QuillAudits

Audit Report
January, 2023

For

ARTSWAP

Table of Content

Executive Summary

Project Name	Artswap
Overview	The LazyMinting.sol contract is solely for minting NFTs in a specific fashion, allows mints only from signatures signed by a specific minter and stores the hashes to reduce the risk of replays.
Timeline	22 December, 2022 to 2 January, 2023
Method	Manual Review, Functional Testing, Automated Testing etc.
Scope of Audit	The scope of this audit was to analyze the LazyMinting codebase for quality, security, and correctness. *https://github.com/artswap-rumsan/artswap-contracts/blob/master/contracts/lazyMinting.sol*
Branch	Master
Commit Hash	b37bcf366eb4ffdfcd49899f81a36768f7a89acf
Fixed In	*https://github.com/artswap-rumsan/artswap-contracts/blob/master/contracts/lazyMinting.sol*
Commit hash	4b7977c1e5b4e370b173337c59cdf9c4f4a74528

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	1	0	1	4

Types of Severities

High

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

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

Low-level severity issues can cause minor impact and or 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 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.

Checked Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Techniques and Methods

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

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token 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 analysed 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

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 analysed, 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 behaviour 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, Truffle Team, Solhint, Mythril, Slither, Solidity statistic analysis.

Manual Testing

A. Contract - LazyMinting.sol

High Severity Issues

1. Uninitialized variables

Description
The isMinted mapping is never initialized and when it is called in mintNft()
require(!isMinted[messageHash], 'LazyMinter:Nft already minted');
The pass in the require statement will always pass and the mint function will never revert. This goes against the contract logic and expected code flow.

Remediation
Push the message hash to the isMinted mapping after a call to mint. If the devs want to restrict the signatures to single use only, ensure the mapping gets updated even if the mint call fails.

Status
Resolved

Medium Severity Issues

No issues found

Low Severity Issues

2. Address casting

Description
The gallery address in the updateTokenInfo() function is casted and assigned as a payable parameter but the TokenStructLibrary holds a non-payable address instead.

```
function updateTokenInfo(...) internal {
        ...
        payable(_metadata.gallery),
        ...
}

TokenStructLib.sol
struct TokenInfo {
    ...
    address gallery; // address of individual gallery
    address payable thirdParty;
    ...
}
```

Also, the require check in the mintNft function checks if the Gallery owner is the same as the minter. This check does not require casting the address to payable.

```
function mintNft(metaData calldata _metadata, bytes calldata _sign) external
 payable nonReentrant {
    ...
    require(Gallery(payable(_metadata.gallery)).owner() == minter,...);
    ...
 }
```

Remediation
If the address would need to receive ether, make it payable else reduce the privileges available to prevent possible attack vectors that can arise.

Status
Resolved

Informational Issues

3. Recursive imports

Description
The majority of contracts, interfaces and libraries imported into the LazyMinting.sol contract file already have been imported in previous contracts.

Remediation
To avoid unnecessary recursive calls, consider removing the import statements.

Status
Resolved

4. Dead code

Description
Lines 24-26 of the INFT interface has unused code commented there. The variable remaningFee is never used in LazyMinting's mintNft function, it is an unnecessary return value from the calculateCommissions() internal call.

Remediation
To improve code readability, consider removing these lines of code since they are unused.

Status
Resolved

5. Spellings and Solidity Style Guide Formatting

Description
In the mintNft function, there is a misspelling of one of the parameters as remaningfee. Also some functions could be declared in NATSPEC format, and following the Solidity Style Guide formatting.

Remediation
To improve readability and follow best practices, consider renaming remaningfee to remainingFee and transferfees to transferFees. Also, other internal functions and variables should follow the style guide formatting to improve code readability.

Status
Resolved

6. Missing events for changes to state

Description
Important functions do not have changes to state trigger events. Without events, blockchain monitoring systems do not detect suspicious behaviour.

Remediation
It is advisable for the changeNftAddress, changeMarketPlaceAddress and any other important functions to trigger events after they are called.

Status
Resolved

Automated Tests

```
LazyMinters.isMinted (contracts/lazyMinting.sol#58) is never initialized. It is used in:
        - LazyMinters.mintNft(LazyMinters.metaData,bytes) (contracts/lazyMinting.sol#90-117)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-state-variables
```

```
Reentrancy in LazyMinters.mintNft(LazyMinters.metaData,bytes) (contracts/lazyMinting.sol#90-117):
        External calls:
        - checkAmount(price,msg.sender) (contracts/lazyMinting.sol#102)
                - (txSuccess) = receiver.call{value: _amount}() (contracts/lazyMinting.sol#192)
        - transferfees(platformAddress,_platformfee) (contracts/lazyMinting.sol#110)
                - (txSuccess) = receiver.call{value: _amount}() (contracts/lazyMinting.sol#192)
        - transferfees(minter,_galleryOwnerfee) (contracts/lazyMinting.sol#111)
                - (txSuccess) = receiver.call{value: _amount}() (contracts/lazyMinting.sol#192)
        - transferfees(_metadata.artist,_artistfee) (contracts/lazyMinting.sol#112)
                - (txSuccess) = receiver.call{value: _amount}() (contracts/lazyMinting.sol#192)
        - tokenId = nft.mint(_metadata.tokenUri,msg.sender) (contracts/lazyMinting.sol#113)
        - nft.setArtistRoyalty(tokenId,_metadata.artist,uint96(_metadata.artistRoyalty)) (contracts/lazyMinting.sol#114)
        - updateTokenInfo(tokenId,_metadata,msg.sender) (contracts/lazyMinting.sol#115)
                - tokenInfo.addTokenInfo(Token) (contracts/lazyMinting.sol#237)
        External calls sending eth:
        - checkAmount(price,msg.sender) (contracts/lazyMinting.sol#102)
                - (txSuccess) = receiver.call{value: _amount}() (contracts/lazyMinting.sol#192)
        - transferfees(platformAddress,_platformfee) (contracts/lazyMinting.sol#110)
                - (txSuccess) = receiver.call{value: _amount}() (contracts/lazyMinting.sol#192)
        - transferfees(minter,_galleryOwnerfee) (contracts/lazyMinting.sol#111)
                - (txSuccess) = receiver.call{value: _amount}() (contracts/lazyMinting.sol#192)
        - transferfees(_metadata.artist,_artistfee) (contracts/lazyMinting.sol#112)
                - (txSuccess) = receiver.call{value: _amount}() (contracts/lazyMinting.sol#192)
        Event emitted after the call(s):
        - Lazyminted(tokenId,msg.sender,_metadata.gallery,_metadata.minPrice) (contracts/lazyMinting.sol#116)
```

```
Struct LazyMinters.metaData (contracts/lazyMinting.sol#31-56) is not in CapWords
Parameter LazyMinters.mintNft(LazyMinters.metaData,bytes)._metadata (contracts/lazyMinting.sol#90) is not in mixedCase
Parameter LazyMinters.mintNft(LazyMinters.metaData,bytes)._sign (contracts/lazyMinting.sol#90) is not in mixedCase
Function LazyMinters._verify(bytes32,bytes) (contracts/lazyMinting.sol#122-125) is not in mixedCase
Parameter LazyMinters._verify(bytes32,bytes)._hashData (contracts/lazyMinting.sol#122) is not in mixedCase
Parameter LazyMinters._verify(bytes32,bytes)._sign (contracts/lazyMinting.sol#122) is not in mixedCase
Parameter LazyMinters.getMessageHash(string)._message (contracts/lazyMinting.sol#129) is not in mixedCase
Parameter LazyMinters.changeNftAddress(address)._newNft (contracts/lazyMinting.sol#138) is not in mixedCase
Parameter LazyMinters.changeMarketPlaceAddress(address)._newMarketPlace (contracts/lazyMinting.sol#145) is not in mixedCase
Parameter LazyMinters.calculateCommissions(LazyMinters.metaData)._metadata (contracts/lazyMinting.sol#152) is not in mixedCase
Parameter LazyMinters.cutPer10000(uint256,uint256)._cut (contracts/lazyMinting.sol#181) is not in mixedCase
Parameter LazyMinters.cutPer10000(uint256,uint256)._total (contracts/lazyMinting.sol#181) is not in mixedCase
Parameter LazyMinters.transferfees(address,uint256)._amount (contracts/lazyMinting.sol#191) is not in mixedCase
Parameter LazyMinters.updateTokenInfo(uint256,LazyMinters.metaData,address)._tokenId (contracts/lazyMinting.sol#214) is not in mixedCase
Parameter LazyMinters.updateTokenInfo(uint256,LazyMinters.metaData,address)._metadata (contracts/lazyMinting.sol#215) is not in mixedCase
Parameter LazyMinters.updateTokenInfo(uint256,LazyMinters.metaData,address)._nftOwner (contracts/lazyMinting.sol#216) is not in mixedCase
Variable LazyMinters.Imarketplace (contracts/lazyMinting.sol#62) is not in mixedCase
Parameter IGallery.sellNft(uint256,uint256,IGallery.FeeInfo,address,uint256,bool,bool).USD (interface/IGallery.sol#84) is not in mixedCase
Struct IGalleryFactory.galleryAndNFT (interface/IGalleryFactory.sol#14-27) is not in CapWords
Parameter IMarketPlace.sell(uint256,uint256,uint256,uint256,uint256,uint256,address,address,bool).USD (interface/IMarketPlace.sol#30) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions

Redundant expression "thirdPartyCommission (contracts/NFTMarketInfo.sol#191)" inNFTMarketInfo (contracts/NFTMarketInfo.sol#14-268)
Redundant expression "artistcommssion (contracts/NFTMarketInfo.sol#192)" inNFTMarketInfo (contracts/NFTMarketInfo.sol#14-268)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#redundant-statements

Variable LazyMinters.Imarketplace (contracts/lazyMinting.sol#62) is too similar to LazyMinters.constructor(address,address,address)._marketPlace (contracts/lazyMinting.sol#75)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#variable-names-are-too-similar
```

Results

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 LazyMinting codebase. We performed our audit according to the procedure described above.

Some issues of High, 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, ArtSwap Team Resolved all issues.

Disclaimer

QuillAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the Artswap Platform. This audit does not provide a security or correctness guarantee of the audited smart contracts.

The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the Artswap Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

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.

700+
Audits Completed

$16B
Secured

700K
Lines of Code Audited

Follow Our Journey

Audit Report
January, 2023

For

ARTSWAP

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com