexes[to_].push(uint32(id_));
    }
```

Recommendations

Either support up to `uint256` Chunk ID/Indexes or ensure that chunk ID and total ERC721 supply does not exceed `uint32`.

[M-02] Reroll becomes unfunctional after all tokens have been minted

Severity

Impact: Medium

Likelihood: Medium

Description

The `_processChunksOnTransfer` function in the SFT418 Native is designed to handle the logistics of transferring chunks between addresses based on the balance changes of ERC20 tokens. It determines whether to transfer tokens to the pool or to mint/redeem new ones, contingent upon the state of the balances. Additionally, the `_reroll` function allows for a 50% chance to reroll a chunk into a newly minted one if certain conditions (`maxChunks > mintedChunks`) are met.

```
function _reroll(address from_, uint256 id_) internal virtual override
  (SFT418Core) {
    require(from_ == _chunkToOwners[id_].owner, "INVALID_FROM");

    if (maxChunks > mintedChunks) {
        uint256 _mintRng = _getRNG(1);

        if ((_mintRng % 100) > 49) { // 0-49, 50-99
            _ERC721Transfer(from_, TOKEN_POOL, id_);
            _ERC721MintSequential(from_);
        }
        else {
            uint256 _rng = _getRNG(uint160(from_));
            uint256 _poolOwnedLength =  _ownerToChunkIndexes[TOKEN_POOL].length;
            require(_poolOwnedLength > 0, "NO_ITEMS_TO_REROLL");

            uint256 _rngIndex = _rng % _poolOwnedLength;
            uint256 _rngId = _ownerToChunkIndexes[TOKEN_POOL][_rngIndex];

            _ERC721Swap(from_, TOKEN_POOL, id_, _rngId);
        }
    }
}
```

POC Output

```
Running 1 test for test/SFTNativeExampleTest.t.sol:SFTMintTest
[PASS] testDOSReroll() (gas: 162416398)
Logs:
  mintedChunks = 1337

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 107.20ms
```

POC Code

```
function testDOSReroll() public {
    address a = address(this);
    uint256 tokenAmount = 1e18;
    hevm.prank(a);
    token.toggleChunkProcessing();
    hevm.prank(a);
    token.mint(a, tokenAmount);

    for (uint256 i = 0; i < 1336; i++) {
        hevm.prank(a);
        token.transfer(b, tokenAmount);
        hevm.prank(b);
        token.transfer(a, tokenAmount);
    }
    console.log("mintedChunks = %s", pToken.mintedChunks());
}
```

Recommendations

To mitigate the issue consider adjusting the reroll function as following:

```
function _reroll(address from_, uint256 id_) internal virtual override
  (SFT418Core) {
    require(from_ == _chunkToOwners[id_].owner, "INVALID_FROM");
+    uint256 _mintRng = _getRNG(1);
-    if (maxChunks > mintedChunks) {
+    if ((maxChunks > mintedChunks) && (_mintRng % 100) > 49)) {
-        uint256 _mintRng = _getRNG(1);

-        if ((_mintRng % 100) > 49) { // 0-49, 50-99
            _ERC721Transfer(from_, TOKEN_POOL, id_);
            _ERC721MintSequential(from_);
        }
        else {
            uint256 _rng = _getRNG(uint160(from_));
            uint256 _poolOwnedLength =  _ownerToChunkIndexes[TOKEN_POOL].length;
            require(_poolOwnedLength > 0, "NO_ITEMS_TO_REROLL");

            uint256 _rngIndex = _rng % _poolOwnedLength;
            uint256 _rngId = _ownerToChunkIndexes[TOKEN_POOL][_rngIndex];

            _ERC721Swap(from_, TOKEN_POOL, id_, _rngId);
-        }
    }
}
```

8.3. Low Findings

[L-01] Incorrect Check in `_reorderChunk`

The `_reorderChunk` function in the contract is designed to allow the reordering of chunks from one index to another. However, due to a problematic conditional check, the function prematurely exits without performing the reorder when the `indexTo_` specified is the last index in the user's chunk collection.

The conditional statement `if (_lastIndex == indexTo_) return;` is intended to check if the chunk is already at the desired position, in which case no action is needed. However, it currently checks if the target index is the last index in the array, which is a valid position for reordering. This results in the function not executing the reorder if a user attempts to move a chunk to the last position, despite it being a legitimate operation.

To resolve this issue, the condition should be corrected to compare the current index of the chunk (`_currentIndex`) with `indexTo_`. If these are equal, then the function should return as the chunk is already in the desired position. This change will allow users to reorder their chunks to any valid position, including the last one.

```
function _reorderChunk
      (address from_, uint256 id_, uint256 indexTo_) internal virtual {
        // from_ must be the owner of id_
        require(from_ == _chunkToOwners[id_].owner, "INVALID_OWNER");

        // The index to be reordered to must be in-bounds
        uint256 _lastIndex = _ownerToChunkIndexes[from_].length - 1;
        require(_lastIndex >= indexTo_, "OOB");

-        // If the _lastIndex and indexTo_ is the same, simply return
-        if (_lastIndex == indexTo_) return;

        // Otherwise, reorder the indexes
        uint32 _currentIndex = _chunkToOwners[id_].index;
        uint32 _idAtIndexTo = _ownerToChunkIndexes[from_][indexTo_];

+        if (_lastIndex == _currentIndex ) return;

        _ownerToChunkIndexes[from_][_currentIndex] = _idAtIndexTo;
        _ownerToChunkIndexes[from_][indexTo_] = uint32(id_);

        // Then, save the new indexes on both chunks
        _chunkToOwners[id_].index = uint32(indexTo_);
        _chunkToOwners[_idAtIndexTo].index = uint32(_currentIndex);

        // Emit a ChunkReordered Event
        emit ChunkReordered(from_, id_, _currentIndex, indexTo_);
    }
```

[L-02] Missing token id existence check

The EIP-721 specification states that "every ERC-721 compliant contract must implement the ERC721 interface (subject to caveats below)".

For the `getApproved()` function, it states that it "Throws if `_tokenId` is not a valid NFT":

```
/// @notice Get the approved address for a single NFT
@>  /// @dev Throws if `_tokenId` is not a valid NFT.
    /// @param _tokenId The NFT to find the approved address for
    /// @return The approved address for this NFT, or the zero address if there
    // is none
    function getApproved(uint256 _tokenId) external view returns (address);
```

This breaks compatibility with the ERC-721 standard and any integrations expected to follow it.

Consider reverting if the token id is not a valid NFT:

```
function _FB721GetApproved(uint256 id_) internal virtual view returns
      (address) {
+       require(_chunkToOwners[id_].owner != address(0), "INVALID_ID");
        return _getApproved[id_];
    }
```