QuillAudits

AUDIT REPORT

January 2024

For

toyow

Table of Content

Executive Summary

Project name	Toyow
Overview	Toyow ERC1155 is a contract designed for users to mint, list and purchase tokens for tokenID 1. The Toyow contract inherits the following contracts from the standard Openzeppelin library; ERC1155, Ownable, Pausable, ERC1155Supply, ReentrancyGuard, and IERC1155Receiver. These libraries and extensions helped in identifying privileged functions for the platform owner, track the total supply of tokenID, prevent reentrancy attacks and build the compatibility feature for the contract to receive ERC1155 tokens. Toyow ERC721 is built in the similar fashion to mint, list and purchase tokens. There is a maximum quantity of 27 tokens which could be increased in the future. These 27 tokens when first listed, the creators of these tokens receive a native token when purchased. Afterwards, these tokens can only be listed in USD.
Timeline	8th december 2023 - 26th december 2023
Audit Scope	The scope of this audit was to analyze the Toyow codebase for quality, security, and correctness.
Source code	https://github.com/XChain-Technologies/Contracts/tree/main/Final%20contracts%20review
Branch	Main
Contracts under scope	1155.sol 721.sol

Update code Received	6th january 2024
Second Review	8th January 2024
Fixed In	https://github.com/XChain-Technologies/Contracts/blob/main/Final%20contracts%20review/1155.sol https://github.com/XChain-Technologies/Contracts/blob/main/Final%20contracts%20review/721.sol

Checked Vulnerabilities

- ☑ Re-entrancy
- ☑ Timestamp Dependence
- ☑ Gas Limit and Loops
- ☑ DoS with Block Gas Limit
- ☑ Transaction-Ordering Dependence
- ☑ Use of tx.origin
- ☑ Exception disorder
- ☑ Gasless send
- ☑ Balance equality
- ☑ Byte array
- ☑ Transfer forwards all gas
- ☑ ERC20 API violation
- ☑ Malicious libraries
- ☑ Compiler version not fixed
- ☑ Redundant fallback function
- ☑ Send instead of transfer
- ☑ Style guide violation
- ☑ Unchecked external call
- ☑ Unchecked math
- ☑ 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, Foundry, Solhint, Mythril, Slither, Solidity statistical 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 Issues

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

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

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.

High Severity Issues

Native token will be locked forever in contract

Resolved

Path

toyow1155.sol

Description

In toyow1155 contract, when user mints tokens with mint function, the matic token which is native token of polygon chain is transferred to the contract. But when the admin/owner of the protocol goes to withdraw the amount, there is no function which lets him do it. Which will result in permanently locking of the tokens in the contract.

Recommendation

To resolve the issue add withdrawal function for withdrawing native matic token.

Attacker can steal primary fees of the creator

Acknowledged

Path

721.sol

Function

setCreator()

Description

In contract 721, when the nfts are minted their creators are set using a mapping for the primary sales. As setCreator() is set as a public function an attacker can come and update the value to make it his address. So when nft is bought by any other user the calculated creator fees/royalty is sent to the attacker's address making loss of the user/creator.

POC

-

```
function test_CreatorIsUnableToGetRoyalty() external {
    vm.startPrank(creator);
    // Created minted NFT
    nft.safeMint();

    assertEq(nft.balanceOf(creator), 1);

    // Creator lists nft for sale
    nft.listNFT(1, 1 ether, "MATIC", creator);
    assertEq(nft.balanceOf(creator), 0);
    vm.stopPrank();

    // Attacker changes the creator address for token ID
    vm.prank(user1);
    nft.setCreator(1, user1);

    vm.prank(user2);
    nft.buyNFT{ value: 1 ether }(1, "MATIC");

    assertEq(nft.balanceOf(user2), 1);
    assertEq(address(creator).balance, 0 ether);
    console2.log("BALANCE of user1::", address(user1).balance);
    console2.log("BALANCE of user2::", address(user2).balance);
    console2.log("BALANCE of creator::", address(creator).balance);
}
```

Recommendation

To resolve the issue you can check that:
1) the creator is the one updating the address who owns that nft or
2) make this function private so it is only called once after successful mint.

Attacker can buy nfts at lower prices by calling editListing()

Acknowledged

Path

721.sol

Function

editListing()

Description

In contract 721, after the user lists nft for sale, an attacker can come and call the function editListing() and change the price to any lower amount and buy it simultaneously. The reason this issue occurs is that the editListing() function takes _userAddress as a parameter which is then checked with the seller address in the require statement bypassing the check.

POC

-

```
    function test_If_AnotherUserTriedToEditListingToBuyNFTForLowerPrice()
external {
        // User/Creator mints the nft
        vm.startPrank(user1);
        nft.safeMint();

        assertEq(nft.balanceOf(user1), 1);

        // User/Creator lists nft for sale
        nft.listNFT(1, 1 ether, "MATIC", user1);
        assertEq(nft.balanceOf(user1), 0);
        vm.stopPrank();

        // Attacker edits the price of nft to lower amount
        vm.prank(user2);
        nft.editListing(1, 0.0000001 ether, user1);

        // Attacker buys the nft at lower price
        vm.prank(user2);
        nft.buyNFT{ value: 0.0000001 ether }(1, "MATIC");

        assertEq(nft.balanceOf(user2), 1);
    }|
```

Recommendation

To resolve the issue the _userAddress can be removed and seller address can be checked with msg.sender.

Low Severity Issues

Extra eth won't be returned to the users

Acknowledged

Path

721.sol

Function

buyNFT()

Description

In contract 721, when user goes to buy nft by calling buyNFT() if extra amount is sent then it'll be distributed among creator, the seller and the protocol fees making loss of buyer as the require statement in buyNFT() function checks if the msg.value is greater than equal to the selling price. There is no calculation where it checks if the price is equal and transfers the remaining amount back to the buyer.

Recommendation

To resolve the issue, check if there is an extra amount sent while purchasing, and send the remaining to the caller.

```
uint256 amountToReturn;
if(msg.value > listing.price) {
    amountToReturn = msg.value - listing.price;
    payable(msg.sender).transfer(amountToReturn);
}
uint256 amountToTransfer = msg.value - amountToReturn;
```

Use call instead of transfer for native token transfer.

Acknowledged

Path

721.sol

Function

withdraw(), buyNFT()

Description

In contract 721, for withdrawing/transferring the native token transfer is used. Which has 2300 capped gas usage. With changing blockchain dynamics the gas usage can vary for the transfer of native tokens. Which can cause the transfer to fail.

Recommendation

To resolve the issue use call instead of transfer
(bool success,) = call{ value: msg.value }(“”);
require(success, “Transfer failed”);

NFTs can be minted without toyow 3rd party wallet

Acknowledged

Path

toyow1155.sol

Function

mint()

Description

In contract, toyow1155 in mint() function the users can mint the ERC1155 tokens. But through explorer such as polygonscan the malicious user can connect wallet and mint the nfts as there is no check or condition where it checks if the wallet is toyow provided or external. As per the toyow team there is no direct incentive in doing such a malicious thing. As the value of the nfts will remain in toyow ecosystem only.

POC

-

```
241     function mint(
242         address account1,
243         uint256 amount1,
244         string memory currency1
245     ) public payable nonReentrant {
246         require(id == 1, "Invalid token ID provided!");
247         if (
248             keccak256(abi.encodePacked(currency1)) ==
249             keccak256(abi.encodePacked("usd"))
250         ) {
251             address owner = owner();
252             require(
253                 msg.sender == owner,
254                 "Only the owner can create USD currency tokens"
255             );
256             require(
257                 totalSupply(id) + amount1 <= maxSupply,
258                 "Sorry we're minted out!"
259             );
260             // As this function is always called by the Owner wallet address we can prevent
261             _mint(account1, id, amount1, "");
262             emit minted(account1, amount1, currency1);
263         } else {
264             require(
265                 totalSupply(id) + amount1 <= maxSupply,
266                 "Sorry we're minted out!"
267             );
268             require(
269                 msg.value == publicPrice * amount1,
270                 "Please send the approriate transaction amount"
271             );
272             _mint(account1, id, amount1, "");
273             emit minted(account1, amount1, currency1);
274         }
275     }
```

Wrong string input can cause nft to be locked in contract

Acknowledged

Path

toyow1155.sol

Function

list()

Description

In contract, toyow1155 the list() function takes currency as string input. The input is taken from the front end either as “usd” or “ether/matic” (native tokens) from selected tokens in dropdown. This prevents the issue of wrong token input in some sense but in contract while listing it does not check if the string is “usd” or “ether/matic”. If the user inputs the wrong token string then while checking the require in buyUsd() function it’ll fail resulting in the token being locked completely.

POC

-

```
127     // create a lising function
        ftrace | funcSig
128     function list(
129         address account!,
130         uint256 amount!,
131         uint256 unitPrice!,
132         string memory currency!
133     ) public nonReentrant {
134         require(id == 1, "Invalid token ID provided!");
135         uint256 currentUserSupply = balanceOf(account!, id);
136         require(
137             currentUserSupply >= amount!,
138             "Market: insufficient token supply."
139         );
140         require(
141             currentUserSupply + amount! <= maxSupply,
142             "Market: insufficient token supply."
143         );
144         lister(account!, amount!, unitPrice!, currency!);
145     }
146
```

```
197     function buyCrypto(uint256 listingId!, uint256 amount!)
198         public
199         payable
200         nonReentrant
201     {
202         TokenListing memory listing = getListing(listingId!);
203         require(
204             keccak256(abi.encodePacked(listing.currency)) !=
205                 keccak256(abi.encodePacked("usd")),
206             "Market: incorrect purchase currency"
207         );
```

Recommendation

To resolve the issue the team can ensure that user won't be able to input any other value than buy/sell tokens or they can take boolean values instead of string which will make sure that if currency is usd then it'll be set to true otherwise false.

Informational Severity Issues

Use latest Openzeppelin libraries

Resolved

Path

Toyow1155.sol

Description

After thoroughly checking the contracts we saw that the contracts are using older OZ libraries. In new OZ 5.0 there have been some breaking changes and bug fixes from the previous version to resolve the issue.

POC

In the newer version of openzeppelin library, there have been some removal of utilities like SafeMath and Counter used in the Toyow contract. The Pausable and ReentrancyGuard files were moved to the utils folder in the new version from the security folder in older version.
https://github.com/OpenZeppelin/openzeppelin-contracts/releases

Recommendation

The libraries can be updated to the latest Openzeppelin version 5.0. It is important to note that integrating the latest version 5.0 will require using the _update function in replacement of the _beforeTokenTransfer function and redesigning the counter mechanism differently since the library does not provide for counters in the newer version.

Remove unused state variables

Acknowledged

Path

Toyow721.sol

Description

There are some variables in the contract that were declared as state variables but were not used anywhere in the contract.

Recommendation

It is recommended that these variables, if not necessary for any use, should be removed because they take a slot each in storage.

State variables that are certain not to change should be marked as constant to save gas

Resolved

Path

Toyow1155.sol

Description

There are some variables in the contract that were used across the contract flow but none of these variables is intended to be changed or updated within the contract. Marking these state variables as constant will save up some change.

POC

-

```
34        uint256 publicPrice = 0.0000001 ether;
35        uint256 public id = 1;
36        uint256 public maxSupply = 72727;
```

Recommendation

Mark these state variables as constant and capitalize the variables' names.

calculateMintPrice function was not used in the contract

Acknowledged

Path

https://github.com/XChain-Technologies/Contracts/blob/1e410c2b6840422a7502a159792cb53265716d44/Final%20contracts%20review/1155.sol#L111

Description

The calculateMintPrice function was created for the sole purpose of estimating the cost of a mint, based on the publicPrice and the amount to be minted. However, this function was not used within the contract after its creation. In line 260, the check ensures that the user is sending an amount of ether that equals the multiplication of the publicPrice and the amount which is what the function, calculateMintPrice, is meant to do.

POC

-

```
require(msg.value == publicPrice * amount, "Please send the approriate transaction amount");
```

Recommendation

Replace 'publicPrice * amount' with the calculateMintPrice function which was not called at all within the contract.

Remove check that does not scrutinize input parameters with an important check of amount not being zero

Acknowledged

Path

https://github.com/XChain-Technologies/Contracts/blob/1e410c2b6840422a7502a159792cb53265716d44/Final%20contracts%20review/1155.sol#L237

https://github.com/XChain-Technologies/Contracts/blob/1e410c2b6840422a7502a159792cb53265716d44/Final%20contracts%20review/1155.sol#L131

Description

There is a “require” check present in the list and mint functions affirming that id is equal to 1. However, what makes this check unnecessary in the function is because the function is not designed to allow users to provide the id. If the user was at liberty of providing the id, it would have been pertinent to validate that input parameter. Removing this check saves the contract over 120 gas. It is important to design the functions to validate the input parameters instead like currency, amount, etc.

POC

require(id == 1,"Invalid token ID provides!");

Add:

require(amount ↑ != 0, "Amount can't be zero");

Recommendation

Remove unnecessary check in functions and add checks instead that validate input parameters. A very important check to add is that the amount is not equal to zero. This check should reflect on the following functions:
a. list
b. buyUsd
c. buyCrypto
d. mint

Floating Solidity Version

Acknowledged

Path

Toyow721.sol, Toyow1155.sol

Description

Contract has a floating solidity pragma version. This is present also in inherited contracts. Locking the pragma helps to ensure that the contract does not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively. The recent solidity pragma version also possesses its own unique bugs.

Recommendation

Making the contract use a stable solidity pragma version prevents bugs occurrence that could be ushered in by prospective versions. It is recommended, therefore, to use a fixed solidity pragma version while deploying to avoid deployment with versions that could expose the contract to attack.

Gas wastage for successful call of withdraw function when contract balance is zero

Acknowledged

Path

Toyow721.sol

Description

The withdraw function is a privileged function that can only be called by the contract owner to withdraw all of the native token accumulated in the contract from all purchases through the buyNFT function. However, if the contract has a balance of zero when the contract owner calls the function, the transaction is successful as well causing a wastage of gas with no native token to withdraw.

Recommendation

Add a check ensuring that the balance of the contract is not equal to zero. This way, anytime it is called when the balance of the contract is zero, it reverts instead of allowing the contract owner to call it successful without getting any token.

General Recommendation

Acknowledged

Description

The Toyow team is integrating a third party wallet into their platform where these ERC1155 tokens would be traded. It is important to highlight that the team intends to use a different provider that handles all USD transactions and allows for the platform owner to perform the blockchain calls. There is a dedicated database that tracks every blockchain call, validates USD transactions and correspondingly, the platform owner calls the needed functions from the smart contract.
With the integration of Pausable and Ownable contracts, the platform owner has the privileges of calling some unique functions in the contract that could pause and unpause the transfer of these tokens. This is in relation to the whenNotPaused modifier added to the _beforeTokenTransfer function. This is added to aid security but it is important to highlight these privileges.

The contract which handles 721 tokens has public safeMint() function which on deployment can be exploited by the attackers and mint nfts for free. The probability of this happening is pretty low however it becomes an issue when the maximum tokens aren't minted immediately.

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

Some issues of High, Low and Informational severity were found. Toyow Team acknowledged few and resolved others

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

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

7+ Years of Expertise	**1M+** Lines of Code Audited
$30B+ Secured in Digital Assets	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

January 2024

For

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com