Audit Report: Module 3

Prepared by: Marco López González
17/03/2023

Table of Contents

Introduction

Purpose of This Report

I am pleased to present my smart contract audit work, done as part of my recent blockchain course. The main target of this audit was to detect and fix vulnerabilities in selected smart contracts, in order to put into practice all the knowledge acquired in the course.

The goals of the audit are as follows:

1. Identify vulnerabilities in the source code of the smart contracts, in order to guarantee its security and reliability.
2. Ensure that the smart contract complies with the standards and best practices recommended for its development and execution.
3. Assess potential risks, such as potential attack vectors or directly exploitable vulnerabilities.
4. Validate the behavior of the smart contract in different conditions and situations to ensure its robustness and resistance to errors.
5. Identify possible improvements and optimisations in the smart contract source code, in order to improve its performance and efficiency.

As a result of this audit, several important vulnerabilities in the selected smart contracts have been identified and possible solutions have been recommended.

Codebase Submitted for the Audit

The audited smart contracts can be found in the following GitHub repository:

https://github.com/jcsec-security/solidity-security-teaching-resources/tree/main/exercises/src

File	SHA256
VulnerableDAO.sol	3d70683284db53a9523a0bc7bcb4d47f4f7c79cdb1f74b8d3f728be44cae4db1
VulnerableShop.sol	748fc8d2c41935ec6cd1a6290bf92b05042d4e3efcf636264f733cb3a4a9d878
VulnerableBank.sol	51dab969b74490041cd4afc4a55436092058886d18999958081ec59bf1b63178
VulnerableVault.sol	194e7e0ed7e329e6e2658b85fb9edf76e490e7f886aa0df6d1f00bb7baae4d4e

Methodology

The audit has been carried out in the following steps:

1. Understanding the high level purpose of the contract.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and usage of best practice guidelines, including but not limited to the following:
    a. *Reentrancy*
    b. Under-/overflow issues
    c. Key management vulnerabilities
    d. *Weak pseudo-randomness*
    e. Access controls
4. Report preparation

Functionality Overview

The purpose of this audit is to present the security flaws or bad security practices found in the following contracts:

-VulnerableBank.sol
-VulnerableDAO.sol
-VulnerableShop.sol
-VulnerableVault.sol

The results of this mock audit are based on the knowledge acquired in the *University Extension Course on Blockchain Technologies* - especially in *Module 3: Auditing Smart Contracts*.

The functionality of the contracts is briefly explained below.

-VulnerableBank.sol

This Smart Contract implements an investment bank that allows users to invest and distribute the profits to the chosen beneficiaries within a given period of time. It also offers a reward to the user who calls the function that distributes the funds.

-VulnerableDAO.sol

Decentralised Autonomous Organisation's voting system that handles disputes between buyers and sellers. In addition, it implements conducts an NFT lottery to reward users who participate in the DAO.

-VulnerablesShop.sol

This contract allows users to sell and buy items using an ERC20 token. It includes different functionalities such as creating, modifying and deleting sales offers, opening disputes for sellers...

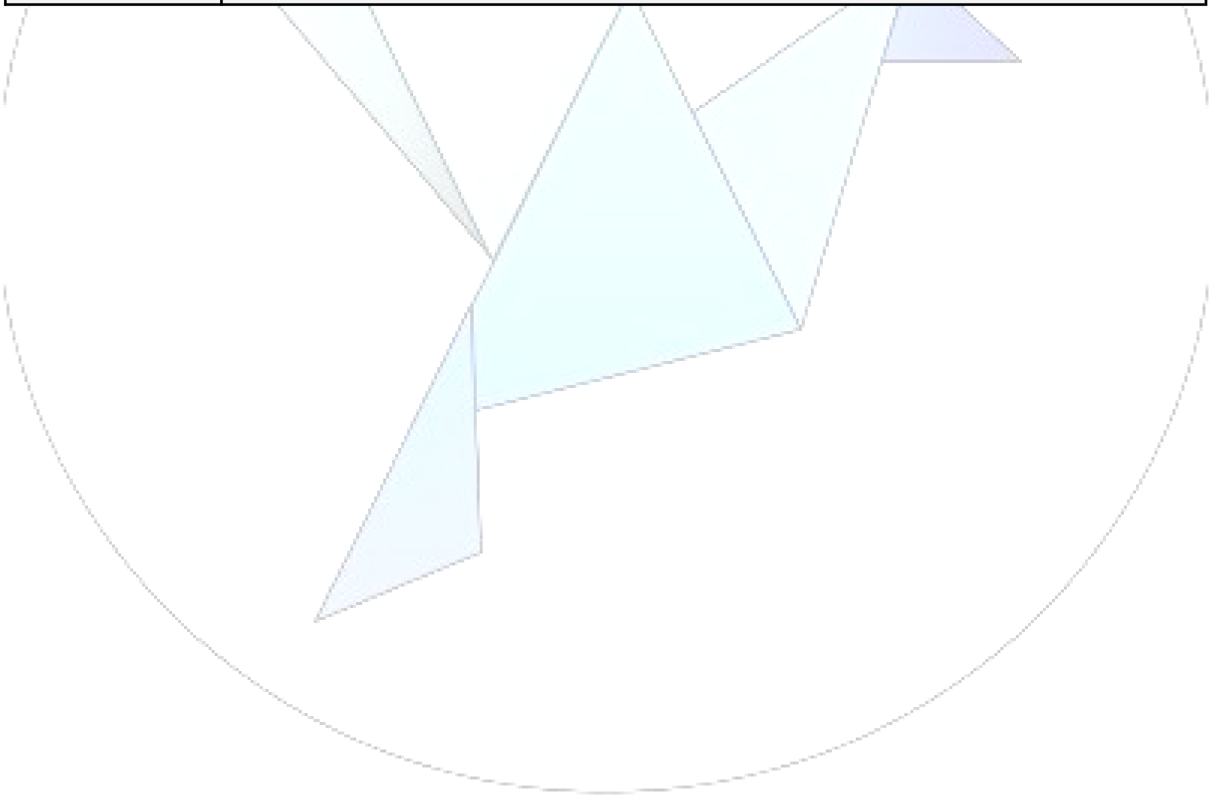
-VulnerableVault.sol

It is a token vault where users can deposit funds, lock them and receive rewards for doing so.

Categories of Severity

This report classifies the issues found into the following severity categories, created by *Oak Security*:

Severity	**Description**
Critical	A serious and exploitable vulnerability that has the potential to cause financial losses, permanent locking of funds, or severe denial-of-service attacks.
Major	A vulnerability or bug that can disrupt the proper operation of a system, resulting in incorrect states or denial of service.
Minor	A breach of common best practices or improper use of primitives that may not pose a significant security threat at present, but could do so in the future or cause inefficiencies.
Informational	Comments and recommendations related to design decisions or potential optimizations that are not related to security. While their implementation may enhance certain aspects like user experience or readability, they are not essential. This category may also encompass opinions and suggestions that the project team may not necessarily agree with.



Executive Summary

During the audit of the smart contracts provided, 5 critical and 3 high severity vulnerabilities were identified that could compromise the security and integrity of the contracts and funds involved.

A table summarizing the failures found is given below.

Nº	Issue	Severity	Affected asset
1	Reentrancy	CRITICAL	`VulnerableBank.sol`
2	Unencrypted secret information on-chain	CRITICAL	`VulnerableDAO.sol`
3	Weak pseudo-randomness	CRITICAL	`VulnerableDAO.sol`
4	Reimbursement misdirection	CRITICAL	`VulnerableShop.sol`
5	Underflow	CRITICAL	`VulnerableVault.sol`
6	Incorrect access control	MAJOR	`VulnerableBank.sol`
7	Unauthorized sale deletion and blacklisting by any user	MAJOR	`VulnerableShop.sol`
8	Unchecked return value of low-level call	MAJOR	`VulnerableVault.sol`

Detailed Findings

1. Reentrancy

Severity: CRITICAL

In `VulnerableBank.sol` , the `distributeBenefits` function uses the `returnRewards` modifier, where a number of tokens are transferred to the `msg.sender` via the `call` low level function. The `distributeBenefits` function use the Check Effect Interactions pattern but the modifier, which is called at the beginning of the function, does not.

In this way, an attacker can modify his receiving function to perform a reentrancy attack, calling the *distributeBenefits* function again, until there are no tokens left in the contract.

Affected area of code:

-`VulnerableBank.sol` lines 45-53:

```
modifier returnRewards(uint percentage) {
     // A hundreth of the distributed amount will be rewarded to the
distributor as incentive
    uint reward = total_invested * percentage / 10_000;

    (bool success, ) =  payable(msg.sender).call{value: reward}("");
    require(success, "Reward payment failed");

    _;
}
```

Recommendation:

There are different ways to avoid Reentrancy, for example set the global variable `reward` and check that it is 0 at the start of the modifier. Also for proper control you should decrement `reward` from the value of `total_vested`. (See appendix)

2. Unencrypted secret information on-chain

Severity: CRITICAL

In `VulnerableDAO.sol`, the `isAuthorized` modifier makes use of a password to control access to key functions such as `newDispute`. This password is submitted in plain text in the constructor and is treated as such throughout the contract. This makes it accessible to anyone, as soon as the contract has been initialized.

This allows any user to perform privileged actions such as initiate disputes just by checking the password in the storage.

Affected area of code:

-`VulnerableDAO.sol` line 34:

```
// Password to access the key functions
string private password;
```

-`VulnerableDAO.sol` lines 46-61:

```
modifier isAuthorized(string calldata magicWord) {
     require(
           keccak256(
               abi.encodePacked(magicWord)) ==
           keccak256(
               abi.encodePacked(password)),
           "Unauthorized");
        _;

}


constructor(string memory magicWord) {
    password = magicWord;
}
```

Recommendation:

Authorisation should be based on a whitelist of addresses, rather than using passwords, since off-chain encryption has the problem of storing the password hash, which is susceptible to brute force attacks and front running. (See appendix)

3. Weak pseudo-randomness

Severity: CRITICAL

In `VulnerableDAO.sol`, the `lotteryNFT` function calculates a random number to award a user with an NFT, the prize is awarded when the generated number falls below a certain threshold. The vulnerability lies in the use of public variables as a source of randomness.

Giving the opportunity for an attacker to pre-calculate the result and win using a single attempt.

Affected area of code:

-`VulnerableDAO.sol` lines 141-159:

```
function lotteryNFT(address user) internal {
    uint randomNumber = uint8(
        uint256(
            keccak256(
                abi.encodePacked(
                    blockhash(block.number - 1),
                    block.timestamp,
                    user
    ))));

    if (randomNumber < THRESHOLD   ) {
        /*
        * Award NFT logic goes here
        */
    }

    emit AwardNFT(user);
}
```

Recommendation:

Use third-party oracles that generate adequate off-chain randomness. (See appendix)

4. Reimbursement misdirection

Severity: CRITICAL

In `VulnerableShop.sol`, the `reimburse` function is intended to return the value of a dispute to the account that made the dispute. However, it transfers the value to the `owner`, since it is a function that can only be executed by the owner and the transaction is sent to `msg.sender`.

Affected area of code:

-`VulnerableShop.sol` lines 147-156:

```
    function reimburse(address user) external onlyOwner {
        uint amount = disputed_items[user].price;
        /*
        * Reimbursement logic goes here
        */

        // Send the tokens back to the user
        token.safeTransfer(msg.sender, amount);
        emit Reimburse(msg.sender);
    }
```

Recommendation:

Change the transaction recipient to `user`. [(See appendix)]

5. Underflow

Severity: CRITICAL

The `VulnerableVault.sol` smart contract uses a Solidity version that does not have built-in checks to prevent under/overflow issues. As a result, the `enoughStaked` modifier, which is responsible for ensuring that the user has staked enough funds, can be bypassed due to an underflow issue. Specifically, when the `doUnstake` function is invoked, the amount is subtracted from the `balance[msg.sender]`, thereby allowing an attacker to trigger an underflow attack and potentially empty the contract.

Affected area of code:

-`VulnerableVault.sol` lines 31-38

```
modifier enoughStaked(uint amount) {
    require(
        (balance[msg.sender] - lockedFunds[msg.sender] - amount) > 0,
        "Amount cannot be unstaked"
    );

    _;
}
```

-`VulnerableVault.sol` lines 74-83:

```
function doUnstake(uint amount) external enoughStaked(amount) {
    require(amount > 0, "Amount cannot be zero");

    balance[msg.sender] -= amount;

    (bool success, ) = payable(msg.sender).call{value: amount}("");
    require(success, "Unstake failed");

    emit Unstake(msg.sender, amount);
}
```

Recommendation:

Update the version of Solidity or use SafeMath to avoid underflow. (See appendix)

6. Incorrect access control

Severity: MAJOR

In `VulnerableBank.sol,` the custom `onlyOwner` modifier restricts access to transactions initiated by the administrator's address using `tx.origin`. However, this allows an attacker to perform phishing attacks via an intermediate malicious contract. In particular, the privileged functionality to set a new distribution period could be accessed by a potential attacker.

The custom `onlyOwner` modifier is used as access control in sensitive contract functions such as `setDistributedPeriod`. By not using `msg.sender` to check the address that invoked the function, bad access control is being performed, leading to a vulnerability that is easy to exploit via a malicious Smart Contract.

Affected area of code:

-`VulnerableBank.sol` lines 37-40:

```
    modifier onlyOwner() {
        require(tx.origin == admin, "Unauthorized");
        _;
    }
```

Recommendation:

Modify the access control, using `msg.sender` instead of `tx.origin`. (See appendix)

7. Unauthorized sale deletion and blacklisting by any user

Severity: MAJOR

In `VulnerableShop.sol`, the `deleteSale` function is treated as if it were an internal function but has not been implemented as such. This means that any user can delete items for sale and add sellers to the blacklist, effectively causing a denegation of service in the contract.

Affected area of code:

-`VulnerableShop.sol` lines 192-200:

```
    function deleteSale(uint itemID) public {
        Sale memory malicious_sale = offered_items[itemID];

        blacklistedSellers.push(malicious_sale.seller);
        delete offered_items[itemID];

        /*
        * Slashing code goes here
        */
    }
```

Recommendation:

Define the function `deleteSale` as internal. (See appendix)

8. Unchecked return value of low-level call

Severity: MAJOR

In `VulnerableVault.sol`, the `claimRewards` function calls the function of another contract via the `call` method but the return value is not checked. Ignorance of the success of an external call can lead to logical errors and inconsistencies with a potentially high impact.

In this case we expect the function to revert only in case that the user does not have the corresponding privileges but the function may fail for other reasons and we will not know this because we do not check the return value. Resulting in the possibility of getting the reward to an unprivileged user.

Affected area of code:

-`VulnerableVault.sol` lines 90-105:

```
    function claimRewards() external {
        uint amount;

        powerseller_nft.call(
            abi.encodeWithSignature(
                "checkPrivilege(address)",
                msg.sender
            )
        );

        /*
        * Rewards distribution logic goes here
        */

        emit Rewards(msg.sender, amount);
    }
```

Recommendation:

Check the return value. (See appendix)

Appendix

There exist various approaches to address the vulnerabilities identified in the smart contracts under review. The following are some feasible options that can be implemented to mitigate the identified vulnerabilities.

Proposed mitigation of reentrancy

Using the reward variable as a mutex would be enough.

```
// Reward to the distributor
uint reward;

// Transfers a percentage of the vested tokens to the caller as reward
modifier returnRewards(uint percentage) {
    require(reward == 0, "Reentrancy");
    // A tenth of the distributed amount will be rewarded to the
distributor
    reward = total_vested * percentage / 1_000;
    total_vested -= reward;
    (bool success, ) =  payable(msg.sender).call{value: reward}("");
    require(success, "Reward payment failed");
    reward = 0;
    _;
}
```

Back to findings

Proposed mitigation of unencrypted secret information

After eliminating the implementation of passwords, it would be sufficient to set up a whitelist-based authorisation system as follows:

```
contract VulnerableDAO is Ownable{

    mapping(address => bool) public whitelist;

    function addToWhitelist(address[] calldata toAddAddresses)
        external onlyOwner
        {
            for (uint i = 0; i < toAddAddresses.length; i++) {
                whitelist[toAddAddresses[i]] = true;
            }
        }

    function removeFromWhitelist(address[] calldata toRemoveAddresses)
        external onlyOwner
        {
            for (uint i = 0; i < toRemoveAddresses.length; i++) {
                delete whitelist[toRemoveAddresses[i]];
            }
        }

    modifier isAuthorized() {
            require(whitelist[msg.sender], "NOT_IN_WHITELIST");
            _;
        }


      .
      .
      .
}
```

Back to findings

Proposed mitigation of weak pseudo-randomness

We can make use of the BTCRelay oracle that establishes a communication bridge between the Ethereum blockchain and the Bitcoin blockchain. Smart contracts executed in the EVM can request future blocks from the Bitcoin network and use them as a source of entropy.

Back to findings

Proposed mitigation of the logical failure

```
///  The owner can reimburse the price of a disputed sale to the buyer
    ///   then the Sale is marked as "Selling"
    function reimburse(address user) external onlyOwner {
        uint amount = disputed_items[user].price;
        /*
        * Reimbursement logic goes here
        */

        // Send the tokens back to the user
        token.safeTransfer(user, amount);
        emit Reimburse(msg.sender);
    }
```

Back to findings

Proposed mitigation of underflow

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.13;
```

Back to findings

Proposed mitigation of incorrect access control

```
/* Modifiers */
    //  Checks that the caller is the admin
    modifier onlyOwner() {
        require(msg.sender == admin, "Unauthorized");
        _;
    }
```

Back to findings

Proposed mitigation of the logical failure

```
/* Internal */
    function deleteSale(uint itemID) internal{
        Sale memory malicious_sale = offered_items[itemID];

        blacklistedSellers.push(malicious_sale.seller);
        delete offered_items[itemID];

        /*
        * Slashing code goes here
        */
    }
```

Back to findings

Proposed mitigation for vulnerability #8

```
// First checks that the user is expected to receive rewards or not
through the checkPrivilege function
    // that will revert if not. Then calculates the rewards that the
user has earned
    function claimRewards() external {
        uint amount;

        (bool success, ) = powerseller_nft.call(
            abi.encodeWithSignature(
                "checkPrivilege(address)",
                msg.sender
            )
        );
        require(success, “Failure”);

        /*
        * Rewards distribution logic goes here
        */

        emit Rewards(msg.sender, amount);
    }
```

Back to findings