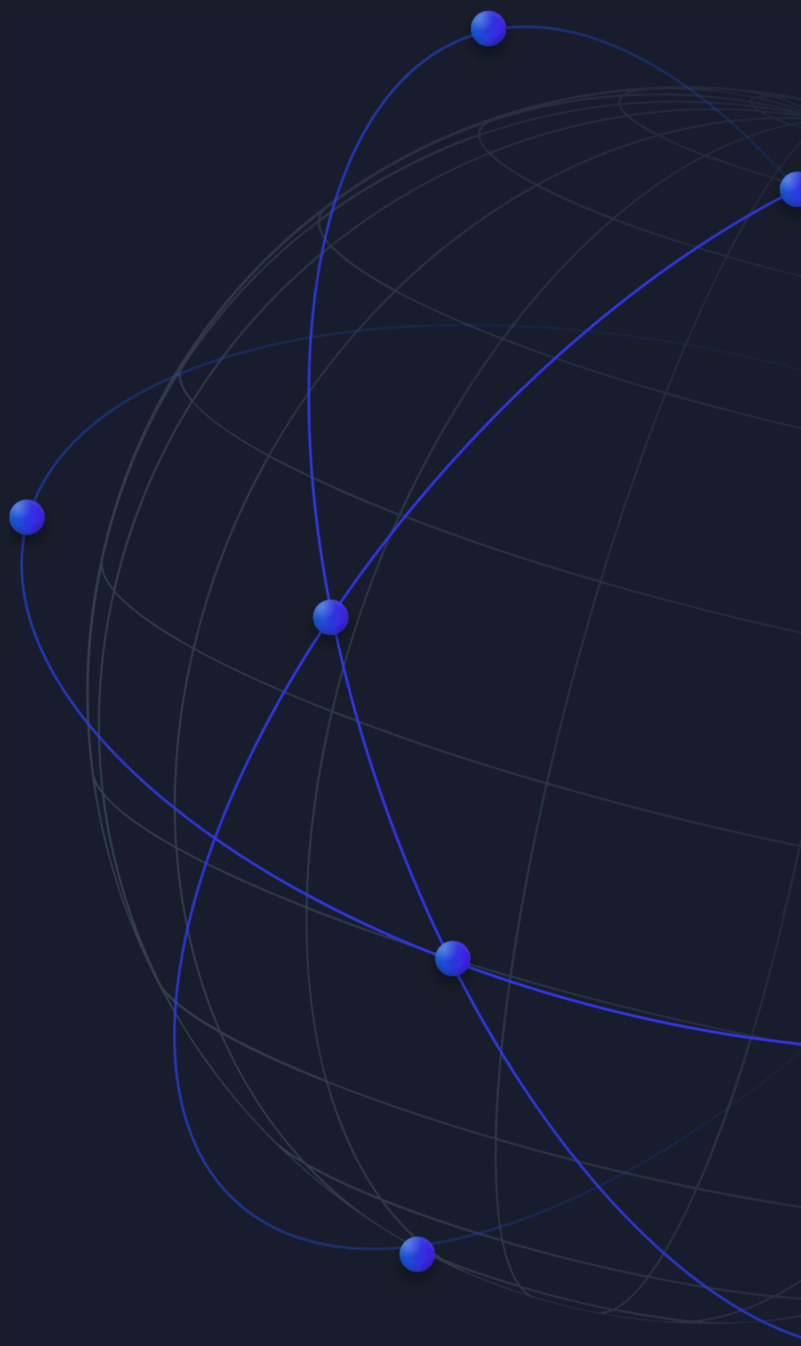
Black paper



Client Name

Security audit

Security audit
[Date]

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

Prior starting the audit process, comprehensive preparation activities were undertaken to ensure a thorough understanding of the Francaise des Jeux smart contracts and to establish clear communication channels with the relevant stakeholders.

1. **Preparation Meeting with Francaise des Jeux Team and Doors3:** On 05/04/2024, a preparatory meeting was conducted with the Francaise des Jeux team and Doors3. The purpose of this meeting was to provide an overview of the entire audit process and to establish expectations. During this session, the audit methodology, timelines, and deliverables were discussed to ensure alignment between all parties involved.

2. **Technical Meeting with Cometh Team:** Subsequently, on 10/04/2024, a technical meeting was held with the Cometh team as an essential component of the preparatory phase. During this session, the Cometh team elaborated on the logic and functionality of the smart contracts under audit. In-depth discussions were conducted to clarify the roles

designated within the contracts, particularly focusing on the `DEFAULT_ADMIN_ROLE` and `MINTER_ROLE`. Additionally, questions regarding the trustworthiness of these roles and the security measures employed to safeguard their associated private keys were addressed comprehensively.

3. **Audit Initiation:** Armed with a thorough understanding of the smart contracts' functionality and the necessary insights gained from the preparatory meetings, the audit officially commenced. All information gathered during the preparatory phase was utilized to inform the audit process, ensuring a comprehensive evaluation of the contracts' security posture.

Through these preparatory activities, a strong foundation was established for conducting a rigorous and effective audit, with a focus on maximizing security and mitigating potential risks within the Francaise des Jeux smart contracts.

b. Review

The code review process constituted a fundamental component of the audit, involving a detailed examination of the solidity codebase to ensure adherence to security best practices, mitigate potential vulnerabilities, and enhance overall code quality. Below is a comprehensive overview of the methodologies employed and considerations addressed during the code review:

1. **Security Vulnerability Assessment:**

   The primary objective of the code review was to identify and mitigate security vulnerabilities that could potentially compromise the integrity and functionality of the smart contracts. This involved a meticulous examination of the code for common vulnerabilities such as reentrancy, arithmetic overflow/underflow, improper access control, and unchecked external calls.

2. **Reentrancy Mitigation:**

   Reentrancy vulnerabilities, which allow malicious actors to manipulate contract state by repeatedly re-entering a function before its execution is complete, were thoroughly scrutinized. Critical functions such as token transfers and state modifications were assessed to ensure that appropriate checks and safeguards are in place to mitigate the risk of reentrancy attacks.

3. **Access Control Logic:**

   Access control mechanisms, particularly role-based access control (RBAC), were carefully reviewed to verify that permissions are correctly assigned and enforced. Special attention was given to roles such as `DEFAULT_ADMIN_ROLE` and `MINTER_ROLE` to ensure that only authorized entities have access to sensitive functions and operations.

4. **Input Validation and Sanitization:**

   Input validation and sanitization were integral to the code review process to prevent potential exploits arising from malicious or unexpected user input. Parameters passed to functions were scrutinized to ensure that they are properly validated and sanitized to mitigate the risk of input-based vulnerabilities such as buffer overflows and injection attacks.

5. **Gas Optimization and Efficiency:**

   Gas optimization techniques were evaluated to ensure that the smart contracts are efficiently designed and minimize transaction costs for users. Code segments were analyzed to identify opportunities for optimization, including reducing redundant computations, minimizing storage usage, and optimizing loop iterations.

6. **Code Readability and Documentation:**

   The clarity and readability of the codebase were assessed to facilitate ease of understanding and maintenance. Clear and descriptive variable names, well-structured functions, and comprehensive comments/documentation were considered essential for enhancing code readability and ensuring that the logic and purpose of the code are easily discernible.

7. **Best Practices Adherence:**

   Adherence to industry best practices and standards, as outlined in the Ethereum Solidity Style Guide and other relevant guidelines, was verified throughout the code review process. Conformance to established best practices ensures code consistency, reduces the likelihood of errors, and promotes overall code quality.

8. **External Dependencies and Interactions:**

Interactions with external contracts and dependencies were scrutinized to assess their security implications and potential impact on the smart contracts' integrity and functionality. Any external calls or dependencies were thoroughly vetted to ensure that they do not introduce vulnerabilities or expose the contracts to potential exploits.

By conducting a comprehensive code review encompassing these considerations and methodologies, the audit aimed to provide with actionable insights and recommendations to enhance the security and reliability of their smart contracts. If you have any further inquiries or require additional details regarding the code review process, please feel free to reach out.

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. For each vulnerability, it contains the following informations.

- Description
- Severity
- 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 following smart contracts are considered in scope: - UltimateNumbers.sol - UltimateNumbersMinter.sol

Other libraries and smart contracts are considered safe, only there implementation is audited.

II. Vulnerabilities

CRIT-1 Unsafe safeTransferFrom function call

Impact: Critical

Description:

The existing `_mint` function in the smart contract is susceptible to exploitation if a user has approved an unlimited allowance to this contract. This vulnerability allows anyone to force the user to mint tokens by utilizing the remaining allowance. To mitigate this risk, it's necessary to modify the minting mechanism.

Recommendation:

To address the vulnerability and prevent potential exploitation of allowance, it's recommended to modify the `mint` function to only allow minting for the caller (`msg.sender`). This change ensures that minting can only be initiated by the user who intends to receive the tokens, thereby mitigating the risk of allowance exploitation.

Here's the recommended modification to the `mint` function:

```
/// Mint with a previously set allowance
/// @dev `proof` is not required to mint when public sales have begun
function mint(
    uint256 packId,
    uint256 quantity,
    uint256 allowedQuantity,
    bytes32[] calldata proof
) external {
    _mint(packId, quantity, allowedQuantity, msg.sender, proof, true);
}
```

With this modification, the `mint` function restricts minting to the caller (`msg.sender`), ensuring that only the intended recipient initiates the minting process.

To allow minting for another user while maintaining security, utilize the `mintWithPermit` function. Ensure that the `mintWithPermit` function incorporates the recommended signature mechanism from CRIT-XXX to validate minting requests.

As we consider MINTER_ROLE address safe, `mintFromFiat` function does not need to be modified.

CRIT-2 Front-Running in mintWithPermit function (Parameter manipulation)

Impact: Critical

Description:

The `mintWithPermit` function in the smart contract is vulnerable to front-running attacks where an attacker can manipulate the function parameters by intercepting and modifying the transaction before it gets confirmed on the blockchain. This vulnerability allows the attacker to change the values of `packId`, `quantity`, and `allowedQuantity`, potentially altering the intended minting behavior.

Recommendation:

To mitigate the front-running vulnerability and prevent parameter manipulation, it's recommended to implement a signature mechanism that signs all function arguments, including `packId`, `quantity`, `allowedQuantity`, and other relevant parameters. Subsequently, within the `mintWithPermit` function, validate that this signature is signed by the intended recipient address. If the signature verification fails, revert the transaction to prevent unauthorized minting attempts.

```
import {SignatureChecker} from "@openzeppelin/contracts/utils/cryptography/SignatureChecker.sol";
import {EIP712} from "@openzeppelin/contracts/utils/cryptography/EIP712.sol";
import {Nonces} from "@openzeppelin/contracts/utils/Nonces.sol";
```

```
contract UltimateNumbersMinter is AccessControl, EIP712, Nonces {
```

```
constructor(address admin, IERC20 _purchaseToken, UltimateNumbers _nft, address _tokenRecipient)
EIP712("UltimateNumbersMinter", "1.0") {
```

```
bytes32 private constant MINT_WITH_PERMIT_TYPEHASH =
    keccak256("mintWithPermit("
    "uint256 packId,"
    "uint256 quantity,"
    "uint256 allowedQuantity,"
    "address recipient,"
    "uint256 deadline,"
    "uint8 v,"
    "bytes32 r,"
    "bytes32 s,"
    "bytes32[] calldata proof,"
    "bytes memory signature)");
```

```
    /// Mint via `Permit` feature bypassing need for upfront allowance
    /// @dev `proof` is not required to mint when public sales have begun
    function mintWithPermit(
        uint256 packId,
        uint256 quantity,
        uint256 allowedQuantity,
        address recipient,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s,
        bytes32[] calldata proof,
        bytes memory signature
    ) external {
        // Check signature
        bytes32 hash = _hashTypedDataV4(keccak256(abi.encode(
            MINT_WITH_PERMIT_TYPEHASH,
            packId,
            quantity,
            allowedQuantity,
            recipient,
            deadline,
            v,
            r,
            s,
            proof,
            _useNonce(recipient))));
        SignatureChecker.isValidSignatureNow(recipient, hash, signature);

        uint256 price = _getPackPrice(packId) * quantity;
        IERC20Permit(address(purchaseToken)).permit(recipient, address(this), price, deadline, v,
r, s);

        _mint(packId, quantity, allowedQuantity, recipient, proof, true);
    }
```

This approach ensures that only minting requests signed by the recipient address are accepted, mitigating the risk of front-running attacks and parameter manipulation.

Note that `SignatureChecker` also works with smart contract signers using ERC-1271.

The constructor and the contracts inheritance also needs to be modified

CRIT-3 Reentrancy vulnerability in mint function

Impact: Critical

Description:

The `mint` function in the smart contract contains a reentrancy vulnerability. This vulnerability arises from the possibility of reentrant calls to the `_safeMint` function within the loop, allowing a malicious user to manipulate the contract's state unexpectedly.

In this function, a user could easily create a malicious `onERC721Received` function allowing to mint more ERC721 tokens than `MAX_SUPPLY`.

Recommendation:

To mitigate the reentrancy vulnerability, we recommend to use OpenZeppelin's ReentrancyGuard library, which provides a robust and battle-tested solution for preventing reentrancy attacks.

Here's how you can integrate the ReentrancyGuard library into your contract and modify the `mint` function to utilize the guard:

````
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
````

```solidity
function mint(uint256 quantity, address receiver) public onlyRole(MINTER_ROLE) nonReentrant {
    require(totalSupply() + quantity <= MAX_SUPPLY, 'Max Supply Hit');
    for (uint256 i = 0; i < quantity; i++) {
        _safeMint(receiver, ++_nextTokenId);
    }
}
```

Moreover, to completely prevent it, the `nonReentrant` modifier should be added to the `mint(address receiver)` function.

The `nonReentrant` modifier ensures that the function is not called recursively from within the same call stack, effectively preventing reentrant calls to the `mint` function.

MAJ-1 Front-running mintWithPermit function causing transaction to revert

Impact: Major

Description:

The `mintWithPermit` function in the smart contract is vulnerable to front-running attacks. A malicious user can exploit this vulnerability by directly calling the `permit` function on the `purchaseToken` smart contract before the `mintWithPermit` function is executed, causing the `mintWithPermit` function to revert.

Recommendation:

To mitigate the front-running vulnerability, it's recommended to add error handling and fallback mechanisms within the `mintWithPermit` function. Specifically, the function should attempt to execute the `permit` function and handle potential revert errors gracefully. If the `permit` function call fails, the contract should directly check the allowance to determine if it's sufficient for the mint operation.

Here's the recommended modification to the `mintWithPermit` function:

```
function mintWithPermit(
    uint256 packId,
    uint256 quantity,
    uint256 allowedQuantity,
    address recipient,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s,
    bytes32[] calldata proof
) external {
    uint256 price = _getPackPrice(packId) * quantity;
    try IERC20Permit(address(purchaseToken)).permit(recipient, address(this), price, deadline, v,
r, s) {
        // Permit succeeded, continue with minting
        _mint(packId, quantity, allowedQuantity, recipient, proof, true);
    } catch {
        // Permit failed, check allowance directly
        require(purchaseToken.allowance(recipient, address(this)) >= price, "Insufficient
allowance");

        // Allowance is sufficient, continue with minting
        _mint(packId, quantity, allowedQuantity, recipient, proof, true);
    }
}
```

With this modification, the `mintWithPermit` function first attempts to execute the `permit` function using a `try` block. If the `permit` function call fails (i.e., reverts), the contract falls back to directly checking the allowance of the `purchaseToken` to ensure it's sufficient for the mint operation.

LOW-1 Unused mint function

Impact: Low

Description:

The `mint(address receiver)` function in the UltimateNumbers contract is not utilized by the UltimateNumbersMinter contract.

As the UltimateNumbersMinter is the only smart contract allowed to mint tokens, it remains dormant and serves no purpose within the contract's functionality.

Recommendation:

To streamline the contract code and improve clarity, we recommend to remove the unused `mint(address receiver)` function entirely. By removing unused code, you reduce the attack surface and potential points of confusion for developers interacting with the contract.

INF-1 Avoiding pack override in createPack function

Impact: Informational

Description:

The `createPack` function in the smart contract currently allows for the override of existing packs, which can lead to unintended behavior, such as allowing users to mint with a pack they did not have access to when the whitelist was set. Although errors in pack creation are typically restricted to the `DEFAULT_ADMIN_ROLE`, it's advisable to avoid pack override to maintain the integrity of the whitelist and prevent potential issues.

Recommendation:

To prevent the override of existing packs and ensure the integrity of the whitelist, it's recommended to implement a check within the `createPack` function to verify whether the pack already exists before creating a new one. If the pack already exists, the function should revert to prevent the override.

Here's the modified implementation of the `createPack` function with the added check:

```
function createPack(uint256 packId, PurchasePack calldata pack) external
onlyRole(DEFAULT_ADMIN_ROLE) {
    require(packs[packId].quantity== 0, "Pack already exists");
    packs[packId] = pack;
}
```

With this modification, the `createPack` function checks whether a pack with the specified `packId` already exists. If an existing pack is found, the function reverts, preventing the override of existing packs and ensuring the integrity of the whitelist.

Incorporating this check enhances the security and robustness of the smart contract, reducing the likelihood of unintended behavior or errors in pack creation.

INF-2 Missing null validation in setWhitelistRoot function

Impact: Informational

Description:

The `setWhitelistRoot` function in the smart contract lacks a validation check to ensure that the `_root` parameter is not set to null (`bytes32(0)`). This oversight leaves the contract vulnerable to potential issues arising from assigning a null value as the whitelist root, which could lead to unexpected behavior or vulnerabilities in the contract's functionality.

Recommendation:

To mitigate the risk associated with assigning a null value as the whitelist root, it's recommended to add a validation check within the `setWhitelistRoot` function. This check should verify that the `_root` parameter is not equal to null before updating the `whitelistRoot` variable.

Here's the modified implementation of the `setWhitelistRoot` function with the added null validation:

```
function setWhitelistRoot(bytes32 _root) external onlyRole(DEFAULT_ADMIN_ROLE) {
    require(_root != bytes32(0), "Invalid root value");
    bytes32 old = whitelistRoot;
    whitelistRoot = _root;
    emit SetWhitelistRoot(old, _root);
}
```

With this modification, the `setWhitelistRoot` function ensures that only non-null values can be assigned as the whitelist root, thereby mitigating potential risks associated with assigning a null value.

INF-3 Missing address validation in setTokenRecipient function

Impact: Informational

Description:

The `setTokenRecipient` function in the smart contract lacks a validation check to ensure that the `newRecipient` address is not set to address(0) (the zero address) by mistake. This oversight leaves the contract vulnerable to potential issues arising from assigning the zero address as the token recipient, which could lead to tokens lost.

Recommendation:

To mitigate the risk associated with assigning the zero address as the token recipient, it's recommended to add a validation check within the `setTokenRecipient` function. This check should verify that the `newRecipient` address is not equal to address(0) before updating the `tokenRecipient` variable.

Here's the modified implementation of the `setTokenRecipient` function with the added address validation:

```
function setTokenRecipient(address newRecipient) external onlyRole(DEFAULT_ADMIN_ROLE) {
    require(newRecipient != address(0), "Invalid recipient address");
    tokenRecipient = newRecipient;
    emit SetTokenRecipient(newRecipient);
}
```

With this modification, the `setTokenRecipient` function ensures that only non-zero addresses can be assigned as the token recipient.