QuillAudits

Audit Report
September, 2022

For

SpatialLabs

Table of Content

Table of Content

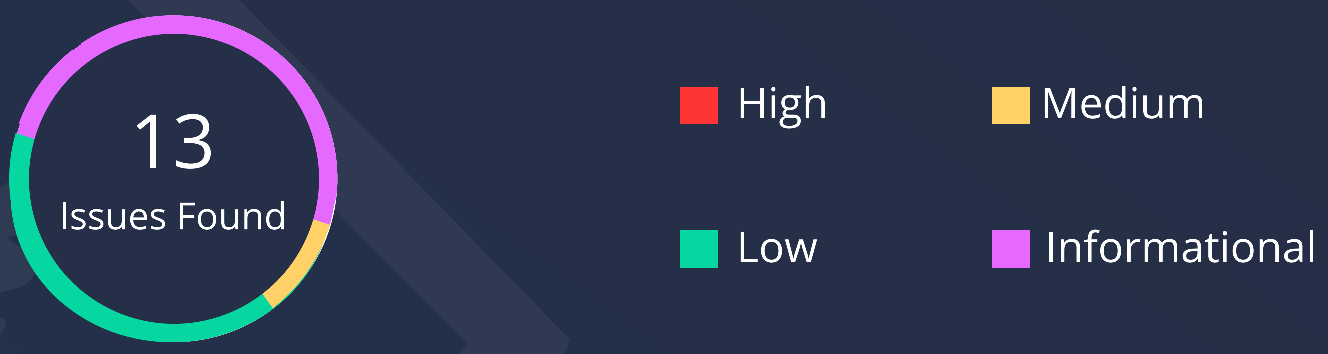
Executive Summary

Project Name Slabs Contracts

Overview Slabs audited contracts are for Non Fungible Tokens related project minting on Polygon blockchain with Marketplace, LNQ ERC20 Token, meta transactions and Swap functionality between the native deployed blockchain token Polygon Matic and LNQ Token.
Manual Review, Functional Testing, Automated Testing etc.

Scope of Audit The scope of this audit was to analyse Slabs smart contract's codebase for quality, security, and correctness. This included testing of smart contracts to ensure proper logic was followed, manual analysis ,checking for bugs and vulnerabilities, checks for dead code, checks for code style, security and more. The audited contracts are as follows:
Git Repo link : *https://github.com/Project-slabs/blockchain*
Git Branch: main branch
Commit Hash: 1e61473ff6170c47d4ff7fb3e34e72b031d026d8

Fixed In Git Branch: audit-fix
Commit Hash: 95733a40dd11366a9afc24e064d33910daa3a0b



	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	0	1	0
Partially Resolved Issues	0	0	0	0
Resolved Issues	0	1	4	7

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.

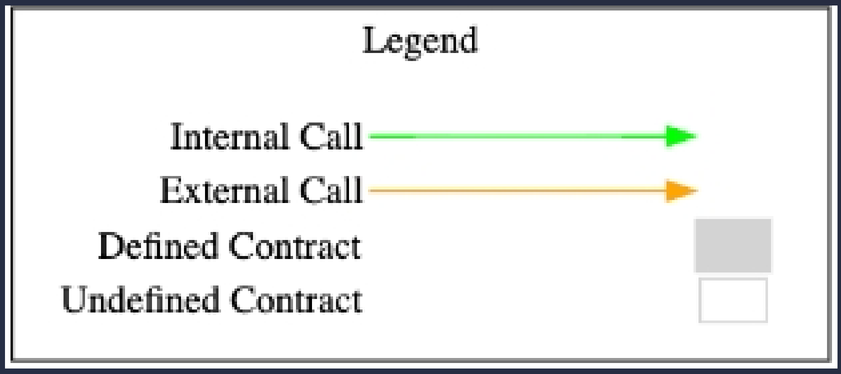
Contract's Information

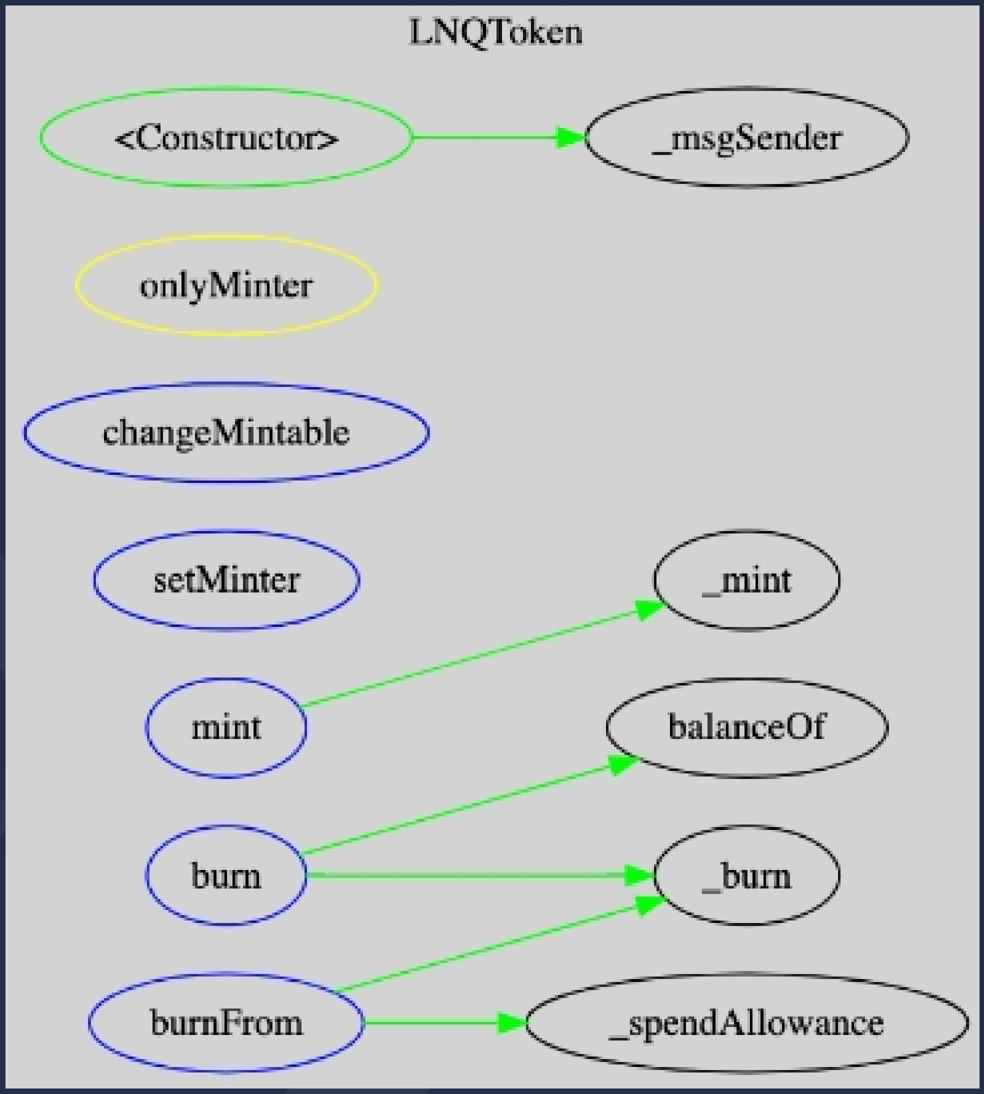
CONTRACTS	Lines	Complexity Score	Capabilities
contracts/Interfaces/IAuctionHouse.sol	155	1	
contracts/Interfaces/IERC2981.sol	27	5	assembly, experimental features, hash functions
contracts/ERC20Forwarder.sol	443	146	payable, initiates ETH value transfer
contracts/Collection.sol	335	125	payable, experimental features
contracts/Swap.sol	90	47	
contracts/Marketplace.sol	744	268	
contracts/UserMintableCollection.sol	377	128	
contracts/utils/CollectionWhitelist.sol	38	16	
contracts/utils/CollectionAuthorizable.sol	56	27	
contracts/utils/MarketPlaceWhitelist.sol	41	16	
contracts/utils/Authorizable.sol	56	27	
contracts/utils/UserMintableCollectionWhitelist.sol	41	16	
contracts/LNQToken.sol	76	35	
contracts/Factory.sol	39	32	create/create2
TOTALS	2518	889	assembly, experimental features, hash functions, initiates ETH value transfer, create/create2, payable

CONTRACTS	COUNT
@chainlink/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol	1
@openzeppelin/contracts/access/Ownable.sol	6
@openzeppelin/contracts/metatx/ERC2771Context.sol	3
@openzeppelin/contracts/security/Pausable.sol	1
@openzeppelin/contracts/security/ReentrancyGuard.sol	2
@openzeppelin/contracts/token/ERC20/IERC20.sol	2
@openzeppelin/contracts/token/ERC20/extensions/draft-ERC20Permit.sol	1
@openzeppelin/contracts/token/ERC20/extensions/draft-IERC20Permit.sol	1
@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol	2
@openzeppelin/contracts/token/ERC721/IERC721.sol	1
@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol	2
@openzeppelin/contracts/token/ERC721/utils/ERC721Holder.sol	1
@openzeppelin/contracts/utils/Counters.sol	1
@openzeppelin/contracts/utils/cryptography/ECDSA.sol	1
@openzeppelin/contracts/utils/introspection/IERC165.sol	1
@openzeppelin/contracts/utils/structs/EnumerableSet.sol	2

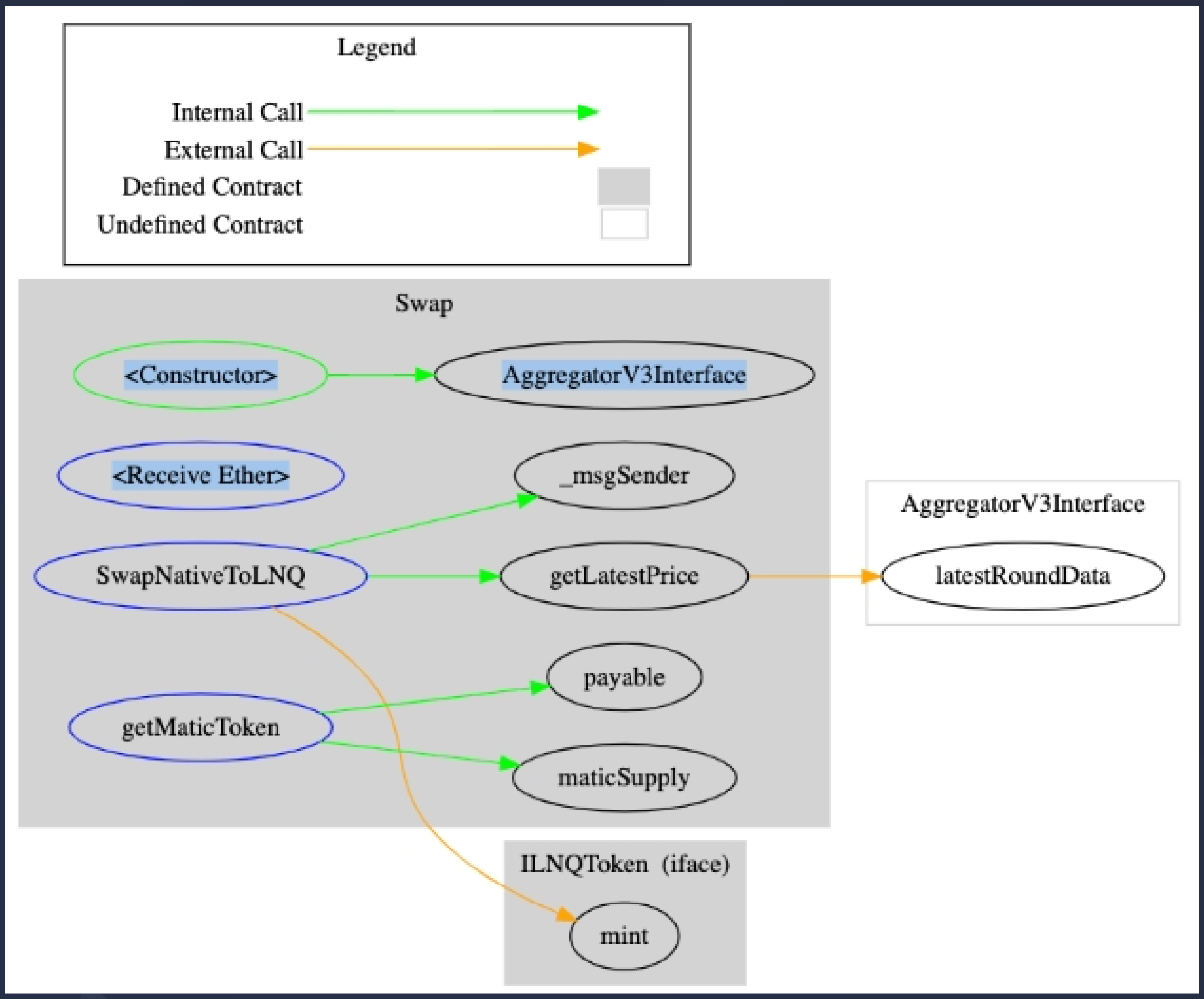
Call Graphs

contracts/LNQTocken.sol

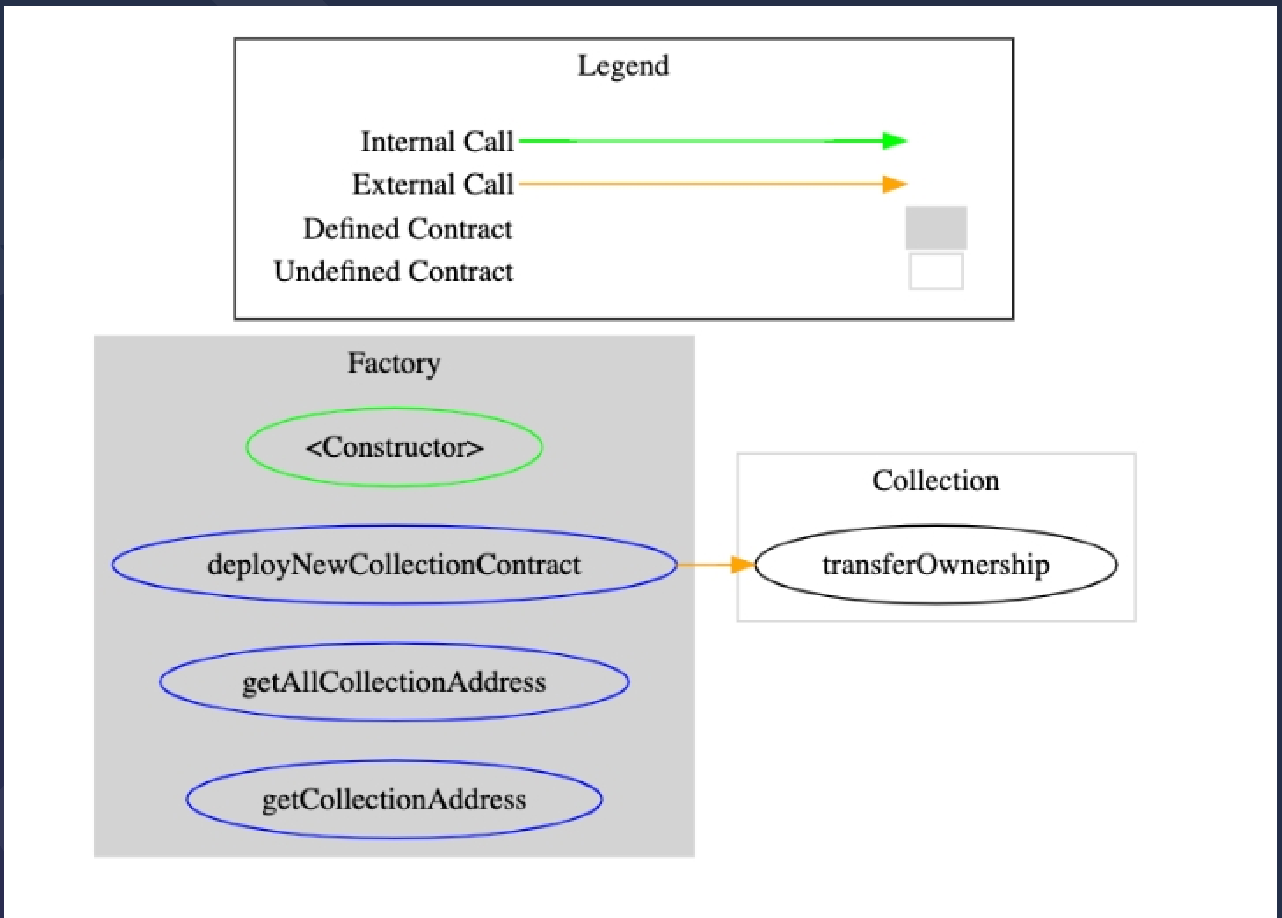




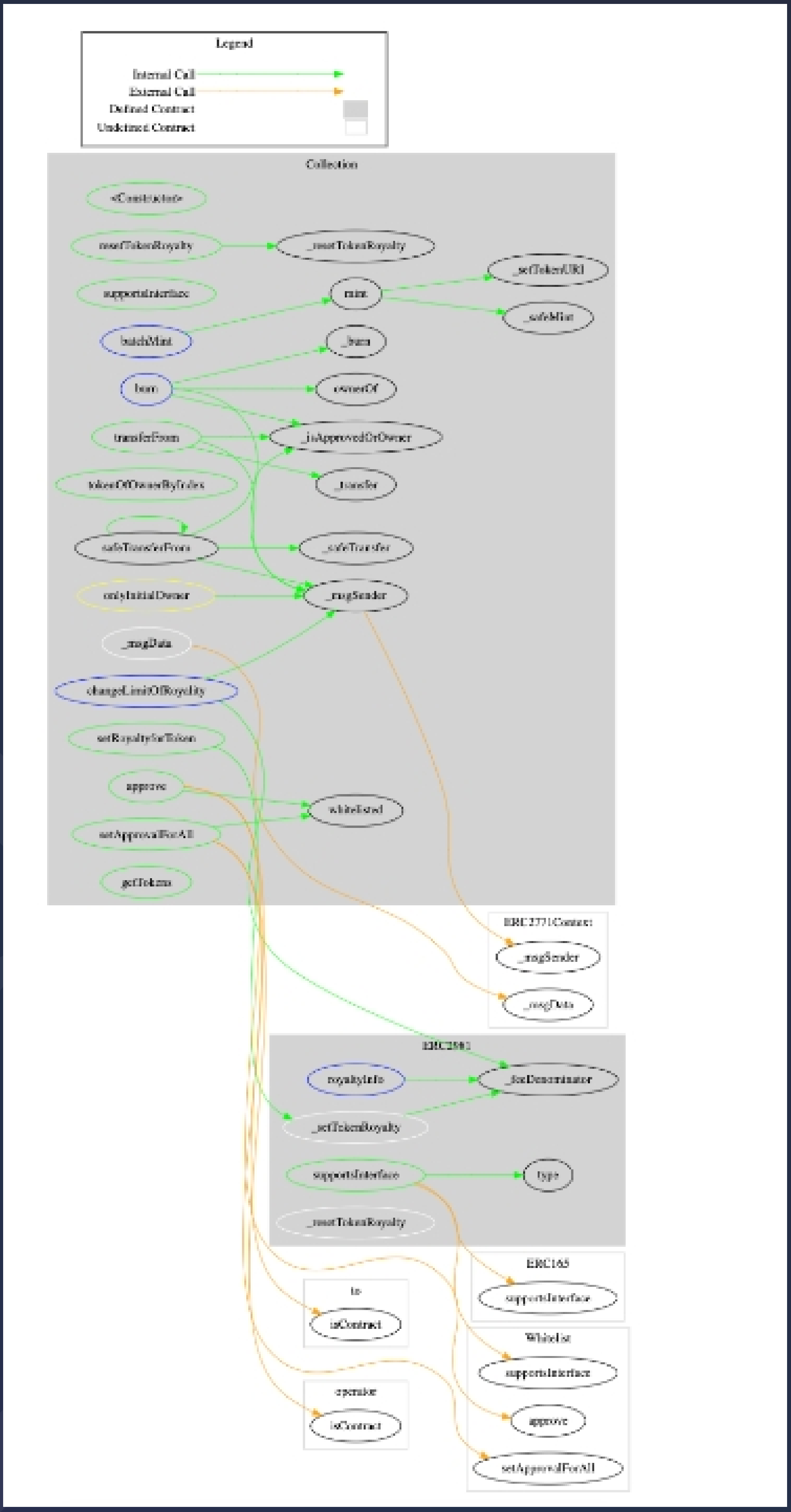
contracts/Swap.sol



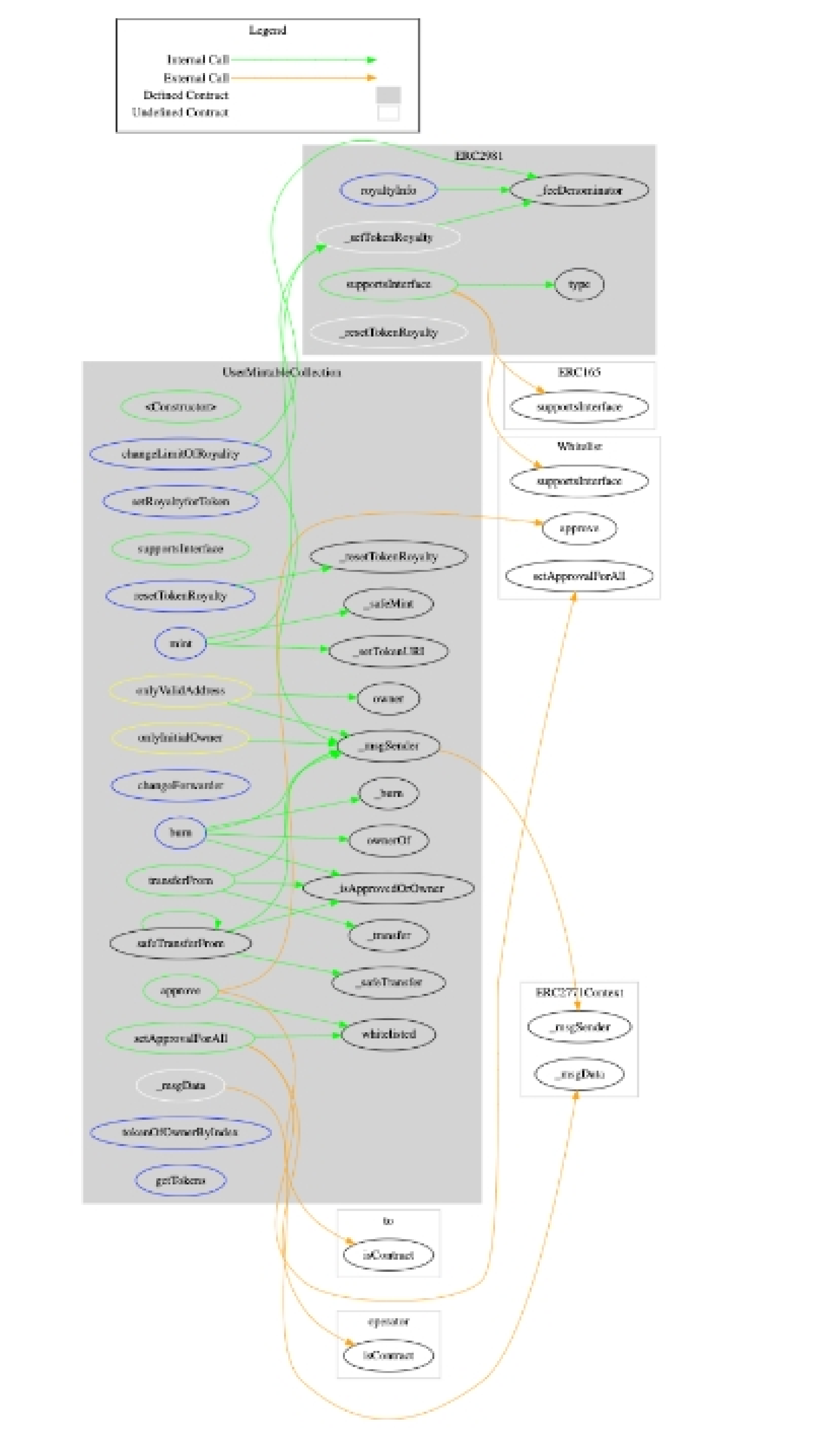
contracts/Factory.sol



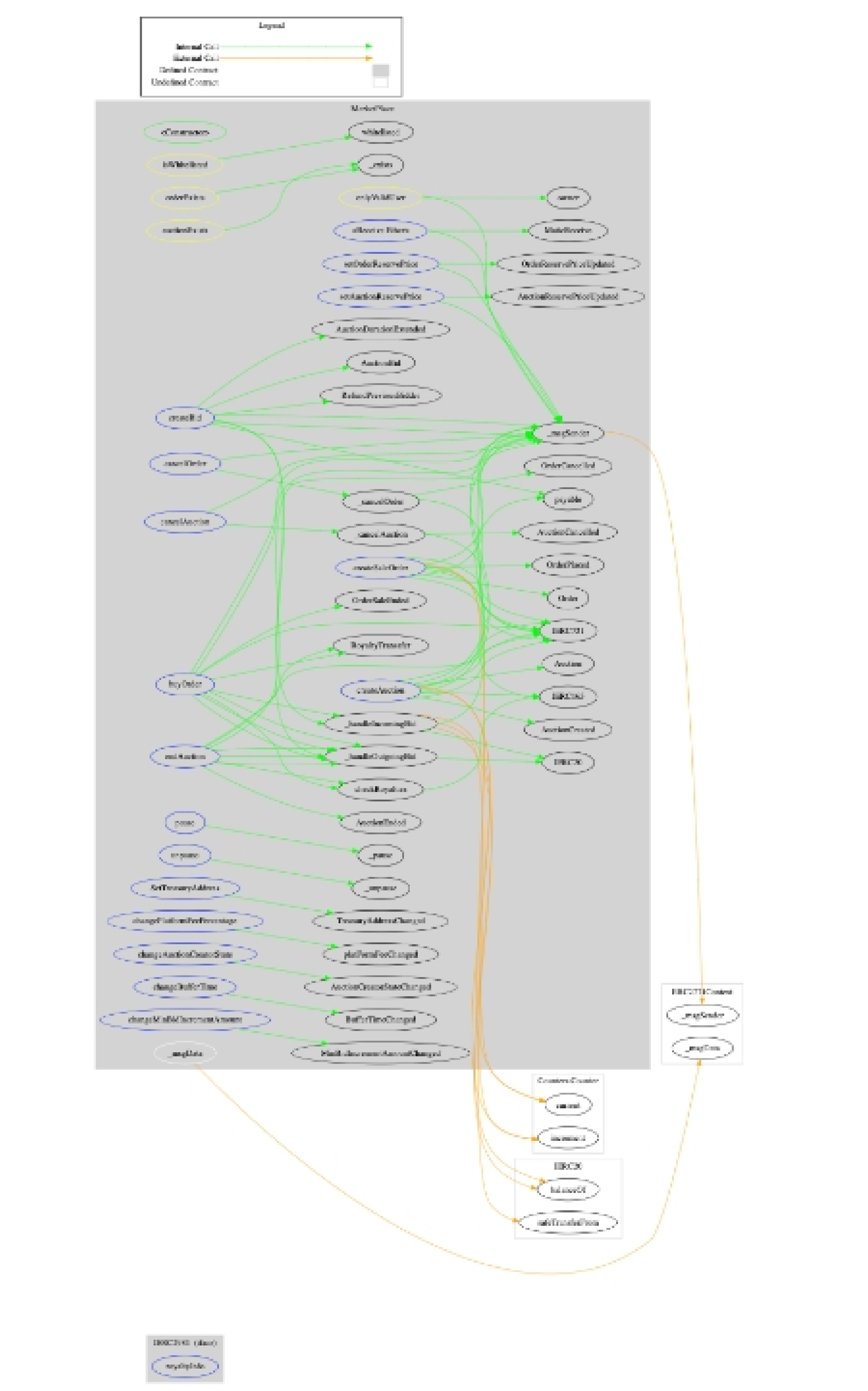
contracts/Collection.sol



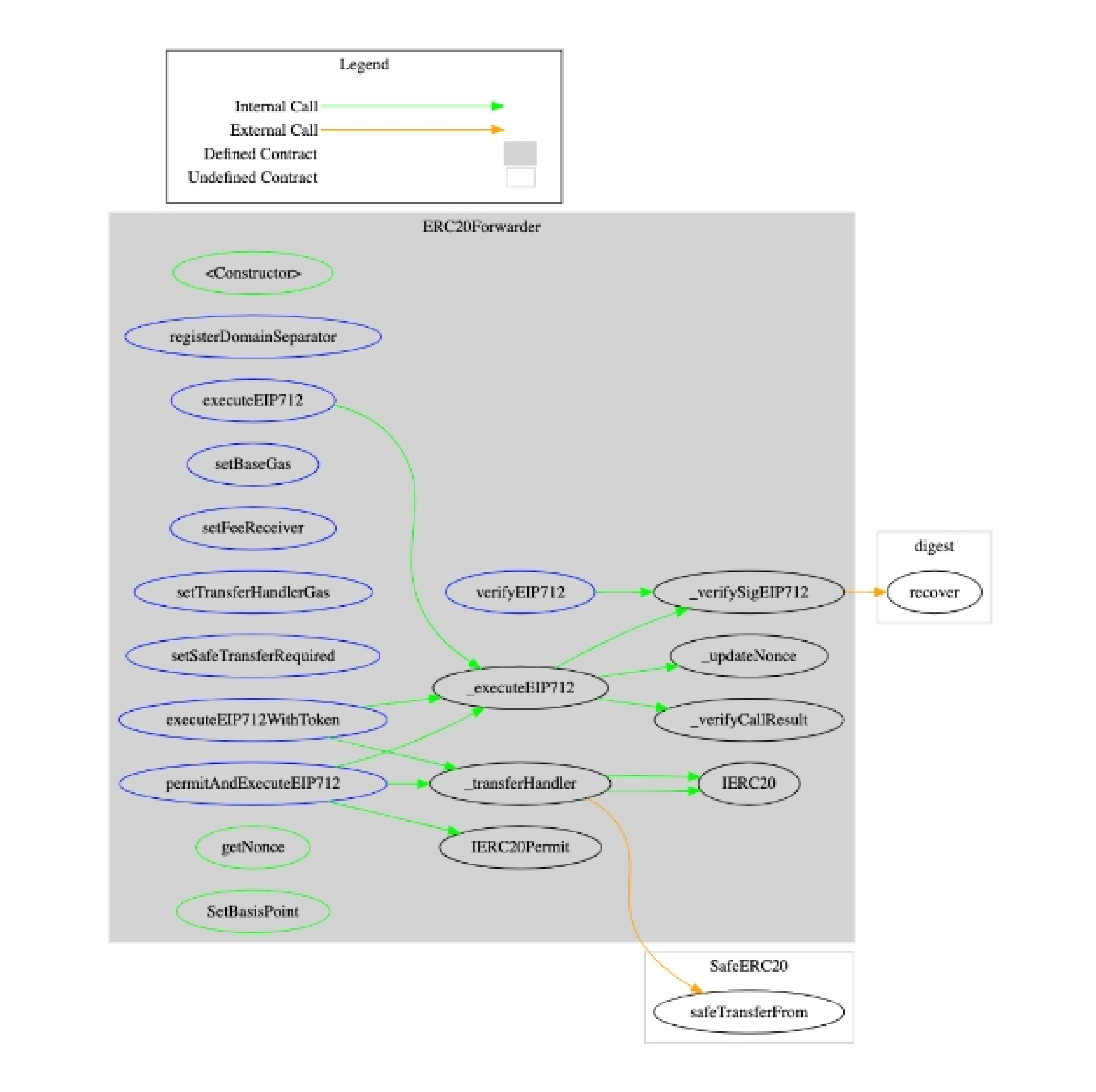
contracts/UserMintableCollection.sol



contracts/Marketplace.sol



contracts/ERC20Forwarder.sol



Manual Testing

High Severity Issues

No issues found

Medium Severity Issues

A1. Reentrancy

Description

Several of the following functions in the contracts may be prone to reentrancy attacks.

UserMintableCollectionWhitelist.sol line 217-240 function mint(...)
_safeMint(..) function line 234 can allow for reentrancy due to onERC721Received allowing a contract to callback into the mint function. State changes like setting token URI, royalty and initial owner are only done after this external call.

Collection.sol line 232-241 function mint(...)
_safeMint(..) function line 238 can allow for reentrancy due to onERC721Received allowing a contract to callback into the mint function. State changes like setting token URI, royalty and initial owner are only done after this external call.

Remediation

It is recommended to follow CEI (checks -effects-interactions) pattern in especially functions with external calls. This implies state updates, events emission must ideally occur before external calls. Additionally may make use of OpenZeppelin Reentrancy guard by adding nonReentrant to potentially vulnerable functions.

Auditor's Response: Fixed with OpenZeppelin nonReentrant modifier

Status

Resolved

Low Severity Issues

A2. Centralization risks and overpowered access control roles

Description

Contracts e.g LNQToken.sol,Swap.sol, UserMintableCollection, Factory.sol etc have ownership and access control roles assigned that control critical functions and operations of the contracts. Critical functions and actions are such as Mintable, Pausable. There is no indication if these accounts will be single addresses or multisigs. If single addresses this can bring about centralization risks.

Remediation

It is recommended to document access control and roles, how they will work, risks of access roles, process of changing roles etc. It may be prudent to have different accounts playing different roles in various Access aspects for contracts or document this if intention. It may be prudent to have multisig account control various owner and access control roles. It may be prudent to consider a two step process with changes of critical roles where e.g new owner role is granted and claimed by the new owner. It may be prudent to ensure renounceOnwership() is never called maliciously or accidentally by current owner e.g override and revert() in inherited Ownable contracts if intention is to never renounce Ownership on these contracts

Auditor's Response: Slabs Team Acknowledged the Issue

Status

Acknowledged

A3. Unchecked return values

Description

Certain operations in functions are not checking the return values. This results in a risk processing function with assumption intermediate steps was a success when it could have failed, especially when returning booleans. Consider the following lines of code:

Collection.sol line 237 _holderTokens[_to].add(tokenId);
Collection.sol line 252 _holderTokens[tokenOwner].remove(_tokenId);
Collection.sol line 291 _holderTokens[from].remove(tokenId);
Collection.sol line 292 _holderTokens[to].add(tokenId);
Collection.sol line 320 _holderTokens[from].remove(tokenId);
Collection.sol line 321 _holderTokens[to].add(tokenId);
UserMintableCollection.sol line 233 _holderTokens[_to].add(tokenId);
UserMintableCollection.sol 258 _holderTokens[tokenOwner].remove(_tokenId);
UserMintableCollection.sol 304 _holderTokens[from].remove(tokenId);
UserMintableCollection.sol 305 _holderTokens[to].add(tokenId);
UserMintableCollection.sol 333 _holderTokens[from].remove(tokenId);
UserMintableCollection.sol 334 _holderTokens[to].add(tokenId);

Remediation

It is recommended to check all return values, were applicable e.g
bool removeHolder = _holderTokens[from].remove(tokensId);
bool addHolder = _holderTokens[to].add(tokensId);
require(removeHolder && addHolder, “Tx failed”);

Auditor's Response: All return values checked and require() statements applied

Status

Resolved

A4. Missing events

Description

Some critical operations, actions and functions are missing events e.g
ERC20Forwarder.sol line 214-220 function setSafeTransferRequired
UserMintableCollection.sol line 245-247 function changeForwarderAddress
LNQToken.sol line 31-34 function changeMintable
Marketplace.sol line 553-555 function pause()
Marketplace.sol line 560-562 function unpause()

Above functions are critical functions and or owner only functions etc that change working of the contracts.

Remediation

Consider using the latest solidity version.

Auditor's Response: Missing events added! OpenZeppelin Pausable already has Paused and UnPaused events. All critical functions have events.

Status

Resolved

A5. Use of .transfer() and .send() to send ETH

Description

Use of transfer() in the following functions
Swap.sol line 64 function getMaticToken
Above mentioned functions were introduced to mitigate reentrancy as they restricted the amount of gas sent. However the best practice is to use .call{} whilst ensuring checks-effects-interactions are done to avoid Reentrancy or make use of Reentrancy guards. Reentrancy protection is already used in function.

Remediation

It is recommended to use .call{value: _amount}("") as in the above example and all other cases where necessary. If external calls to untrusted accounts ensure reentrancy protection and or checks effects interactions are always followed e.g event is emitted before external interactions.

Auditor's Response: Added .call{} method instead of transfer in getMeticToken

Status

Resolved

A6. Zero address checks

Description

Some addresses are not checked for zero address which can result in loss or burning of funds or tokens or incorrect logic
Collection.sol line 216 address _to
Factory.sol line 19 address trustedForwarder
LNQToken.sol line 40 address _minter
Swap.sol line 55 address _address
UserMintableCollaction.sol line 245 address _newOwner;

Remediation

It is recommended to check if address inputs are not zero addresses in above cases and all other cases that may be relevant.
E.g require(address _receiver != address(0),"error string")

Auditor's Response: Zero address checks added.

Status

Resolved

Informational Issues

A7. Variables that can be declared immutable

Description

Variables below are assigned during the contract creation phase, but remain constant throughout the life-time of a deployed contract
Marketplace.sol line 71 address public LNQTokenAddress;

Remediation

It is recommended to make the above variables immutable to save on gas costs

Auditor's Response: Immutable applied to LNQTokenAddress variable.

Status

Resolved

A8. Public functions that can be made external

Description

There are several functions that have public visibility but are never called within the contracts. See examples below.
Collection.sol line 151 function setsetRoyaltyforToken(..) public
Collection.sol line 162 function resetTokenRoyalty(..) public
Collection.sol line 186 function tokenOfOwnerByIndex(..) public
Collection.sol line 328 function getTokens(..) public
CollectionWhitelist.sol line 17 addAddress(...) public
CollectionWhitelist.sol line 30 removeAddress(...) public
MarketPlaceWhitelist.sol line 38 whitelisted(...) public
ERC20Forwarder.sol line 286 getNonce(..) public
ERC20Forwarder.sol line 297 SetBasisPoint(..) public
UserMintableCollectionWhitelist.sol line 38 whitelisted(...) public

Remediation

It is recommended to save on gas costs by making the above functions and all other instances of public functions never called in contracts external. External functions cost less than public functions

Auditor's Response: All functions that can be made external applied

Status

Resolved

A9. Variables without explicit visibility

Description

Certain variables such as the example below have no visibility specified explicitly. Default visibility of variables is internal if not specified. However by not stating explicitly it is not clear if that was the intention.
Factory.sol line 12 address[] collectionAddresses
Marketplace.sol line 73 bytes4 constant INTERFACE_ID = 0x80ac58cd;
ERC20Forwarder.sol line 51 uint256 chainId;
ERC20Forwarder.sol line 77
mapping(address => mapping(uint256 => uint256)) nonces;

Remediation

It is recommended to explicitly declare variable visibility e.g mapping(address => mapping(uint256 => uint256)) public nonces; Not only does it improve code readability or maintainability, it allows for automatic getter functions if the intention is to read the variables as well. It is recommended to think carefully about function visibility and start with being restrictive e.g private, and only expand if necessary for a variable.

Auditor's Response: Visibility applied to all variables

Status

Resolved

A10. Implicit use of uint

Description

In the code there are places where uint is used without being specific if it is uint256, uint96 etc although it defaults to uint256 this can cause confusion and or errors.
Factory.sol line 36 function getCollectionAddress(uint _gameId)

Remediation

It is recommended for readability, maintainability and avoidance of errors to be explicit with uint declarations. If it is to be uint256 specify as such or specify as required uint e.g uint32 etc. Remember that smaller units e.g uint8 etc are not always cheaper than uint256.

Auditor's Response: explicit uint256 applied to function parameter

Status

Resolved

A11. require() missing error strings

Description

Some require() statements are missing error strings .
ERC20Forwarder.sol line 298 require(_bp > 0)
ERC20Forwarder.sol line 403
require(IERC20(req.token).transferFrom(req.from, feeReceiver, charge))
Marketplace.sol line 91
require(_LNQAddress != address(0) | | trustedForwarder != address(0));
Marketplace.sol line 568 require(_treasury != address(0));
Marketplace.sol line 671 require(currency != address(0));
Marketplace.sol line 689 require(currency != address(0));

Remediation

It is recommended that all require() statements have an error string to describe the failure. E.g require(_treasury != address(0), “treasury address required”);

Auditor's Response: Visibility applied to all variables

Status

Resolved

A12. Unused code

Description

Some contracts have code that is not used. This may lead to challenges in readability, maintainability of code, increase code size unnecessarily or represent incomplete or missing logic in the contracts.
Collection.sol line 203-210 function _msgData(...)
Marketplace.sol line 647-654 function _msgData(...)
UserMintableCollection.sol line 347-654 function _msgData(...)
Swap.sol line 22-26 event BurnLNQAndGetMaticToken(...)

Remediation

It is recommended to remove code if fully checked and confirmed that it is not required.

Auditor's Response: Removed unused event. Kept _msgData() used to override(Context, ERC2771Context)

Status

Resolved

A13. Unused code

Description

Some code logic makes unnecessary comparisons to boolean constants
UserMintableCollectionWhitelist.sol line 16
 whitelistedMap[_address] != true,
UserMintableCollectionWhitelist.sol line 28
whitelistedMap[_address] != false,
CollectionWhitelist.sol line 19
whitelistedMap[_address] != true,
CollectionWhitelist.sol line 32
whitelistedMap[_address] != false,
MarketPlaceWhitelist.sol line 16
whitelistedMap[_address] != true,
MarketPlaceWhitelist.sol line 28
whitelistedMap[_address] != false,

Remediation

It is recommended to use existing boolean value and negate where needed to check for truthiness or falseness of value e.g
(!whitelistedMap[_address]) vs whitelistedMap[_address] != true
(whitelistedMap[_address]) vs whitelistedMap[_address] != false

Auditor's Response: Unnecessary comparisons removed and changed.

Status

Resolved

Closing Summary

Some issues of Medium, Low and Informational severity were found in the Initial Audit and were all fixed by the client. Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

QuillAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the Slabs 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 Slabs 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.

600+
Audits Completed

$15B
Secured

600K
Lines of Code Audited

Follow Our Journey

Audit Report
September, 2022

For

SpatialLabs

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com