QuillAudits

Audit Report February, 2024

For

QUANTELICA

Table of Content

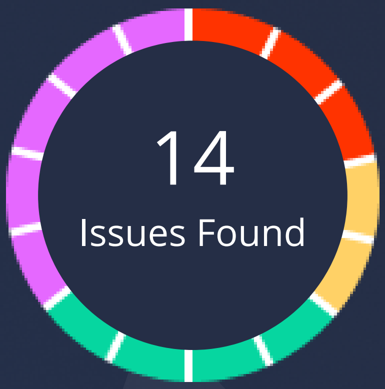
Table of Content

Executive Summary

Project Name	Quantelica
Project URL	*https://quantelica.com/*
Overview	Quantelica is a gaming ecosystem tailored to the needs of Web3 gaming. The platform is designed to integrate third-party games, provide cutting-edge software development kits (SDKs), and offer unique NFT capabilities. It will feature a marketplace, a community, unique NFT collections, play-to-earn games, and incentives for players. Audited Contracts Overview:- 1]Contract PioneerCollection: allows users to mint NFTs from different subcollections, each with its own unique properties and characteristics. Supports GSN and Chainlink VRF. 2]Contract PartnershipProcessor: Implements the logic for processing partnershipoperations, including discount application, commission distribution to various participants (partners, shareholders), and management of DApp project settings.
Audit Scope	*https://git.quantelica.com/blockchain/audit-271122*
Contracts in Scope	Branch: Main Contracts:- -PioneersCollection.sol -PartnershipProcessor.sol and interfaces
Commit Hash	7eba366
Language	Solidity
Blockchain	Ethereum
Method	Manual Review, Automated Tools, Functional Testing

Review 1	11th January 2024 - 22nd January 2024
Updated Code Received	8th February 2024
Review 2	8th February 2024 - 9th February 2024
Fixed In	f45b4346e82b9441997687d9147c008a898ef7ac

Number of Security Issues per Severity



🟥 High 🟨 Medium

🟩 Low 🟪 Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	0	0	0
Partially Resolved Issues	0	0	0	0
Resolved Issues	**3**	**2**	**4**	**5**

Checked Vulnerabilities

- ✓ Access Management
- ✓ Arbitrary write to storage
- ✓ Centralization of control
- ✓ Ether theft
- ✓ Improper or missing events
- ✓ Logical issues and flaws
- ✓ Arithmetic Correctness
- ✓ Race conditions/front running
- ✓ SWC Registry
- ✓ Re-entrancy
- ✓ Timestamp Dependence
- ✓ Gas Limit and Loops
- ✓ Exception Disorder
- ✓ Gasless Send
- ✓ Use of tx.origin
- ✓ Malicious libraries
- ✓ Compiler version not fixed
- ✓ Address hardcoded
- ✓ Divide before multiply
- ✓ Integer overflow/underflow
- ✓ ERC's conformance
- ✓ Dangerous strict equalities
- ✓ Tautology or contradiction
- ✓ Return values of low-level calls
- ✓ Missing Zero Address Validation
- ✓ Private modifier
- ✓ Revert/require functions
- ✓ Multiple Sends
- ✓ Using suicide
- ✓ Using delegatecall
- ✓ Upgradeable safety
- ✓ Using throw

Checked Vulnerabilities

- ✓ Using inline assembly
- ✓ Style guide violation
- ✓ Unsafe type inference
- ✓ Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments, match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Remix IDE, Truffle, Solhint, Mythril, Slither, Solidity Static Analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

A. Contract - PioneersCollection.sol

High Severity Issues

A.1 Lack of Access Control in **fulfillRandom** Words Function

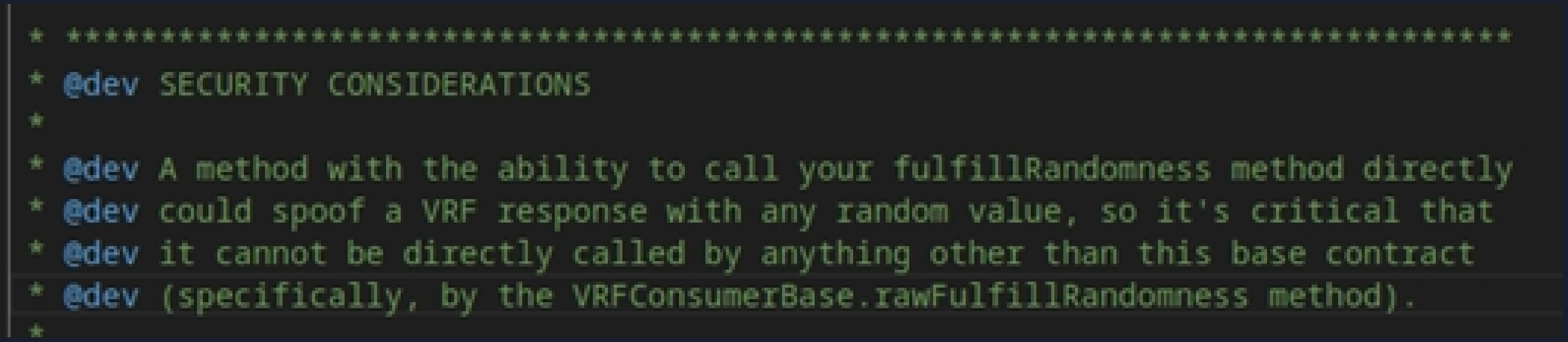
Line

658-664

Function - fulfillRandomWords

```
function fulfillRandomWords(
    uint256 requestId,
    uint256[] memory randomWords
) internal override {
    MintRequest memory request = _vrfRequests[requestId];
    _mint(request, randomWords[0], requestId);
}
```

POC



```
* ****************************************************************************
* @dev SECURITY CONSIDERATIONS
*
* @dev A method with the ability to call your fulfillRandomness method directly
* @dev could spoof a VRF response with any random value, so it's critical that
* @dev it cannot be directly called by anything other than this base contract
* @dev (specifically, by the VRFConsumerBase.rawFulfillRandomness method).
*
```

https://github.com/smartcontractkit/chainlink/blob/e4bde648582d55806ab7e0f8d4ea05a721ca120d/contracts/src/v0.8/vrf/VRFConsumerBaseV2.sol#L66

Description

The **fulfillRandomWords** function lacks proper access control, allowing any external entity to call this internal function. As a result, there is a critical security vulnerability where an unauthorized entity could potentially spoof a VRF response with arbitrary random values. It is crucial to restrict the invocation of this function to only trusted entities, specifically the **VRFConsumerBase.rawFulfillRandomness** method, to ensure the integrity of the VRF process.

Remediation

To address this issue, implement proper access control in the **fulfillRandomWords** function to ensure that only the intended and trusted entities, specifically the **VRFConsumerBase.rawFulfillRandomness** method, can invoke it.

Status

Resolved

A.2 Incorrect token burn during subcollection removal

Line

181-189

Function - removeSubcollection

```
if(subcollections[subcollectionIdx].totalMinted > 0){
            uint256 startIndex = subcollections[subcollectionIdx].startIndex;
            uint256 endIndex = subcollections[subcollectionIdx].endIndex;
            for (uint256 i = startIndex; i <= endIndex; i++) {
            if (_exists(i)) {
                        _burn(i);
            }
        }
  }
```

Description

The contract has a critical logic flaw in the **removeSubcollection** function, where tokens from other subcollections can be mistakenly burned. The iteration over token IDs for removal does consider that token IDs are the startIndex and endIndex of a subCollection but that's not the case for any subcollection. As a result, this could lead to the unintended burning of NFTs that do not belong to the subcollection being removed, causing a severe impact on token ownership and integrity of the protocol.

Remediation

To address this issue, modify the logic of **removeSubcollection** function to ensure that only tokens belonging to the subcollection being removed are burned.

Status

Resolved

A.3 Lack of blacklist check in batch transfer function

Line

312-319

Function - batchTransfer and mintGoldenToken

```
function batchTransfer(address to, uint256[] memory tokenIds) external {
    for (uint256 i = 0; i < tokenIds.length; i++) {
        uint256 tokenId = tokenIds[i];
        _requireMinted(tokenId);
        require(_isApprovedOrOwner(_msgSender(), tokenId), "Not token owner nor
approved");
        safeTransferFrom(_msgSender(), to, tokenId, "");
    }
}
```

and

```
function mintGoldenToken(
    string calldata subcollectionName,
    uint256[] calldata goldenTicketNumbers,
    address to
) external onlyRole(CLAIMER_ROLE) {
    for (uint i = 0; i < goldenTicketNumbers.length; i++) {
        uint256 subcollectionIdx = getSubcollectionIdxByName(subcollectionName);
        require(
            goldenTicketNumbers[i] <= subcollections[subcollectionIdx].params.goldSupply &&
                goldenTicketNumbers[i] > 0,
            "Invalid token ID"
        );
        uint256 endIndex = subcollections[subcollectionIdx].endIndex;
        uint256 tokenId = endIndex + goldenTicketNumbers[i];
        _safeMint(to, tokenId);
        _subIdByTokenId[tokenId] = subcollectionIdx;
    }
}
```

Description

The contract exhibits a potential security risk due to a missing check for whether the destination address (**to**) is blacklisted or not in the **batchTransfer** and **mintGoldenToken** function. Without proper validation, blacklisted addresses can potentially receive tokens through batch transfers, leading to a bypass of the blacklist mechanism. This oversight poses a security vulnerability, as it allows tokens to be transferred to blacklisted addresses, contrary to the intended restrictions.

Remediation

To address this issue, include a check to verify whether the destination address (**to**) is blacklisted before proceeding with the **batchTransfer** and **mintGoldenToken** functions. It's good to implement the same on **withdraw** function.

Status

Resolved

Medium Severity Issues

A.4 Inability to delete TokenCID due to private set function

Line

735-738

Function - burn & _setTokenCID

```
function burn(uint256 tokenId) external {
    // ...
    if (bytes(_tokenCIDs[tokenId]).length != 0) {
        delete _tokenCIDs[tokenId];
    }
}

function _setTokenCID(uint256 tokenId, string calldata tokenCID) private {
    require(_exists(tokenId), "CID set of nonexistent token");
    _tokenCIDs[tokenId] = tokenCID;
}
```

Description

The contract has an issue where the **burn** function attempts to delete a tokenCID, but the corresponding set function **_setTokenCID** is marked as private and cannot be directly called by the **burn** function. As a result, there is no mechanism in place to set or update the tokenCID, making it impossible to delete the CID within the **burn** function. This limitation hinders the contract's functionality and could lead to undesired consequences.

Remediation

To address this issue, consider adjusting the visibility of the **_setTokenCID** function or providing a public function that allows updating or deleting the tokenCID. If the function needs to remain private, create a controlled and secure way to call it within the contract.

Status

Resolved

A.5 Missing check for token existence in **burn** function

Line

370-380

Function - burn

```
function burn(uint256 tokenId) external {
    uint256 subcollectionIdx = getSubcollectionIdxByTokenId(tokenId);
    bool burnEnabled = subcollections[subcollectionIdx].burnEnabled;
    require(_isApprovedOrOwner(_msgSender(), tokenId), "Not token owner nor approved");
    require(burnEnabled, "Burn disabled");
    _burn(tokenId);
    delete _subIdByTokenId[tokenId];
    if (bytes(_tokenCIDs[tokenId]).length != 0) {
        delete _tokenCIDs[tokenId];
    }
}
```

Description

The contract has a high-severity issue in the burn function where it fails to check whether the specified token exists or not before attempting to burn it. This absence of a pre-check on token existence creates a critical vulnerability, as attempting to burn a nonexistent token could lead to unexpected behavior, compromising the integrity of the contract and causing potential disruptions in token ownership.

Remediation

To address this issue and ensure the proper synchronization of LP tokens and shares, you should include a redemption status check before allowing users to deposit assets.

Status

Resolved

Low Severity Issues

A.6 Missing Zero checks

Line **Function - transferFee**

207
```
function mint(
        string calldata subcollectionName,
        address to,
        bytes32[] memory proof,
        uint256 mintAmount,
        uint256 initialPrice
        ) external payable {

// proof, mintAmount and initialPrice is not checked.//
}
```

403
```
function setMaxMintPerAddress(
        string calldata subcollectionName,
        uint16 maxMintPerAddress
        ) external onlyRole(TRUSTED_SIGNER_ROLE) {
        ...
        }
```

417
```
function setTrustedForwarder(
        address forwarder
        ) external onlyRole(TRUSTED_SIGNER_ROLE) {
        // ... //
        }
```

Line	Function - transferFee
467	function setBlacklist(address adrs, bool ban) external onlyRole(TRUSTED_SIGNER_ROLE) { // .... // }

```
function setBlacklist(
        address adrs,
        bool ban
        ) external onlyRole(TRUSTED_SIGNER_ROLE) {
        // .... //
        }
```

Description

Several critical functions within the contract lack validation checks to ensure that zero addresses are not accepted as input parameters. This oversight can lead to unintended behavior, security vulnerabilities, and potential exploitation if malicious actors pass zero addresses as arguments to these functions.

Remediation

To address this issue and enhance the security of the contract, you should implement zero-address validation checks in critical functions. Here's a recommended remediation approach:

1. **Require Non-Zero Address:** Add a validation check at the beginning of each critical function to ensure that the input addresses are non-zero. If an input address is zero, the function should revert with an error message.
2. **Input Validation:** Validate the parameters provided to the functions, especially those involving asset transfers or administrative actions, to prevent potential vulnerabilities or loss of assets.

Status

Resolved

A.7 State-changing methods are missing event emissions

Functions

Setter function PioneersCollection.setSubscriptionId(uint64) (contracts/PioneersCollection.sol#386-390) does not emit an event

Setter function PioneersCollection.setKeyHash(bytes32) (contracts/PioneersCollection.sol#396-400) does not emit an event

Setter function PioneersCollection.setSubcollectionLocked(string) (contracts/PioneersCollection.sol#490-493) does not emit an event

Setter function PioneersCollection.setAvailableMainSupply(string,uint256) (contracts/PioneersCollection.sol#522-528) does not emit an event

Setter function PioneersCollection.setRoyaltyInfo(address,uint96) (contracts/PioneersCollection.sol#558-563) does not emit an event

Setter function ReferralProcessor.setProject(address,uint256,uint256,uint256[],uint256,bool) (contracts/ReferralProcessor.sol#125-143) does not emit an event

Description

Critical functions within the contract lack the emission of events. Events play a vital role in providing transparency, enabling external systems to react to changes, and offering a way to track important contract activities. The absence of events can hinder monitoring and auditing efforts, making it difficult to detect and respond to critical contract actions.

Remediation

To address this issue and improve the transparency and auditability of the contract, you should add appropriate event emissions in critical functions. These events should capture essential information about the function's execution, including input parameters and outcomes. Additionally, consider emitting events both before and after critical state changes, where applicable.

Status

Resolved

A.8 Unused codes

Line	Function - transferFee
28	using Strings for uint256;
42	bool private _blacklistEnabled;
69	// Random values received from Chainlink to shift on. mapping(uint256 => uint256) private _randomOffsets;
73	// User's nonce mapping(address => uint256) public nonces;
77	event BatchMetadataUpdate(uint256 indexed _fromTokenId, uint256 indexed _toTokenId);

Description

The contract contains sections of code that are defined but not referenced or executed during the contract's operation. This unused code might include functions, variables, or entire code blocks that serve no functional purpose or are not part of the contract's intended behavior. Unused code can increase the contract's complexity, potentially introducing confusion for developers and auditors and leading to unnecessary gas consumption.

Remediation

To address this issue, carefully review the contract code and remove any sections that are not actively used or required for the contract's functionality. This helps streamline the codebase, making it more concise and easier to manage. If there are functions or variables that were intended for future use or testing, consider either implementing them or commenting on their purpose to provide clarity.

Status

Resolved

A.9 Checks that are must to implement

Function - N/A

```
==================== PioneersCollection.sol ====================

function addSubcollection(
SubcollectionParams calldata params
) external onlyRole(TRUSTED_SIGNER_ROLE) {
...
// no check on gold supply here.
// good to have a check that the availableMainSupply <= mainSupply  &&
availableMainSupply > 0 while adding an subcollection.
// Merkle root is also not checked.
// Check can be added that availableMainSupply <= the mainSupply of the subcollection.
}
 Not enough contract space

function removeSubcollection(string calldata subcollectionName) external
onlyRole(DEFAULT_ADMIN_ROLE) {
...
```

```
// not checking if burning is enabled or not for that subcollection.
...
// below things as well are not followed which are followed during burning an NFT.
// delete _subIdByTokenId[tokenId];
// if (bytes(_tokenCIDs[tokenId]).length != 0) {
// delete _tokenCIDs[tokenId];
...
}
No need

function mint(
        string calldata subcollectionName,
        address to,
        bytes32[] memory proof,
        uint256 mintAmount,   // user will be having control over it and can send any
amount. Not possible to implement
        uint256 initialPrice // user will be having control over it and can send any number.
Not possible to implement
        ) external payable {
...
bool onWhitelist = _verifyProof(subcollectionName,
keccak256(bytes.concat(keccak256(abi.encode(_msgSender(), initialValue /
mintAmount)))),proof);
// take subcollectionName from the struct subcollection.params.name instead of user
passed input.
}

function mintGoldenToken(
        string calldata subcollectionName,
        uint256[] calldata goldenTicketNumbers,
        address to
        ) external onlyRole(CLAIMER_ROLE) {
...
// there is no check on the total Minted Golden tokens and how many can be minted for an
user.
}

function withdraw(
        address payable recipient
        ) external onlyRole(TRUSTED_SIGNER_ROLE) {
....
```

```
// the balcklist check can be implemented here as well for the recipient address.
}

function setAvailableMainSupply(
            string calldata subcollectionName,
            uint256 availableMainSupply
            ) external onlyRole(TRUSTED_SIGNER_ROLE) {
...
// Check can be added that it availableMainSupply <= the mainSupply of the subcollection.
}

function fulfillRandomWords(
            uint256 requestId,
            uint256[] memory randomWords  because:
            ) internal override {
....
// Have an check if the requestId even exists or not.
// instead of passing directly it is good to be stored first because of below reason.
// https://docs.chain.link/vrf/v2/security#fulfillrandomwords-must-not-revert

==================== ReferralProcessor.sol ====================

function distributeFunds(
            address referee,
            uint256 subIndex
            ) external payable nonReentrant onlyRole(TRUSTED_DAPP_ROLE) {
...
// This function call is not there in the Pioneers collection contract.
}
```

Description

Several functions within the contract lack necessary checks and verifications, potentially leading to unintended behaviours or vulnerabilities. These missing checks can impact the contract's security, correctness, and adherence to intended rules. It's crucial to incorporate these checks to ensure proper functionality and prevent unauthorized or erroneous actions.

Remediation

Implement the above-mentioned checks and verifications in the respective functions.

Status

Resolved

Informational Issues

A.10 Vulnerable Merkle Proof Library Version

Line

11

Function - Import

import "@openzeppelin/contracts/utils/cryptography/MerkleProof.sol";
// OpenZeppelin Contracts (last updated v4.8.0)

Description

Affected versions of this package are vulnerable to Improper Input Validation.

Remediation

Upgrade **@openzeppelin/contracts** to version 4.9.2 or higher.
https://security.snyk.io/vuln/SNYK-JS-OPENZEPPELINCONTRACTS-5711902

Status

Resolved

A.11: Unlocked pragma (pragma solidity ^0.8.19)

Description

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Remediation

Here all the in-scope contracts have an unlocked pragma, it is recommended to lock the same. Moreover, we strongly suggest not to use experimental Solidity features (e.g., pragma experimental ABIEncoderV2) or third-party unaudited libraries. If necessary, refactor the current code base to only use stable features.

Status

Resolved

A.12: Consider using custom errors

Description

Custom errors reduce the contract size and can provide easier integration with a protocol. Consider using those instead of require statements with string error.

Status

Resolved

A.13: Change function visibility from public to external

Description

LM Internal Functions(except isValidDepositAmount() function) in LoanManager.sol are never called from within contracts but yet declared public. Their visibility can be made external to save gas.

Status

Resolved

A.14 General Recommendation

In the case of stable coins, the hierarchy of authority plays a crucial role and our team recommends that the roles with privileges must be given to the accounts with proven authority and functions like “Minting” and “Burning” must be used cautiously because once these functions are called on the mainnet then could be no minting or burning. Hence, it will be an irreversible change.

Status

Resolved

Functional Tests

Some of the tests performed are mentioned below:

PioneersCollection:

- ✓ Should allow the owner to update referral fees
- ✓ Should revert if a non-owner tries to update referral fees
- ✓ Should handle referral fees with discount correctly

Collection Modification Access:

- ✓ Should revert if an unauthorized address tries to add a sub-collection
- ✓ Should show the correct URI after the mint
- ✓ Should not mint without proof, when whitelist is enabled
- ✓ Should mint with pseudo-random ID assignment
- ✓ Should disable whitelist for minting and mint token without proof
- ✓ Should sell all tokens

Access Control Tests:

- ✓ Should allow the owner to add a sub-collection
- ✓ Should not allow the owner to add a duplicate sub-collection
- ✓ Should mint a token and assign it to the sender
- ✓ Should transfer a token from one address to another
- ✓ Should revert on available main supply exceeded
- ✓ Should revert on max supply pre address exceeded
- ✓ Should set the trusted forwarder
- ✓ Should set the mint price
- ✓ Should set the maximum number of tokens that can be minted by a single address
- ✓ Should set the whitelist minting option
- ✓ Should enable/disable the burn functionality
- ✓ Should set the royalty information

Functional Tests

- ✓ Should return the correct sub-collection ID by token ID
- ✓ Should revert when sending to the blacklisted address
- ✓ Should revert when approving to the blacklisted address
- ✓ Should revert when no nested collections

ReferralProcessor

- ✓ Should set project and register owner pools
- ✓ Should process referral operation with discount when eligible
- ✓ Should distribute funds according to owner pools
- ✓ Should remove owner pools

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the Quantelica. We performed our audit according to the procedure described above.

Some issues of High, Medium, Low and informational severity were found. Some suggestions and best practices are also provided in order to improve the code quality and security posture. In the end, Quantelica team resolved all issues.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in Quantelica smart contracts. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of Quantelica smart contracts. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services. It is the responsibility of the Quantelica to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

850+
Audits Completed

$30B
Secured

$30B
Lines of Code Audited

Follow Our Journey

Audit Report
February, 2024

For

QUANTELICA

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com