Security audit

Omega

First security audit
Date : July 2nd, 2023

Summary

I. Introduction

1. About Black Paper

Black Paper has been created to help developer teams. Our goal is to help you to make your smart contract safer.
Cybersecurity requires specific expertise which is very different from smart contract development logic. To ensure everything is well fixed, we stay available to help you.

2. Methodology

a. Preparation

This smart contract is an NFT contract for Omega DAO.
A first meeting was held on 26/07/2023. It allows the team to explain the contract workflow, to define the exact scope, and start the audit process.

b. Audit

Before manually auditing, we pass contracts into automatic tools. This allows us to find some easy-to-find vulnerabilities.
Afterward, we manually go deeper. Every variable and function in the scope are analyzed.

You can find many articles on the lesson website. Here is a snippet list of what we test :

- Constructor Mismatch
- Ownership Takeover
- Redundant Fallback Function
- Overflows & Underflows
- Reentrancy
- Money-Giving Bug
- Blackhole
- Unauthorized Self-Destruct
- Revert DoS
- Unchecked External Call
- Gasless Send
- Send Instead Of Transfer
- Costly Loop
- Use Of Untrusted Libraries
- Use Of Predictable Variables
- Transaction Ordering Dependence
- Deprecated Uses

This is not an exhaustive list since we also focus on logic execution exploits, and help optimizing gas price.

c. Reporting

Every point in the code is subject to internal discussions with the team. At this stage, a majority of the probable issues have already been identified and documented.

Post the completion of the code review, analysis, and testing, we prepare a report which contains for each vulnerability :

- Explanation
- Severity score
- How to fix it / Recommendation

Here are severity score definitions.

Critical	A critical vulnerability is a severe issue that can cause significant damage to the contract and its users. These vulnerabilities are easy to exploit and can result in the loss of funds, theft of sensitive data, or other serious consequences. Immediate attention is required to address these vulnerabilities.
Major	A major vulnerability is an issue that can cause significant problems for the contract and its users, but not to the same extent as a critical vulnerability. These vulnerabilities are also easy to exploit and may result in the loss of funds or other negative consequences, but they can be mitigated with timely action.
Medium	A medium vulnerability is an issue that could potentially cause problems for the contract and its users, but the difficulty to exploit is higher than major or critical vulnerabilities. These vulnerabilities may pose a risk to the contract's functionality or security, but they can be addressed without causing significant disruption.
Low	A low vulnerability is a minor issue that does not pose a significant risk to the contract or its users. These vulnerabilities are difficult to exploit and may be cosmetic or technical in nature, but they do not compromise the contract's security or functionality.
Informational	An informational finding is not a vulnerability but rather a suggestion or recommendation for improvement. These findings may include best practices for contract design, suggestions for improving code readability, or other non-critical issues. While not urgent, addressing these findings can help to optimize the contract's performance and reduce the risk of future vulnerabilities.

3. Disclaimer

In this audit, we sent all vulnerabilties found by our team. **We can't guarantee all vulnerabilities have been found.**

4. Scope

The scope of the audit is all smart contracts stored in the "contracts/" folder of Omega Github (commit: b1ba43487eec9c8fbe074a0456fbc368590956dc).

We assume that the following smart contracts don't need to be audited :
- Erc721a contracts
- Chainlink implementation contracts
- Openzeppelin contracts

II. Vulnerabilities

Critical

0 critical severity issues were found.

Major

1 major severity issue was found :

- MAJ-1 Bad generation of metadata URI

Medium

0 medium severity issues were found.

Low

3 low severity issues were found :

- LOW-1 notContract modifier can be bypass
- LOW-2 `publicSalePrice` and `privateSalePrice` need to be changed in the same transaction as `paymentAssetAddress`
- LOW-3 Missing check on `publicSalePrice` and `privateSalePrice`

Informational

0 informational severity issues were found.

Major-1 Bad generation of metadata URI

Impact: Major

Description:

To get the URI of a specific NFT, `getMetadata` function is used.

```
function getMetadata(uint256 tokenId) public view returns (string memory) {
    if (!_exists(tokenId)) revert NonExistentToken();

    if (!isRevealed()) return preRevealURI;

    if (seed == 1) return Strings.toString(tokenId);

    uint256[] memory metadata = new uint256[](maxTotalSupply);

    for (uint256 k; k < maxTotalSupply; k = unchecked_inc(k)) {
        metadata[k] = k;
    }

    for (uint256 i; i < maxTotalSupply; i = unchecked_inc(i)) {
        uint256 j =
            (uint256(keccak256(abi.encode(seed, i))) % (maxTotalSupply));
        (metadata[i], metadata[j]) = (metadata[j], metadata[i]);
    }

    return Strings.toString(metadata[tokenId]);
}
```

Using a random seed, it randomly mixes tokenIds.

However, the mix can be modified whenever by modifying the seed with those 2 functions:

- `resetRequestChainlinkVRF()` and `requestChainlinkVRF()`
- `setSeedManually(randomNumber)`

It can also be modified by calling `setMaxTotalSupply` function which can be done by mistake.

This is an undesired behavior since NFT's metadata should not be mixed after minting (or reveal).

It can cause users loss of trustworthy and Opensea blacklisting if the collection has emitted the `PermanentURI` event.

Moreover, there is no need to change the URI after the reveal if the chainlink call did not fail.

Recommendation:

To fix this, we recommend to

1. Delete `resetRequestChainlinkVRF` function
2. On `setSeedManually` function, add a check that verifies `randomNumber` is not equal to 0 to prevent an error, and add a check on `randomSeedRequest` to prevent override.

```
function setSeedManually(uint256 randomNumber) external onlyAdmin {
    require(randomNumber !=0, "Random number should not be 0");
    require(!randomseedRequested, "Seed is already set");
    randomseedRequested = true;
    seed = randomNumber;
    emit RandomVRFSeedFulfilmentManually();
}
```

3. Don't save the randomNumber if Chainlink call failed.

```
function fulfillRandomness(bytes32 requestId, uint256 randomNumber)
    internal
    override
{
    if (randomNumber > 0) {
        seed = randomNumber;
        emit RandomVRFSeedFulfilmentSuccess(requestId, seed);
    } else {
        randomseedRequested = false;
        emit RandomVRFSeedFulfilmentFail(requestId);
    }
}
```

4. Ensure setMaxTotalSupply can't be modified after the reveal

```
function setMaxTotalSupply(uint256 newMaxTotalSupply) external onlyAdmin {
    require(!isRevealed(), "maxTotalSupply can't be updated after reveal");
    maxTotalSupply = newMaxTotalSupply;
    emit MaxTotalSupplyUpdated(newMaxTotalSupply);
}
```

LOW-1 `notContract` modifier can be bypass

Impact: Low

Description:

On some functions, the `notContract` modifier is used.

```
modifier notContract() {
    /// @dev If the caller is a contract, we revert.
    /// No need to use `_msgSender()` context here.
    if (msg.sender.isContract()) revert ContractIsNotAllowed();
    _;
}
```

Following comments, the goal is to check that `msg.sender` is an EOA.

This modifier is unsafe : a contract in construction could bypass this check, allowing a contract to call this function.
Here is the **link to Openzeppelin warnings**.

Recommendation:

There is no need to add a `notContract` modifier. A contract could be used to call some functions in case of a multisignature wallet for example.

We recommend removing the `notContract` modifier.

LOW-2 `publicSalePrice` and `privateSalePrice` need to be changed in the same transaction as `paymentAssetAddress`

Impact: Low

Description:

In the case of `setPaymentAssetAddress` function call, it can cause an unwanted token price. There is no case where the administrator calls `setPaymentAddress` without modifying the prices in the same transaction. Even for two stablecoins, decimals can be different.

Recommendation:

We recommend setting the new price too when calling `setPaymentAssetAddress` function. It will avoid administrator critical mistakes.

```
    function setPaymentAssetAddress(address newPaymentAssetAddress, uint256 newPublicSalePrice, uint256
newPrivateSalePrice)
        external
        onlyAdmin
    {
        require(newPublicSalePrice != 0, "publicSalePrice can't be equal to 0");
        require(newPrivateSalePrice != 0, "privateSalePrice can't be equal to 0");
        paymentAssetAddress = newPaymentAssetAddress;
        publicSalePrice = newPublicSalePrice;
        privateSalePrice = newPrivateSalePrice;
        emit PublicSalePriceUpdated(newPublicSalePrice);
        emit PrivateSalePriceUpdated(newPrivateSalePrice);
        emit PaymentAssetAddressUpdated(newPaymentAssetAddress);
    }
```

LOW-3 Missing check on `publicSalePrice` and `privateSalePrice`

Impact: Low

Description:

When modifying public or private sale prices, administrator mistakes can be avoided by adding a check on 0 value.

Recommendation:

We recommend adding a check following the code below.

```
function setPublicSalePrice(uint256 newPublicSalePrice)
    external
    onlyAdmin
{
    require(newPublicSalePrice != 0, "publicSalePrice can't be equal to 0");
    publicSalePrice = newPublicSalePrice;
    emit PublicSalePriceUpdated(newPublicSalePrice);
}
```

```
function setPrivateSalePrice(uint256 newPrivateSalePrice)
    external
    onlyAdmin
{
    require(newPrivateSalePrice != 0, "privateSalePrice can't be equal to 0");
    privateSalePrice = newPrivateSalePrice;
    emit PrivateSalePriceUpdated(newPrivateSalePrice);
}
```