Protocol Audit Report

Version 1.0

L.Petroulakis

October 14, 2024

Halborn CTF Audit Report

L.Petroulakis

October 13, 2024

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

Table of Contents

  - [H-5] Logical error in `HalbornLoans::returnLoan` adds the amount to `usedCollateral` instead of subtracting it.
  - [H-6] Poor Collateral management in `HalbornLoans` can lead to under-collateralization.
  - [H-7] Missing constructors with `_disableInitializers()` can lead to unintended initialization or reinitialization of the implementation contracts (Root + Impact)
  - [H-8] `HalbornLoans` sets `collateralPrice` in a constructor while it should be using the initiliaze function instead
  - [H-9] Reentrancy in `HalbornLoans::withdrawCollateral` can potentially drain the protocol's funds

- Medium
  - [M-1] Missing Events Emission for critical operations leads to less transparency and loss of auditability
- Low
- Informational
- Gas

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

I sincerely hope this audit report has managed to debug the most critical/high vulnerabilities the CTF wished to demonstrate.

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 Access Control in all three contracts' `authorizeUpgrade()` functions **Description**

All three contracts are implementation contracts using the UUPS pattern. That means that a function like `upgradeTo()` must be called from their respective proxy contract to upgrade the implementation contract. However, the `authorizeUpgrade()` function in all three contracts does not have any access control, meaning that anyone able to potentially access the proxy contract's upgrade function, may call this function and authorize an upgrade.

Impact

When `_authorizeUpgrade()` lacks access control, anyone who can access the `upgradeTo()` function in the proxy could potentially trigger an unauthorized upgrade.

Proof of Concept

1. To exploit this vulnerability, the attacker must first gain access to the upgradeTo function of the proxy contract.
2. If the upgradeTo function in the proxy is public or lacks proper access control (e.g., onlyOwner), the attacker can call it directly to switch the proxy's implementation to a malicious contract.
3. The proxy contract would then delegate all calls to the new (malicious) implementation, enabling the attacker to execute arbitrary code.

Recommended mitigation

Consider implementing access control mechanics in the `_authorizeUpgrade` functions of all three contracts to ensure that only authorized addresses can perform upgrdes to the protocol. A good solution is using OpenZeppelin's `Ownable` contract to provide such functionality securely.

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

[H-2] Missing access control in `HalbornNft::setMerkleRoot()` leads to anyone being able to set the MerkleRoot and mint for themselves

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;
}
```

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.

```
interface IhalbornNFT {
    function setMerkleRoot(bytes32 _merkleRoot) external;
}

contract AttackSetMerkleRoot {
    function attack(address halbornNFTaddress, bytes32
        _maliciousMerkleRoot) external {
        IhalbornNFT(halbornNFTaddress).setMerkleRoot(
            _maliciousMerkleRoot);
    }
}
```

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

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

[H-3] Wrongful Require statement in `HalbornNFT::minAirdrops` enhances the opposite effect.

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

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

This require statement is meant to be the exact 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

Change 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-4] 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:

```
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.

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

[H-5] Logical error in `HalbornLoans::returnLoan` adds the amount to `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-6] Poor 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 will 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.

```

interface IHalbornLoans {
    function depositNFTCollateral(uint256 id) external;
    function getLoan(uint256 _amount) external;
    function withdrawCollateral(uint256 id) external;}

interface IERC721 {
    function approve(address to, uint256 tokenId) external;}

contract exploitHalbornLoans {
    IHalbornLoans public halbornLoans;
    IERC721 public nft;
    uint256 public exploitId;

    constructor(address _halbornLoans, address _nft, uint256 _exploitId
        ) {
        halbornLoans = IHalbornLoans(_halbornLoans);
        nft = IERC721(_nft);
        exploitId = _exploitId;
    }

    function attackHalbornLoans() external {
        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);
}
```

[H-7] Missing constructors with `_disableInitializers()` can lead to unintended initialization or reinitialization of the implementation contracts (Root + Impact)

Description

All three contracts are implementation contracts using the UUPS pattern. The UUPS pattern allows for the separation of the contract's logic from its storage, enabling the upgrade of the contract's logic while preserving the storage. However, the contracts do not include a constructor that calls `_disableInitializers()` from the `Initializable` contract, which can lead to unintended initialization or reinitialization of the implementation contracts.

Impact

The absence of constructors with `_disableInitializers()` can lead to unintended initialization or reinitialization of the implementation contracts, potentially causing unexpected behavior or vulnerabilities.

Proof of Concept

Unauthorized initialization attack on `HalbornLoans`

1. Suppose the proxy is set up to point to this implementation contract.
2. Since the constructor does not include `_disableInitializers()`, an attacker can directly interact with this implementation and call initialize().
3. The attacker sets token and nft to their own addresses. Any function using token or nft (such as token minting, burning, or collateral management) will now be compromised and directed to the attacker's contracts instead of the intended HalbornToken and HalbornNFT.

Recommended mitigation

As per the recommended best practices, consider adding the following.

```
+ constructor() {
        _disableInitializers();
    }
```

[H-8] `HalbornLoans` sets `collateralPrice` in a constructor while it should be using the initiliaze function instead

Description

Since the proxy does not call the implementation's constructor, any values set in the constructor won't be stored in the proxy's storage.

Impact `collateralPrice` will remain uninitialized in the proxy, leading to incorrect contract behavior.

Proof of Concepts Upgradeable contracts rely entirely on the initialize function to set state variables correctly. Any deviation from this pattern can easily lead to a deployment with missing or incorrect initial state.

Recommended mitigation Consider moving the initialization of `collateralPrice` to the `initialize` function.

[H-9] Reentrancy in `HalbornLoans::withdrawCollateral` can potentially drain the protocol's funds

Description The `withdrawCollateral` function in `HalbornLoans` allows users to withdraw their collateral. However, the function is susceptible to reentrancy attacks due to the lack of proper checks and the order of operations. The function does not follow the CEI pattern and is not protected by OZ's ReentrancyGuard. It first transfers the NFT to the user and then updates the collateral state, which can lead to reentrancy attacks.

Impact Repeated Collateral Inflation: By repeatedly redepositing the NFT and withdrawing it without completing each state change, the attacker could artificially inflate their `totalCollateral`, creating a false balance that they could exploit in other functions (like obtaining loans).

Proof of Concept An attacker can call `withdrawCollateral(id)` to initiate the NFT withdrawal, then exploit the reentrancy opportunity in `safeTransferFrom` by calling `depositNFTCollateral(id)` within `onERC721Received`, attempting to deposit the NFT again before the state changes in `withdrawCollateral` are fully applied.

Proof of Code The attacker deploys their own contract

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

import "./HalbornLoans.sol";
import "@openzeppelin/contracts/token/ERC721/IERC721Receiver.sol";

contract ReentrancyAttack is IERC721Receiver {
    HalbornLoans public halbornLoans;
    uint256 public targetNFTId;

    constructor(HalbornLoans _halbornLoans) {
        halbornLoans = _halbornLoans;
    }

    function attackLoans(uint256 nftId) external {
        targetNFTId = nftId;
        halbornLoans.withdrawCollateral(targetNFTId);
    }

    function onERC721Received(address, address, uint256, bytes calldata
        ) external override returns (bytes4) {
        halbornLoans.depositNFTCollateral(targetNFTId);
        halbornLoans.withdrawCollateral(targetNFTId);
        return IERC721Receiver.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 id) external {
        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];
    }
+ function withdrawCollateral(uint256 id) external nonReentrant {
        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);
}
```

Medium

[M-1] Missing Events Emission for critical operations leads to less transparency and loss of auditability

Description Emitting events is critical for maintaining transparency and traceability of significant contract actions, especially when state changes affect user funds or contract configurations. Missing event emissions may not be directly exploitable like traditional vulnerabilities, but they can create security and operational risks, particularly in scenarios involving user funds or access changes.

Impact Loss of auditability, leading to undetected changes in critical contract parameters.

Recommended mitigation Consider adding events to critical functions like `setMerkleRoot` and `setPrice` to ensure transparency and traceability of contract actions

```
+ event MerkleRootUpdated(bytes32 indexed oldMerkleRoot, bytes32
    indexed newMerkleRoot);
+ event PriceUpdated(uint256 indexed oldPrice, uint256 indexed newPrice
    );

function setMerkleRoot(bytes32 merkleRoot_) public onlyOwner {
+     emit MerkleRootUpdated(merkleRoot, merkleRoot_);
    merkleRoot = merkleRoot_;
}

function setPrice(uint256 price_) public onlyOwner {
    require(price_ != 0, "Price cannot be 0");
+     emit PriceUpdated(price, price_);
    price = price_;
}
```

Low

Informational

Gas