QuillAudits

Audit Report
October, 2022

For

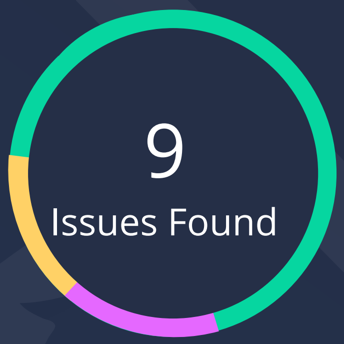
Nefta

Table of Content

Table of Content

Executive Summary

Project Name	Nefta Digital Asset Contract
Overview	The given contract Nefta Digital Asset is used to store, process, and auction the NFTs to the users. NeftaDigitalAsset.sol is a modified version of the ERC721 standard, adding commission/royalty payouts & auctions.
Timeline	07/09/2022 to 03/10/2022
Method	Manual Review, Functional Testing, Automated Testing etc.
Scope of Audit	The scope of this audit was to analyze Nefta Digital Asset Contract codebase for quality, security, and correctness. Zip File provided by Nefta Team
Fixed In	https://bitbucket.org/gwillink-nefta/nefta-smart-contracts/src/master/NeftaDigitalAsset.sol **Commit hash:** e68d861



🟥 High 🟨 Medium

🟩 Low 🟪 Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	0	**1**	**1**
Partially Resolved Issues	0	0	0	0
Resolved Issues	0	**2**	**4**	**1**

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

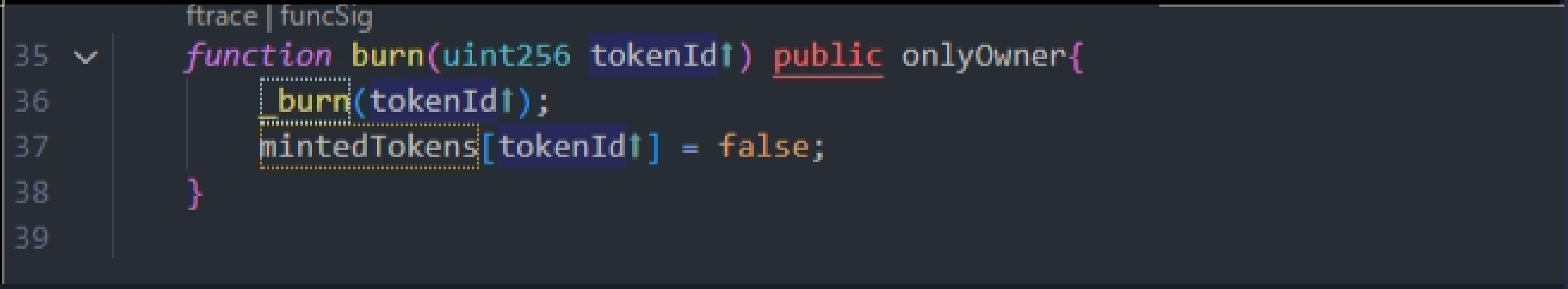
A. Contract - Nefta Digital Asset

High Severity Issues

No issues found

Medium Severity Issues

A.1 Centralization Power of the Admin

Line	Function - burn()
35-39	

```
        ftrace | funcSig
35 ⌄    function burn(uint256 tokenId↑) public onlyOwner{
36          _burn(tokenId↑);
37          mintedTokens[tokenId↑] = false;
38      }
39
```

Description

The contract possesses the feature of centralization of power. This feature allows for admin/ malicious admin to burn all tokens of any user without their consent.

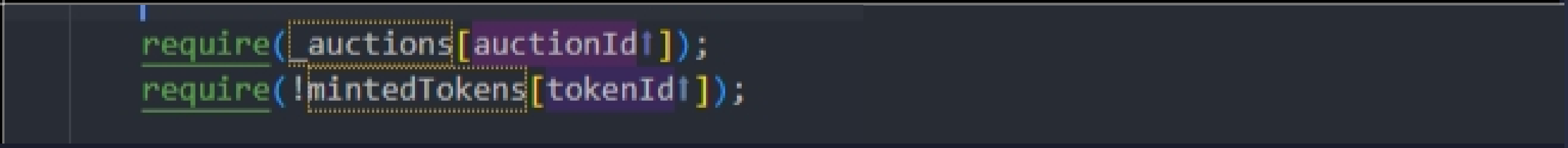
Remediation

To remediate the issue make sure that the token Id is not owned by any user before burning the token by the admin.

Status

Resolved

A.2 Possibility that the auction might not be set live.

Line	Function - createDigitalAssetFromAuctionEnd(), TransferDigitalAssetFromAuctionEnd()
48	

```
require(_auctions[auctionId]);
require(!mintedTokens[tokenId]);
```

Description

Whenever createDigitalAssetFromAuctionEnd(), TransferDigitalAssetFromAuctionEnd() functions are called it is possible that _auctions[auctionId] might not be set to true which can cause in failure of the function

Remediation

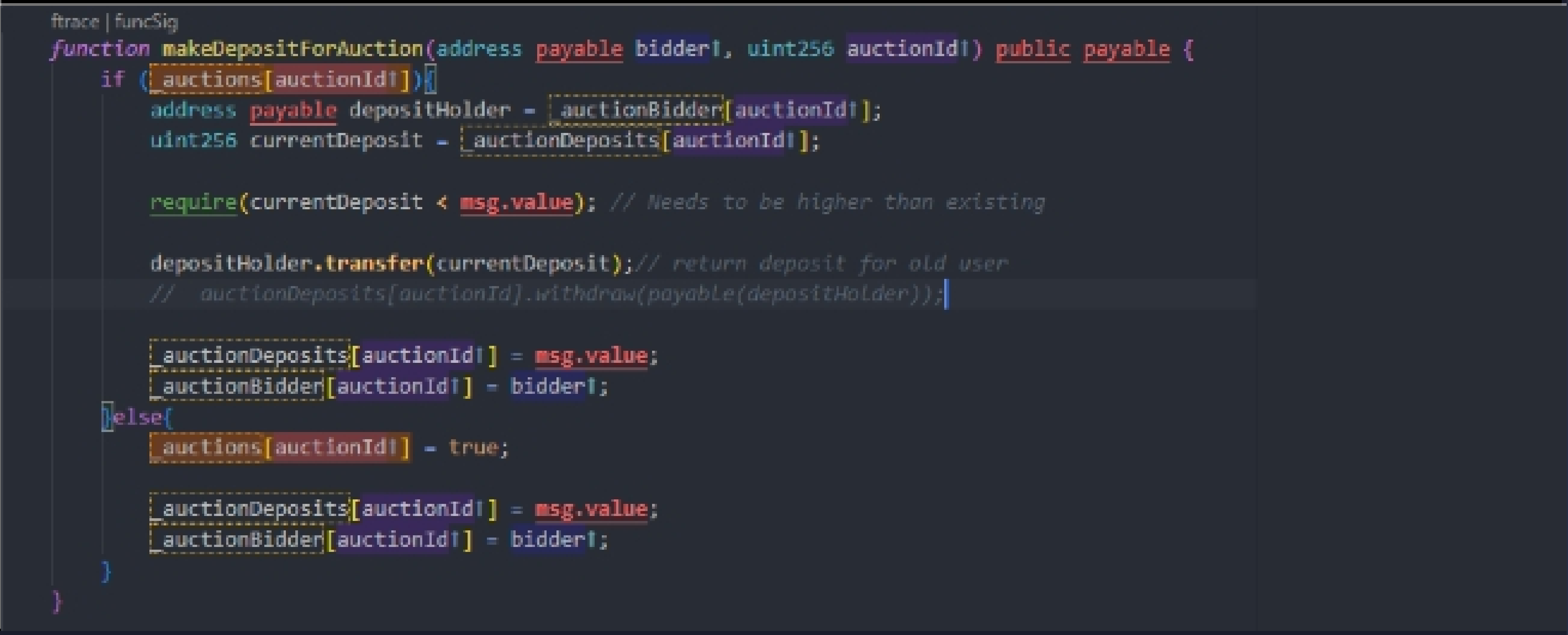
To remediate the issue you can set toggle like function which sets an auction on/off or set true in setAuctionForBuyNow.

Status

Resolved

Low Severity Issues

A.3 NFT from auction can be owned at lower price than bidding amount by users

Line	Function - makeDepositForAuction()
129-148	

```
ftrace | funcSig
function makeDepositForAuction(address payable bidder, uint256 auctionId) public payable {
    if (_auctions[auctionId]){
        address payable depositHolder = _auctionBidder[auctionId];
        uint256 currentDeposit = _auctionDeposits[auctionId];

        require(currentDeposit < msg.value); // Needs to be higher than existing

        depositHolder.transfer(currentDeposit);// return deposit for old user
        //  auctionDeposits[auctionId].withdraw(payable(depositHolder));

        _auctionDeposits[auctionId] = msg.value;
        _auctionBidder[auctionId] = bidder;
    }else{
        _auctions[auctionId] = true;

        _auctionDeposits[auctionId] = msg.value;
        _auctionBidder[auctionId] = bidder;
    }
}
```

Description

In makeDepositForAuction() after an auction is created when the first buyer tries to make a deposit for an tokenId he can only put 1 wei of amount and become highest bidder till someone else deposits an amount higher than that. Because when the first bidder bids an amount that time currentDeposit is always 0 so the comparison is 0 < 1 wei.

Remediation

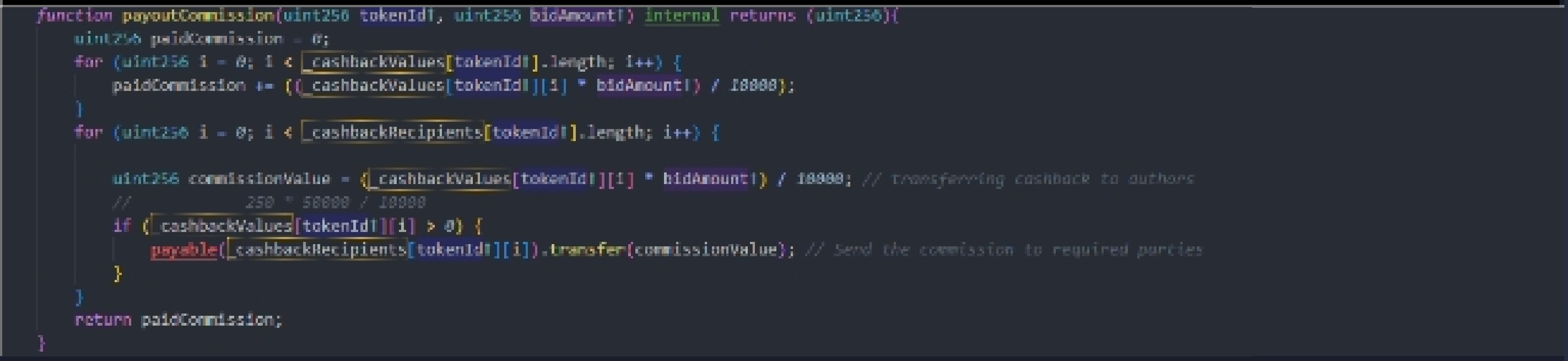
Make sure to update the values according to the deposit and add a check when the first user deposits the amount.

Status

Resolved

A.4 for loop used can cause out of gas issue/code doesn't affect anything in payoutCommission()

Line	Function - payoutCommission()
69-83	see code below



```
function payoutCommission(uint256 tokenId_, uint256 bidAmount_) internal returns (uint256){
    uint256 paidCommission = 0;
    for (uint256 i = 0; i < _cashbackValues[tokenId_].length; i++) {
        paidCommission += ((_cashbackValues[tokenId_][i] * bidAmount_) / 10000);
    }
    for (uint256 i = 0; i < _cashbackRecipients[tokenId_].length; i++) {

        uint256 commissionValue = (_cashbackValues[tokenId_][i] * bidAmount_) / 10000; // transferring cashback to authors
        //              250 * 50000 / 10000
        if (_cashbackValues[tokenId_][i] > 0) {
            payable(_cashbackRecipients[tokenId_][i]).transfer(commissionValue); // Send the commission to required parties
        }
    }
    return paidCommission;
}
```

Description

In payoutCommission() function different loops are used to calculate commission amount which can cause out of gas issues in some cases.

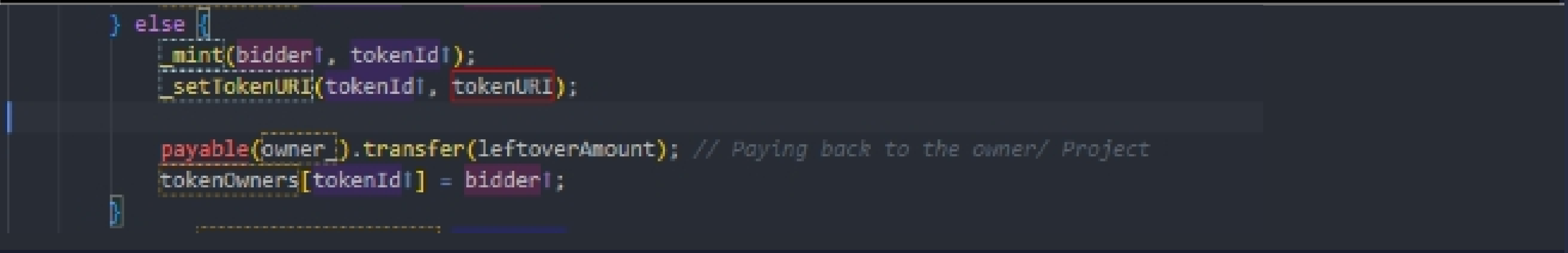
Remediation

As cashbackValues and cashbackRecipients mappings will always be the same length so the first loop can be removed.

Status

Resolved

A.5 State is not updated in buyNowActiveAuction() function

Line	Function - buyNowActiveAuction()
194-200	

```
} else {
    _mint(bidder, tokenId);
    _setTokenURI(tokenId, tokenURI);

    payable(owner_).transfer(leftoverAmount); // Paying back to the owner/ Project
    tokenOwners[tokenId] = bidder;
}
```

Description

In buyNowActiveAuction() function if tokenId is not minted then in else block it is minted to the bidder but mintedTokens[tokenId] = true;
is not set.

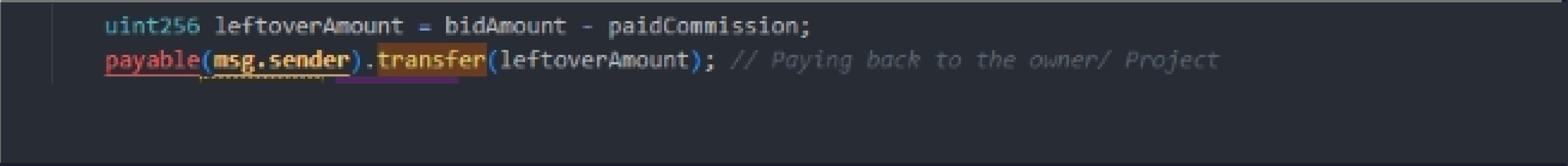
Remediation

Make sure to add mintedTokens[tokenId] = true in the else block after the _setTokenURI(tokenId, tokenURI)

Status

Resolved

A.6 transfer() is not recommended for eth transfer.

Line	Function - createDigitalAssetFromAuctionEnd(), payoutCommission(), TransferDigitalAssetFromAuctionEnd(), makeDepositForAuction(), buyNowActiveAuction()
65, 79, 99, 136, 191, 198	 `uint256 leftoverAmount = bidAmount - paidCommission;` `payable(msg.sender).transfer(leftoverAmount); // Paying back to the owner/ Project`

Description

In buyNowActiveAuction() function if tokenId is not minted then in else block it is minted to the bidder but mintedTokens[tokenId] = true;
is not set.

Remediation

Instead following is the correct way:
(bool success,) = _to.call{value: _amount}("");
require(success, "Failed to send Ether");

So as to mitigate possibility of reentrancy use check-interaction-effect pattern or use re-entrancy guard from openzeppelin
https://consensys.net/diligence/blog/2019/09/stop-using-soliditys-transfer-now/

Status

Resolved

A.7 Use of older versions of OpenZeppelin libraries.

Description

The contracts have been compiled using openzeppelin 3.x version which is an older version of openzeppelin. Using older versions of libraries is less secure and can have bugs which can be dangerous.

Remediation

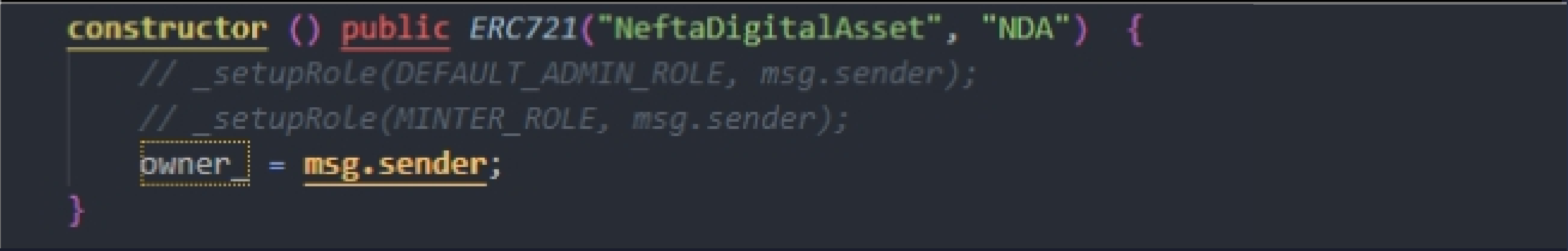
OpenZeppelin has been keeping their libraries updated quite fairly so it is strongly recommended to use the latest version of their libraries. The current latest version is 4.x

Status

Acknowleged

Low Severity Issues

A.8 Visibility of constructor can be ignored

Line	Function - constructor()
30-34	

```
constructor () public ERC721("NeftaDigitalAsset", "NDA")  {
    // _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
    // _setupRole(MINTER_ROLE, msg.sender);
    owner_ = msg.sender;
}
```

Description

Visibility (public / internal) is not needed for constructors anymore: To prevent a contract from being created, it can be marked abstract. This makes the visibility concept for constructors obsolete. This is taken from solidity docs

Remediation

So, if you're compiling your contract with Solidity version 0.7 or newer, the constructor visibility (in your case public) is ignored, and you can safely remove it.

Status

Resolved

A.9 Missing require statement error messages

Description

Not having error messages can make it harder to understand the issue which has caused the function to fail.

Remediation

In createDigitalAssetFromAuctionEnd(), transferDigitalAssetFromAuctionEnd() functions can have error message stating the issue which occurs if the condition fails such as, require(_auctions[auctionId], “Auction is not enabled”)

Status

Resolved

General Recommendation

In every function there are some state changes so it will be better to have events which expose these changes to blockchain.

If possible use Counters from openzeppelin to increment auctionId

Some functions can be set as view: - getCurrentBidAmount(), getBalanceOfSender()

Some functions can be set as external if not used in contract rather than public: getCurrentBidAmount(), getBalanceOfSender(), createDigitalAssetFromAuctionEnd(), TransferDigitalAssetFromAuctionEnd(), makeDepositForAuction(), buyNowActiveAuction()

Functional Testing

- ✓ Should Be Able To Return Address
- ✓ Should Be Able To Create Digital Asset
- ✓ Should Be Able To Create Digital Asset From Gift
- ✓ Should Be Able To Buy From Auction
- ✓ Token ID Should Not Be Able To Burn By Anyone Other Than Owner
- ✓ Should Be Able To Transfer Digital Asset From Auction End
- ✓ Should Be Able To Make Deposit For Auction

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 Nefta Digital Asset NFT Smart Contract. We performed our audit according to the procedure described above.

Some issues of 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.

Disclaimer

QuillAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the Nefta Digital Asset 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 Nefta 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
October, 2022

For

Nefta

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com