Halborn - CTF

Recruitment Security Assessment

Prepared by: 0xumarkhatab

Date of Engagement: February 23rd , 2024 - February 24th , 2024

Visit: Halborn.com

Document Summary

Contacts

Name	Company	Email
Umar Khatab	0xumarkhatab	umarkhatabfrl@gmail.com

EXECUTIVE OVERVIEW

1. INTRODUCTION

HalbornLoans provides loans for HalbornToken to the users based on the HalbornNFT.
Here’s what users of the protocol can do

- Deposit HalbornNFTs as collateral . Each NFT deposit will gain the user some **`collateralPrice`** credit that they can use later for getting loans
- Withdraw HalbornNFTs from the protocol.
- Get HalbornToken Loan, Effectively Minting new HalbornTokens based on the amount of credit they have after depositing the HalbornNFTs in the protocol.
- Return the loan, effectively burning their shares of HalbornToken.

0xumarkhatab is here to secure Halborn Smart contracts and secure users' funds on chain.

1.1 ASSESSMENT OVERVIEW

0xumarkhatab , took 2 days to analyse the smart contracts based off of Ethereum and solidity CTF . 0xumarkhatab is a security researcher with nobel experience in the field who has started his programming adventure with low-level languages like c++ , assembly and c. It’s the matter of time he realised the potential of blockchains and devoted 3 years in developing secure on-chain applications. Then he turned to leveraging his abilities to secure funds on-chain , however his major focus for the future is zero knowledge proof systems as he’s currently learning PLONKish frameworks on his own.
The purpose of this assessment is to:

- Identify potential security issues within the programs

In Summary , 0xumarkatab has found some security vulnerabilities in the CTF code he was given for solidity and Ethereum.

1.2 Test & Approach

0xumarkhatab has performed manual review of the source code.

Manual testing is recommended to uncover flaws in business logic, processes, and implementation.

The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture, purpose, and use of the platform.
- Manual program source code review to identify business logic issues.
- Mapping out possible attack vectors
- Thorough assessment of safety and usage of critical solidity variables
- and functions in scope that could lead to arithmetic vulnerabilities.
- Finding unsafe solidity code usage (unchecked blocks)

2. RISK METHODOLOGY

Every vulnerability and issue observed by 0xumarkhatab is ranked similar to what Halborn does. As they state (i have paraphrased a bit) ,

“Each issue is based on two sets of Metrics and a Severity Coefficient

This system is inspired by the industry standard Common Vulnerability Scoring System. The two Metric sets are: Exploitability and Impact.

Exploitability

It captures the ease and technical means by which vulnerabilities can be exploited and Impact describes the consequences of a successful exploit.

Severity Coefficients

The Severity Coefficients is designed to further refine the accuracy of the ranking with two factors: Reversibility and Scope. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart Contracts.

The system is designed to assist in identifying and prioritising vulnerabilities based on their level of risk to address the most critical issues in a timely manner. “

– *Halborn sol - solana Audit Report*

Here’s how, in summary , 0xumarkhatab scores the severity of the issue.

Severity	Score Value Range
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

2.1 Assets in scope

Smart contracts that are in-scope are following

- HalbornToken
    - ERC20 Token
    - Upgradable
    - Ownable
    - Initializable
    - Multicall enabled
- HalbornNFT
    - ERC721 Token
    - Upgradable
    - Ownable
    - Initializable
    - Multicall enabled
- HalbornLoans
    - Upgradable
    - Initializable
    - Multicall enabled

The HalbornLoans contract is the main smart contract that uses both HalbornToken and HalbornNFT contracts for collateral and loan purposes.

2.2 Findings Summary

Severity	Instances
Critical	8
High	4
Medium	4
Low	0

FINDINGS & TECH DETAILS

3 Findings and Tech Details

This section will demonstrate all the findings I've found so far.

Note :-

- All of the findings are independent of each other until and unless linked explicitly. It is because some of the findings will seem linked (vulnerability on top of another vulnerability) but for the sake of this Audit , i wanted to come up with all of the possible attack vectors that can exploit the Halborn Loan Platform .
- Additionally , when stating particular findings , I have focused more on the particular attack vector in the specified set of functions; if there are other issues contained in those functions as well (which I'll state in subsequent findings), they are gracefully ignored.
  Please remember it.
- For this engagement , the main focus was criticals , highs and mediums instead of low/informational vulnerabilities to provide the utmost value to the protocol.
- The important thing to remember is that I hereby agree that some bugs might still be missed and this audited code would not be 100% bug free. For always trying to get secure , the developers need to have a closer look at the latest trends and security best practices.

3.1 Users can, for free , get more NFTs using airdrop than intended using proof replay - CRITICAL

Description

Inside HalbornNFT smart contract , we have an external function MintAirdrops that intends to Mint airdrops to the users who has a valid proof of the merkle tree . They provide the new NFT Token id and the correct proof of the merkle tree and the airdrop function mints an NFT on their address.

When we closely have a look at the function , we can see that there is no caching of the proof that some proof has already been used for airdrop and their is no updation of the merkle root after the airdrop , rather it just verifies the proof and mints the NFT for user.

Impact

The protocol will incur loss of NFTs and funds because the NFTs are to be purchased By sending `price` amount of Eth to the smart contract . Also the other users who have bought the NFTs by giving the price to the protocol and not attacking it will feel cheated By the fact that they could get the NFT for free. So there will be loss of Trust too.

Code Location

HalbornNFT.sol - mintAirdrops

```
function mintAirdrops(uint256 id, bytes32[] calldata merkleProof) external {
    require(_exists(id), "Token already minted");

    bytes32 node = keccak256(abi.encodePacked(msg.sender, id));
    bool isValidProof = MerkleProofUpgradeable.verifyCalldata(
        merkleProof,
        merkleRoot,
        node
    );
    require(isValidProof, "Invalid proof.");

    _safeMint(msg.sender, id, "");
```

```
        }
```

Proof Of Concept

Here is a sample PoC that you can use :

```
        HalbornNFT nft = new HalbornNFT();
        nft.initialize(root, 1 ether);
        bytes32[] memory data = new bytes32[](4);
        data[0] = keccak256(abi.encodePacked(ALICE, uint256(15)));
        data[1] = keccak256(abi.encodePacked(ALICE, uint256(19)));
        data[2] = keccak256(abi.encodePacked(BOB, uint256(21)));
        data[3] = keccak256(abi.encodePacked(BOB, uint256(24)));
        nft.mintAirdrops(1, data);
        nft.mintAirdrops(2, data);
        nft.mintAirdrops(3, data);
        nft.mintAirdrops(4, data);
```

Recommendation

Add Caching Mechanism for proofs that has been used and most importantly , update the markleroot to the updated one.

3.2 Unlimited Loan using getLoan without depositing any NFT collateral - CRITICAL

Description

Inside HalbornLoans Contract , we can see the function getLoan which gives the loan based upon “enough collateral” requirement. However , the way enough collateral is being checked is incorrect . Instead of checking if the amount to be loaned is less or equal to what the user can loan , it checks if the collateral is less than the amount that is to be loaned.

So if the user of the protocol has deposited less than the amount collateral (which can even be 0) , the protocol will freely give the loan to the caller by minting tokens to its address.

Impact

Users can loan any amount of tokens without having any collateral.
As there is no upper limit on minting , the user can mint as much as 2^256 = 10 ^ 77 HalbornTokens

Code Location

HalbornLoans.sol - getLoan

```
    function getLoan(uint256 amount) external {
        require(
     totalCollateral[msg.sender] - usedCollateral[msg.sender] < amount,
            "Not enough collateral"
        );
        usedCollateral[msg.sender] += amount;
        token.mintToken(msg.sender, amount);
    }
```

Proof Of Concept

Let’s say `price` is the price of the NFT to be purchased with and `tc` and `uc` are totalCollateral and UsedColalteral of the person.

Then look at this scenario :

```
Initially


price=10

tc = 0

uc=0


User calls deposit -> tc=10 , uc=0

User calls getLoan

require(10-0<uint256.max ,”Not enough collateral”) // passes
```

As 10 is less than any amount greater than 10 (potentially some large number like uint256.max) , the check will pass and the tokens will be minted on the user's address.

Recommendation

Change the require condition to correct one.

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

3.3 Use of delegatecall in a payable Multicall function inside a loop - CRITICAL

Description

The HalbornNFT contract uses multicall and inherits its externally callable multicall function that users can call to make multiple transactions in one call to the function. This function uses delegate call and it is payable. And takes an array of user-provided call data, iterates over it in a loop and executes Each calldata in the context of itself.

But there is an issue , as the function is payable it means it can receive some ether as msg.value and the caller will be msg.sender .
The delegatecall within the for loop however retains the msg.value of the transaction

If an attacker calls the multicall of halbornNFT with 1 ether , crafts a calldata that makes a call to a payable function like mintBuyWithEth in this case , the msg.value will always be 1 ether if it's the first call to mint his first halbornNFT Or 100000th. This way he could use this vulnerability to get 100000000 or even more NFTs and then sell in the market . Making a decent amount of profit.

Impact

We will see following impacts :

- Loss of NFTs from the protocol
- Loss of funds which the protocol was intended to collect for selling the NFTs
- Loss of market value of NFTs if the person gets so many NFTs , he might sell at lower price at other places in the market

Code Location

HalbornNFT.sol - - Multicall.sol - library # multicall

```
    function multicall(
        bytes[] calldata data
    ) external payable returns (bytes[] memory results) {
        results = new bytes[](data.length);
        for (uint256 i = 0; i < data.length; i++) {
            results[i] = AddressUpgradeable.functionDelegateCall(
                address(this),
                data[i]
            );
        }
        return results;
    }
```

Recommendation

Well it depends upon what protocol intends to do but for now , i think a good idea would be to not let the payable functions get called from inside multicall by removing the payable keyword from its definition.

So the function definition would look like follows

```
    function multicall(
        bytes[] calldata data
    ) external returns (bytes[] memory results) {
```

3.4 Front running setMerkleRoot can stop the functionality of mintAirdrops - CRITICAL

Description

Halborn NFT has a setMerkleRoot function that it uses to set the value of merkleroot which is Used inside mintAirdrops function to mint NFTs for people who has correctly computed the proof of the new merkle tree resulting from including their proposed id.

Unfortunately , this function can be front-runned by a bot repeatedly and not letting anyone set the new merkleroot which might be a valid one .

This could be done either for fun or damaging the reputation of the protocol or using this vulnerability Combined with others (as we shall see in the next vulnerability) to exploit the system.

Impact

This will have following impacts :

- This scenario will enable the attacker to make airdrops unavailable for the users which will be frustrating. And protocol will lose trust and traction from people who care for airdrops , facing loss of revenue that might be generated from those people .
- Secondly , the protocol has intended to keep the airdrops alive and that might be their core focus towards Giving to and making a broad community but their core functionality would be broken which will have Some serious consequences
- Last but not the least, the protocol might want to upgrade their NFT contract and include more functions that rely on merkleTree proofs that deals closely with money , in that case it will cause loss of funds too.

Code Location

HalbornNFT.sol - setMerkleRoot

```
function setMerkleRoot(bytes32 merkleRoot_) public {
    merkleRoot = merkleRoot_;
  }
```

Proof Of Concept

This is a simple idea to exploit by making a front-running bot and setting merkleRoot to an invalid bytes data and watching over mempool for halbornNFT.setMerkleRoot function calls . Whenever the attacker's bot sees this type of transaction, it generates a similar transaction by passing the same previous one and gives a relatively high gas to keep delaying the transaction of setting a valid merkle root.

Recommendation

Add onlyOwner modifier to setMerkleRoot so that only the privileged address can call this function.

3.5 DOS attack for minting Unlimited NFTs as airdrops using front running by using proof replay - CRITICAL

Description

Using the issue #3.1 (Users can, for free , get more NFTs using airdrop than intended using proof replay) and issue #3.4 (No modifier on setMerkleRoot can be front-runned to stop the functionality of mintAirdrops) with a combined effort , a clever front-runner can make a bot to mint unlimited NFTs using airdrops until the max limit of uin256
(2^256 tokens - astronomically large amount) is reached.

This is clearly the most important critical issue.

Impact

- Loss of funds for protocol .
- System aidrops functionality and any other functions relying on correct merkleRoot proof can be stopped as long as the attacker's bot is running.

Code Location

HalbornNFT.sol - mintAirdrops

```
    function setMerkleRoot(bytes32 merkleRoot_) public {
        merkleRoot = merkleRoot_;
    }
    function mintAirdrops(uint256 id, bytes32[] calldata merkleProof) external {
        require(_exists(id), "Token already minted");

        bytes32 node = keccak256(abi.encodePacked(msg.sender, id));
        bool isValidProof = MerkleProofUpgradeable.verifyCalldata(
            merkleProof,
            merkleRoot,
            node
        );
        require(isValidProof, "Invalid proof.");

        _safeMint(msg.sender, id, "");
    }
```

Proof Of Concept

Take following scenario ,

- Attacker sets the merkle root to one that includes no nodes at all (setMerkleRoot)
- Attacker generates a valid proof and spins his bot to watch the mempool
- When someone tries to update merkleRoot using setMerkleRoot , the bot front runs it
  And sets the existing merkleRoot (having no child nodes) again using setMerkleRoot.
- While in parallel maybe making another bot or the same bot or by himself ,
  Attacker calls mintAirdrops again and again to mint all the NFTs on his address.
- Attacker sells these NFTs and make profit.

Recommendation

The protocol can mitigate this critical vulnerability by :

- Marking setMerkleRoot as onlyOwner to eliminate un-authorized access
- Caching proofs inside mintAirdrops to eliminate proof replay attack
- Updating the merkleRoot with updated root after node insertion inside mintAirdrops.

3.6 Uninitialized collateralPrice variable can deem entire HalbornLoans protocol to be useless - CRITICAL

Description

According to the code of HalbornLoans, we can see it is Universal proxy based implementation (UUPS) where we expect the implementation contract to be deployed by the proxy and then proxy initialise the contract values in the storage context of Proxy when proxy calls its initialize function.

However , we see that code is setting the value of collateralPrice in the constructor which will be kept just locally and when the initialize function is called , we are not storing any collateralPrice variable in the context of the proxy .

This will lead the collateralPrice to be zero in the storage context of the proxy the users interact with to access HalbornLoans.

Each time collateralPrice is going to be accessed , it will be accessed from proxy's storage because there will be a delegate call from proxy to implementation contract using context of proxy for implementation and the price will be zero because it has not been initialised into proxy's storage

Impact

This will impact the workings of all the core functions :

- Users will not get any collateral credit after depositing NFTs
- Withdraw will be working
- getLoan and returnLoan will be totally stopped because user will not have any collateral credits to operate with

Code Location

HalbornLoans.sol

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


  function initialize(address token_, address nft_) public initializer {
      __UUPSUpgradeable_init();
      __Multicall_init();


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

Recommendation

Move collateralPrice updation inside initialize function of HalbornLoans instead of initialisation in constructor.

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

```
        collateralPrice = collateralPrice_;
    token = HalbornToken(token_);
    nft = HalbornNFT(nft_);
  }
```

3.7 Invalid NFT Existence Checks will always make the functions revert - CRITICAL

Description

The _exists method from ERC721 is used at multiple places in the code where the logic of the require statement is incorrect. The functions should move forward to minting only if the NFT Token Id does not exist. But in the current implementation , the function tries to mint already minted tokens And it will revert to safeMint because it is already minted.

But interestingly , it will always revert for non-existing token Ids because require statement will be false

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

As in the above code , the require will always fail for minting new tokens because _exists will return false

Impact

Broken Functionality of Airdropping leaving protocol to be stale

Code Location

HalbornNFT.sol - mintAirdrop &

```


      function mintAirdrops(uint256 id, bytes32[] calldata merkleProof) external {
          require(_exists(id), "Token already minted");

          bytes32 node = keccak256(abi.encodePacked(msg.sender, id));
          bool isValidProof = MerkleProofUpgradeable.verifyCalldata(
            merkleProof,
            merkleRoot,
            node
```

```
        );
        require(isValidProof, "Invalid proof.");

        _safeMint(msg.sender, id, "");
    }
```

Recommendation

We just need a small change to get things working . Add the not operator in existence check as :

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

3.8 Incorrect Calculation of usedCollateral can permanently freeze user's NFTs - CRITICAL

Description

The used collateral mapping is intended to know how much of the available totalCollateral a person has used. It is incremented by the amount of loan a person is borrowing in getLoan Function. It is supposed to be decremented by the amount of loan tokens the person is returning back.

However , the current implementation is incorrectly modifying the usedCollateral mapping. It is also increasing the owed amount when returning the loan.

This is a logical implementation error that will cause protocol to always think that people
still has to return the loan even if they did and not let them withdraw their collateral.
This is not an intended behaviour

Impact

- Freezing Collateral withdrawal of users
- Loss of Trust of users on protocol
- Breaks the core functionality of Lending and Borrowing of HalbornLoans

Code Location

HalbornLoans.sol

```
    function withdrawCollateral(uint256 id) external {
        require(
          totalCollateral[msg.sender] - usedCollateral[msg.sender] >=
            collateralPrice, "Collateral unavailable"
        );
```

See if usedCollateral is more than totalCollateral (usedCollateral is always increasing)

```
     function getLoan(uint256 amount) external {
                 //
         usedCollateral[msg.sender] += amount;
                 //
       }
     function returnLoan(uint256 amount) external {
         //
         usedCollateral[msg.sender] += amount;
         //
       }
```

Recommendation

Subtract amount from UsedCollateral in returnLoan Function

3.9 The reentrancy vulnerability in _safeMint can allow an attacker to steal all NFTs - High

Description

There is a reentrancy vulnerability in the _safeMint function

```
function _safeMint(
    address to,
    uint256 tokenId,
    bytes memory _data
  ) internal virtual {
    _mint(to, tokenId);
    require(
      _checkOnERC721Received(address(0), to, tokenId, _data),
      "ERC721: transfer to non ERC721Receiver implementer"
    );
  }
  ...
  function _checkOnERC721Received(
      address from,
      address to,
      uint256 tokenId,
      bytes memory _data
  ) private returns (bool) {
      if (to.isContract()) {
          try IERC721Receiver(to).onERC721Received(_msgSender(), from, tokenId, _data)
returns (bytes4 retval) {
              return retval == IERC721Receiver.onERC721Received.selector;

}
```

If we look at the code of Minting Function , "MintAirdrops" in HalbornNFT, We can see that the re-entrancy vulnerability exists because we are not updating any state Variables that would stop the re-entrancy . Rather there are only tokenId existence check and proof validation but nothing to act like a re-entrant modifier.

Which shows that a clever NFT Minter can steal all the NFTs using re-entrancy in "mintAirdrops" method.

Impact

The severity of issue is high because of the impact involves
Loss of All NFTs to the platform leading to lose of future revenue

Code Location

HalbornNFT.sol - mintAirdrops

```
        function mintAirdrops(uint256 id, bytes32[] calldata merkleProof) external {
            require(_exists(id), "Token already minted");

            bytes32 node = keccak256(abi.encodePacked(msg.sender, id));
            bool isValidProof = MerkleProofUpgradeable.verifyCalldata(
                merkleProof,
                merkleRoot,
                node
            );
            require(isValidProof, "Invalid proof.");
            _safeMint(msg.sender, id, "");
        }
```

Recommendation

Use a non-reentrant modifier from openzeppelin or create a custom re-entrant modifier

3.10 Malicious or hacked Owner can rug users by updating the NFT and Token Contracts to malicious addresses - High

Description

You would argue that trusted accounts do not subject to this issue but if the private key of the owner Is hacked (which happens a lot these days unless it’s a multisig), or owner becomes malicious, the owner can change the addresses of the underlying NFT and Token contracts using the initialise method .

The User might think they are interacting with legit contracts however the malicious owner can hiddenly implement any logic that would cause loss of funds to the user.

Even just changing the NFT and Token contract addresses will be a huge loss for users because they might think they are paying price for something valuable but all they might get is just a mapping entry on blockchain and nothing else.

Although this is unlikely , it is possible and hence remains the reason of anxiety for the token holders.

Impact

- Loss of funds for user when they don’t get what they pay for
- Distress in the community about upgrading power to owner

Code Location

HalbornNFT.sol

```
contract HalbornNFT is
    Initializable,
    ERC721Upgradeable,
    UUPSUpgradeable,
    OwnableUpgradeable,
    MulticallUpgradeable
{
    bytes32 public merkleRoot;

    uint256 public price;
    uint256 public idCounter;

    function initialize(
        bytes32 merkleRoot_,
        uint256 price_
    ) external initializer {
        __ERC721_init("Halborn NFT", "HNFT");
        __UUPSUpgradeable_init();
        __Ownable_init();
        __Multicall_init();

        setMerkleRoot(merkleRoot_);
        setPrice(price_);
    }
```

HalbornToken.sol

```
contract HalbornToken is
    Initializable,
    ERC20Upgradeable,
    UUPSUpgradeable,
    OwnableUpgradeable,
    MulticallUpgradeable
{
    address public halbornLoans;

    modifier onlyLoans() {
        require(msg.sender == halbornLoans, "Caller is not HalbornLoans");
        _;
    }

    function initialize() external initializer {
        __ERC20_init("HalbornToken", "HT");
        __UUPSUpgradeable_init();
        __Ownable_init();


        __Multicall_init();
    }
```

Recommendation

Consider disabling the owner to upgrade the logic if the contract is already initialised, which terminates the possibility of rugging users with malicious contracts, Or, putting a timelock to this function at least.

3.11 HalbornToken Deployer can rug users having unlimited minting and burning power - High

Description

Inside HalbornToken , halbornLoans address state variables that serve as the owner who can mint and burn as many tokens as it wants . However while looking at the other contracts , HalbornLoans is a contract whose address is to be appointed as onlyLoans which is safe however there are two very dangerous attack vectors through which unlimited minting and un-authorized burning can happen

Impact

- Loss of NFTs for platform
- Loss of funds of user

Proof Of Concept

We have 2 scenarios :

1. **Upgradability of HalbornLoans**

Currently the Halborn Loans contract seems safe because there is no method that can do arbitrary minting and burning even through privileged users. But remember , the HalbornLoans smart contract is still upgradable by the owner and the owner might be a clever developer who might in future release insert some owner privileged methods to mint and burn arbitrary tokens when people are less likely to notice after some time of the launch of the protocol.

There is no restriction for owner to not burn and mint unlimited NFTs if we see in the code of HalbornNFT.

2. **Deployer of HalbornToken Contract**
   We can see the HalbornToken Contract is ownership; it means whoever has deployed the contract will always be the owner (ownership is never transferred) however we have 2 owners where one is primary and other is replaceable . The primary owner has the power to change the halbornLoans address and that halbornLoans can mint and burn the tokens .

   But wait a minute , if the halbornLoans address is changeable , then who is stopping the primary owner to change this address for some time , do arbitrary minting and burning with the new halbornLoans address and then give the access back to the original secure contract .

   Well no one is . That is why there is a critical issue in the protocol that if users become aware of , they will not want to continue with it anymore.

Code Location

Halborn.sol -

```
function setLoans(address halbornLoans_) external onlyOwner {
     require(halbornLoans_ != address(0), "Zero Address");
     halbornLoans = halbornLoans_;
  }
```

Recommendation

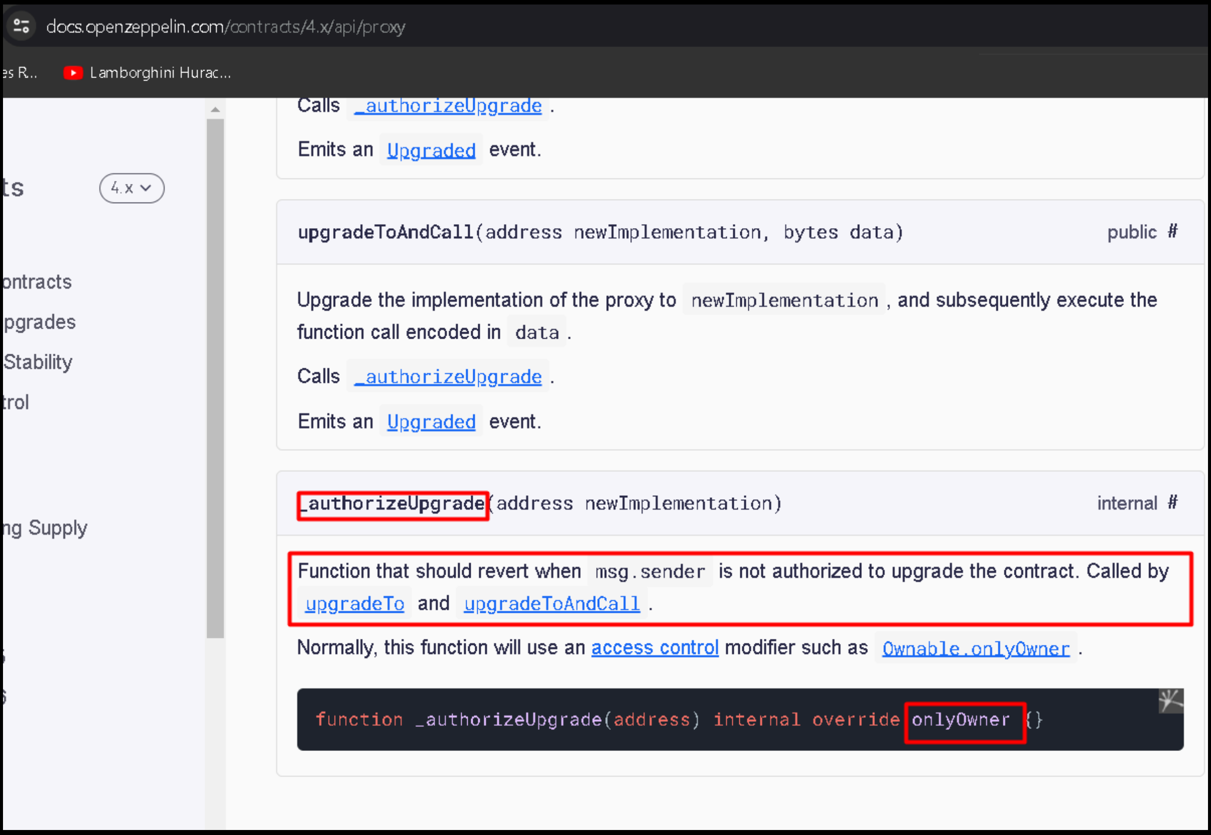
Discuss with the team and try to come up with the suitable ownership management method . I would suggest transferring ownership altogether from the Deployer of HalbornNFT contract to HalbornLoans . This can be done securely through 2-step ownership transfer .

3.12 Lack of access control on _authorizeUpgrade leaves chances for unauthorised upgrades - High

Description

All three smart contracts uses UUPS with the clear intent of upgradability. However ,the core function that is used to upgrade the implementation is not restricted using onlyOwner or onlyInitializer modifiers.

When we look at the official open zeppelin docs for UUPS pattern,
we see this strict guide :



docs.openzeppelin.com/contracts/4.x/api/proxy

Calls _authorizeUpgrade .

Emits an Upgraded event.

`upgradeToAndCall(address newImplementation, bytes data)` public #

Upgrade the implementation of the proxy to `newImplementation` , and subsequently execute the function call encoded in `data` .

Calls _authorizeUpgrade .

Emits an Upgraded event.

`_authorizeUpgrade(address newImplementation)` internal #

Function that should revert when `msg.sender` is not authorized to upgrade the contract. Called by upgradeTo and upgradeToAndCall .

Normally, this function will use an access control modifier such as Ownable.onlyOwner .

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

The smart contract should implement best security practices however , in its current implementation , the smart contract allows un-authorized future upgrades

Impact

Loss of user funds and stopping the entire protocol if someone is able to upgrade this contract by some means of calling this internal function in some way in the future upgrades.
Also it leaves the system un-interoperable because it does not meet the assumptions that every other protocol thinks (_authorizeUpgrade is always restricted using access control).

Code Location

HalbornLoans.sol , HalbornNFT.sol , HalbornToken.sol

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

Recommendation

Add access control in the function by either using onlyOwner or onlyInitialzer modifiers

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

3.13 Sandwich attack on HalbornNFT set price - Medium

Description

HalbornNFT has a onlyOwner modified setPrice function that can change the price the user has to pay while minting the NFT . Unfortunately , this setPrice transaction can be sandwiched to maximise the gains of users using front-running.

See Attack Scenario for more details

Impact

Unfair profit to attacker

Code Location

HalbornNFT.sol - setPrice

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

Attack Scenario

Say the current price of one HalbornNFT is 1 ether.

Alice runs a front running bot on ethereum mainnet and her bot continuously watches over mempool for price change transactions `setPrice` on HalbornNFT contract. When the bot see an increase in price, It sandwiches this setPrice transaction between her buy and sell transactions in following order :

- Buy Transaction of 10 NFTs at low price
- setPrice Transaction with increased price
- Sell transaction to sell 10 NFTs to some marketplace at higher price

When setPrice sets the price to low price , the bot sandwiches transaction in similar order

- Buy Transaction of 10 NFTs at old price
- setPrice Transaction with increased price
- Sell transaction to sell 10 NFTs to some marketplace

Recommendation

Currently , I'm trying to find a solution to this problem if we can.
But I am available for a quick chat on eliminating this type of issue.

3.14 Abrupt price change in HalbornNFT can cause issues to users - Medium

Description

In HalbornNFT contract , there is an owner-only privileged function setPrice that updates the price of the NFT to mint. However , the change can be abrupt. The user might need to be informed early on about this change of price so that they can make informed decisions about their NFTs' investments.
But right now , there is no method like this .

Impact

- Users will not be able to make informed decisions
- Confusion in the community about when will the price be changed and what it will be

Code Location

HalbornNFT.sol -   set price

```
        function setPrice(uint256 price_) public onlyOwner {
```

```
        require(price_ != 0, "Price cannot be 0");
        price = price_;
    }
```

Recommendation

HalbornLoans Protocol should devise a time locked process of changing the price so that users of the HalbornNFT can make informed decisions.

3.15 No Storage Gap for Upgradeable Contract Might Lead to Storage Slot Collision - Medium

Description

As suggested by 0x1337 in his Finding , When using upgradeable contracts, there must be a storage gap to

"allow developers to freely add new state variables in the future without compromising the storage compatibility with existing deployments"

- OpenZeppelin.

Otherwise it may be very difficult to write new implementation code. Without a storage gap, the variable in the child contract might be overwritten by the upgraded base contract if new variables are added to the base contract. This could have unintended and very serious consequences to the child contracts, potentially causing loss of user funds or causing the contract to malfunction completely.

Refer to the bottom part of this article: https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable

Proof of Concept

Several contracts are intended to be upgradeable contracts in the code base, including

HalbornLoans
HalbornToken
HalbornNFT

However, none of these contracts contain storage gap. The storage gap is essential for an upgradeable contract because "It allows us to freely add new state variables in the future without compromising the storage compatibility with existing deployments". Refer to the bottom part of this article:

https://docs.openzeppelin.com/contracts/3.x/upgradeable

As an example, both the AlchemicTokenV2Base and the CrossChainCanonicalBase are intended to act as the base contracts in the project. If the contract inheriting the base contract contains additional variables, then the base contract cannot be upgraded to include any additional variable, because it would overwrite the variable declared in its child contract. This greatly limits contract upgradeability.

Recommendation

Recommend adding appropriate storage gap at the end of upgradeable contracts such as the below. Please reference OpenZeppelin upgradeable contract templates.

```
uint256[50] private __gap;
```

3.16 Paying Less or More loan to HalbornLoans than borrowed - Medium

Description

HalbornLoans has a price associated with each NFT deposit as collateral proportional to how many HalbornTokens the person can loan out. And then give back the loan and get his NFT back.

But there are scenarios when after the user has deposited his NFT and took the loan of tokens , the price of the tokens can fluctuate drastically and people would suffer loss if they bought a loan of tokens when the price is low and has to return loan when the price is high.It is somehow analogous to impermanent Loss in AMMs

The main issue here is that HalbornLoans Protocol here is operating in the amount of Halborn tokens instead of their Real value worth which can be determined using a price feed oracle like Chainlink.

Impact

Loss of funds for users

Code Location

HalbornLoans.sol - getLoan & returnLoan

```
function getLoan(uint256 amount) external {
    require(
        totalCollateral[msg.sender] - usedCollateral[msg.sender] < amount,
        "Not enough collateral"
    );
    usedCollateral[msg.sender] += amount;
    token.mintToken(msg.sender, amount);
}

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

Proof Of Concept

Suppose Alice has a HalbornNFT and she wants to deposit it as collateral and take HalbornLoans. Say the collateralPrice is set to 10^18 , which if you deposit one HalbornNFT , you will get one Halborn Token.

Let's say the current price of HalbornToken is 1 ETH per 1 HalbornToken.
Suppose 1 ETH is valued at 1800$.
Now Alice takes the loan , and she has spent this HalbornToken for buying something for her.

After a month , Alice wants to get her beloved NFT back , but she can only get it back after paying 1 HalbornToken back . Suppose the price of the token reaches the all-time high of 10 ETH which in our scenario would be

6000$ instead of just 1800$.

Alice is required to pay 3x the amount she has borrowed. This is unfair and this will be a dead end for Alice on how to recover her important NFT due to lack of 3x money even though she has arranged the 1800$ which she has borrowed in the first place.

Recommendation

Use a price oracle like Chainlink to fetch the price of the HalbornToken and make a way to give and receive back loans in terms of relatively stable currency.

Conclusion

This marks the completion of my effort to secure HalbornLoans ecosystem . Even though I might have missed some critical points, deep inside I know that I have given my best and left no stones unturned.
This Audit is an effort to join Halborn ecosystem by showing my security abilities which will be improved over time :)

Thank you for reading my report . If you find ways to improve the report , I am open to feedback .

Thanks once again.

Credits :

- Thanks to Thomas Richard for giving this opportunity
- I've taken inspiration from Halborn's theme of their security reports. Thanks to Halborn
- Reading Past reports on Code4rena has helped me a lot in mapping creative possible attack vectors
- My Blockchain development experience

THANK YOU FOR CHOOSING

// HALBORN