

Audit Report July, 2022

For



THE GAMBLE KINGDOM

Table of Content

Executive Summary	01
Checked Vulnerabilities	03
Techniques and Methods	04
Manual Testing	05
A. Contract - TGK/NFT	05
High Severity Issues	05
A.1 Wrong check of Total supply	05
Medium Severity Issues	06
A.2 Withdraw before RenounceOwnership	06
Low Severity Issues	07
A.3 Redundant Require check in refundIfOver	07
A.4 The events PublicSaleMinted and NftsAirdropped should be indexed	07
A.5 Missing require in owner update methods	08
A.6 Taking constructor arguments as input	09
Informational Issues	10
A.7 Use of loops	10
A.8 Enhanced teamMint functionality	10
A.9 Avoid use of multiple zeros	10



A.10	Following solidity naming conventions	11
A.11	Follow same pattern of transfer/call	12
A.12	Improper variable name in airdropNft	13
A.13	Functionality of teamMint	14
B. Contract - ERC721A		15
High Severity Issues		15
Medium Severity Issues		15
Low Severity Issues		15
Informational Issues		15
B.1	Downside of ERC721A	15
B.2	Unnecessary underflow/overflow checks	16
B.3	Variables declared anywhere in between	18
B.4	Use of _setOwnersExplicit	18
Functional Testing		19
Automated Testing		19
Closing Summary		20
About QuillAudits		21

Executive Summary

Project Name	TGK Contract Initial Audit Report
Overview	The project is a NFT contract based on the ERC721A standard with the added advantage of batch minting with lesser gas as compared to ERC721 standard. On top of that TGK contract also adds the functionality of air dropping of NFTs by the owner.
Timeline	14 June,2022 - 6 July, 2022
Method	Manual Review, Functional Testing, Automated Testing etc.
Scope of Audit	The scope of this audit was to analyze TGK codebase for quality, security, and correctness.
Deployed at	https://github.com/shrishtieth/TGK/blob/6a40f189280bea16572ac65a602870f58ac5dbf0/contracts/nft.sol
Fixed In	https://github.com/shrishtieth/TGK/commit/724123e9bd2cb02cd1370d1137055c74e0974bb2



	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	0	0	4
Partially Resolved Issues	0	1	0	1
Resolved Issues	1	0	4	6



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 - TGK/NFT

High Severity Issues

A1. Wrong check of Total supply

Line	Function - airdropNft
1329	<pre>1329 function airdropNft(address[] memory users, uint256[] memory quantity) 1330 external onlyOwner 1331 1332 { 1333 require(users.length == quantity.length,"Invalid Input"); 1334 uint256 total = quantity.length; 1335 require(1336 totalSupply() + total <= collectionSize, 1337 "Total Supply Reached" 1338); 1339 for (uint256 i = 0; i < total; i++) { 1340 _safeMint(users[i], quantity[i]); 1341 emit NftsAirdropped(users[i], quantity[i]); 1342 } 1343 } 1344 }</pre>

Description

The sum of all quantities in the quantity array should be used to check if total supply is reached and not the length of the array

Remediation

Inside the airdropNft, use a loop to calculate sum of all values in quantity array and check totalSupply() + sumOfQuantities <= collectionSize

Status

Resolved



Medium Severity Issues

A2. Withdraw before RenounceOwnership

Line	Function - renounceOwnership
1195	<pre>1188 /** 1189 * @dev Leaves the contract without owner. It will not be possible to call 1190 * `onlyOwner` functions anymore. Can only be called by the current owner. 1191 * 1192 * NOTE: Renouncing ownership will leave the contract without an owner, 1193 * thereby removing any functionality that is only available to the owner. 1194 */ 1195 function renounceOwnership() external virtual onlyOwner { 1196 _setOwner(address(0)); 1197 }</pre>

Description

If renounceOwnership is called then all the ether in the contract will be locked and become inaccessible

Remediation

Inside renounceOwnership before calling _setOwner, withdrawMoney method should be called

Status

Partially Resolved

Auditors Comment: There is no need to remove the renounceOwnership function as it is part of standard OZ Ownable contract. You can just override the renounceOwnership function and call moneyWithdraw in it.

Client's comment: We don't need the function



Low Severity Issues

A3. Redundant Require check in refundIfOver

Line	Function - refundIfOver
1305	<pre>1305 function refundIfOver(uint256 price) private { 1306 require(msg.value >= price, "Need to send more ETH."); 1307 if (msg.value > price) { 1308 payable(msg.sender).transfer(msg.value - price); 1309 } 1310 }</pre>

Description

The require statement at Line no. 1305 is not needed as the require statement in the function publicSaleMint does the same before calling refundIfOver.

Remediation

Removing the require statement at Line no. 1305

Status

Resolved

A4. Taking constructor arguments as input

Line	Constructor
1278	<pre>1278 constructor(1279 string memory baseURI 1280) ERC721A("Azuki", "AZUKI", 5, 5200) { 1281 _baseTokenURI = baseURI; 1282 } 1283</pre>

Description

The arguments of the constructor are hardcoded.

Remediation

Taking the values as user input and passing them to the constructor

Status

Resolved



A5. Missing require in owner update methods

Line	Function - airdropNft
1351	<pre>1351 function upadteMaxPerAddressDuringMint(uint256 max) external onlyOwner{ 1352 maxPerAddressDuringMint = max; 1353 emit MaxPerAddressDuringMintUpdated(max); 1354 }</pre>
Line	Function - upadteMaxBatchSize
1356	<pre>1356 function updateMaxBatchSize(uint256 size) external onlyOwner{ 1357 maxBatchSize = size; 1358 emit maxBatchSizeUpdated(size); 1359 }</pre>
Line	Function - upadteMaxBatchSize
1366	<pre>1366 function updateTeamNftStock(uint256 quantity) external onlyOwner{ 1367 amountForTeam = quantity; 1368 emit AmountForTeamUpdated(quantity); 1369 }</pre>

Description

There is no check(require statement) to ensure that these update values are less than the totalSupply.

Remediation

Using a require statement in all the functions

Status

Resolved

Auditor's Comment: The current implementation of require statements in updateMaxPerAddressDuringMint and updateMaxBatchSize is wrong In updateMaxPerAddressDuringMint,it is require(max + totalSupply() <= collectionSize,"Enter a valid value" but there is no use of totalSupply here. It is the maximum tokens an address can have In updateMaxBatchSize, it is require(size + totalSupply() <= collectionSize,"Enter a valid value") Here also there is no need of totalSupply.

A6. The events PublicSaleMinted and NftsAirdropped should be indexed

Line	Function - Event PublicSaleMinted & Event NftsAirdropped	
1271	1271	event PublicSaleMinted(address user, uint256 quantity);
	1273	event NftsAirdropped(address user, uint256 quantity);

Description

The indexed parameters for logged events will enable searching of these events using the indexed parameters as filters

Remediation

Using indexed user in both the events like
event PublicSaleMinted(address indexed user, uint256 quantity);
event NftsAirdropped(address indexed user, uint256 quantity);

Status

Resolved



Informational Issues

A7. Use of loops

Loops are used in teamMint and airDropNft methods. It is important to note that the looping has a limit in smart contracts. We cannot loop indefinitely as if we do that the gas consumed will be more than the block gas limit and the transaction will never be added in a block. Owner should perform a gas analysis before making these transactions.

Status

Acknowledged

A8. Avoid use of multiple zeros

Description

The public sale price is defined as 10000000000000000000 wei.

Recommendation

Using ether keyword and rewriting as 1 ether increases code readability

Status

Resolved

A9. Following solidity naming conventions

Description

Event names should start with capital letters

Recommendation

Modifying maxBatchSizeUpdated event to MaxBatchSizeUpdated

Status

Resolved



A10. Enhanced teamMint functionality

Line	Function - teamMint
1329	<pre>1312 // For marketing etc. 1313 function teamMint(uint256 quantity) external onlyOwner { 1314 require(1315 totalSupply() + quantity <= amountForTeam, 1316 "too many already minted before dev mint" 1317); 1318 require(1319 quantity % maxBatchSize == 0, 1320 "can only mint a multiple of the maxBatchSize" 1321); 1322 uint256 numChunks = quantity / maxBatchSize; 1323 for (uint256 i = 0; i < numChunks; i++) { 1324 _safeMint(msg.sender, maxBatchSize); 1325 } 1326 emit TeamMinted(quantity); 1327 }</pre>

Description

Currently, the teamMint function only mints when quantity is divisible by the maxBatchSize. It could be changed to mint any number of NFTs. If team minting should only be done in batches of maxBatchSize and is the required behavior please ignore this point

Remediation

By changing the functionality such that a loop runs (quantity / maxBatchSize) times calling _safeMint with maxBatchSize as quantity parameter and then calling _safeMint once again with quantity parameter as (quantity % maxBatchSize)

Status

Partially Resolved

Auditors comment: Please confirm functionality of teamMint method. The current code will not work, say maxBatchSize is 5 and you pass 13 as quantity. Do you want it to be minted in multiples of maxBatchSize like 5+5+3.



A11. Follow same pattern of transfer/call

Line	Function - withdrawMoney
1383	<pre>1383 function withdrawMoney() external onlyOwner nonReentrant { 1384 uint256 amount = address(this).balance; 1385 (bool success,) = msg.sender.call{value: amount}(""); 1386 require(success, "Transfer failed."); 1387 emit MoneyWithdrawn(amount); 1388 }</pre>
Line	Function - publicSaleMint
1285	<pre>1285 function publicSaleMint(uint256 quantity) 1286 external 1287 payable 1288 { 1289 require(1290 publicSaleStarted, 1291 "public sale has not begun yet" 1292); 1293 require(totalSupply() + quantity <= collectionSize, "reached max supply"); 1294 require(1295 numberMinted(msg.sender) + quantity <= maxPerAddressDuringMint, 1296 "can not mint this many" 1297); 1298 require(msg.value >= quantity * publicSalePrice, "Enter correct Amount"); // we 1299 _safeMint(msg.sender, quantity); 1300 refundIfOver(publicSalePrice * quantity); 1301 emit PublicSaleMinted(msg.sender, quantity); 1302 } 1303</pre>

Description

The withdrawMoney method uses call with reentrancy guard, and publicSaleMint function uses transfer. Both are safe options with no possibilities of a reentrancy attack but still for better code readability we would suggest using one of them at both the places.

Remediation

Using either transfer OR call with reentrancy guard, at both the places.

Status

Resolved



A12. Improper variable name in airdropNft

Line	Function - airdropNft
1329	<pre>1329 function airdropNft(address[] memory users, uint256[] memory quantity) 1330 external onlyOwner 1331 1332 { 1333 require(users.length == quantity.length,"Invalid Input"); 1334 uint256 total = quantity.length; 1335 require(1336 totalSupply() + total <= collectionSize, 1337 "Total Supply Reached" 1338); 1339 for (uint256 i = 0; i < total; i++) { 1340 _safeMint(users[i], quantity[i]); 1341 emit NftsAirdropped(users[i], quantity[i]); 1342 } 1343 } 1344 }</pre>

Description

The parameter array quantity should be renamed to quantities

Status

Resolved



A13. Functionality of teamMint

Line	Function - teamMint
1313	<div><div>1312</div><div>1313</div><div>1314</div><div>1315</div><div>1316</div><div>1317</div><div>1318</div><div>1319</div><div>1320</div><div>1321</div><div>1322</div><div>1323</div><div>1324</div><div>1325</div><div>1326</div><div>1327</div></div> <div><div>// For marketing etc.</div><div>function teamMint(uint256 quantity) external onlyOwner {</div><div>require(</div><div>totalSupply() + quantity <= amountForTeam,</div><div>"too many already minted before dev mint"</div><div>);</div><div>require(</div><div>quantity % maxBatchSize == 0,</div><div>"can only mint a multiple of the maxBatchSize"</div><div>);</div><div>uint256 numChunks = quantity / maxBatchSize;</div><div>for (uint256 i = 0; i < numChunks; i++) {</div><div>_safeMint(msg.sender, maxBatchSize);</div><div>}</div><div>emit TeamMinted(quantity);</div><div>}</div></div>

Description

Need to confirm that the owner won't be able to mint more NFTs for the dev team after the amountForTeam number of NFTs have been minted.

If this is the required behavior of the contract then why is there a updateTeamNftStock function, which can be used by the owner to update amountForTeam at any time.

Status

Acknowledged

Client's Comment: Owner can update it and mint more if he wishes



B. Contract - ERC721A

High Severity Issues

No issues found

Medium Severity Issues

No issues found

Low Severity Issues

No issues found

Informational Issues

B1. Downside of ERC721A

Description

Although ERC721A saves gas fees on minting but transferFrom and safeTransferFrom transactions cost more gas, which means it may cost more to gift or sell an ERC721A NFT after minting on secondary markets.

Status

Acknowledged

Client's Comment: We are aware of this, but want to have lesser gas fees while minting.



B2. Unnecessary underflow/overflow checks

Description

Each time a loop is run, underflow/overflow conditions are checked when incrementing/decrementing loop variables. In some cases there is no possibility of an underflow or overflow and in those cases using unchecked scoping to perform computations saves a significant amount of gas. The functions that can make use of unchecked keyword are:-

Line	Function - ownershipOf
726	<div><pre>726 for (uint256 curr = tokenId; curr >= lowestTokenToCheck; curr--) { 727 TokenOwnership memory ownership = _ownerships[curr]; 728 if (ownership.addr != address(0)) { 729 return ownership; 730 } 731 }</pre></div> <p>curr can never be lower than zero so loop at line 726 can be put in an unchecked scope.</p>
Line	Function - _safeMint
912	<div><pre>912 AddressData memory addressData = _addressData[to]; 913 _addressData[to] = AddressData(914 addressData.balance + uint128(quantity), 915 addressData.numberMinted + uint128(quantity) 916); 917 _ownerships[startTokenId] = TokenOwnership(to, uint64(block.timestamp)); 918 919 uint256 updatedIndex = startTokenId; 920 921 for (uint256 i = 0; i < quantity; i++) { 922 emit Transfer(address(0), to, updatedIndex); 923 require(924 _checkOnERC721Received(address(0), to, updatedIndex, _data), 925 "ERC721A: transfer to non ERC721Receiver implementer" 926); 927 updatedIndex++; 928 } 929 930 currentIndex = updatedIndex;</pre></div> <p>unchecked can be used from line 912 to line 930 as overflows are very unrealistic here because</p> <ul style="list-style-type: none">- At line 914, overflow will occur if, $\text{balance} + \text{quantity} > 3.4\text{e}38 \ (2^{128}) - 1$- At line 915, overflow will occur if, $\text{numberMinted} + \text{quantity} > 3.4\text{e}38 \ (2^{128}) - 1$- At line 921, overflow will occur if, $i > 1.56\text{e}77 \ (2^{256}) - 1$- At line 930, overflow will occur if, $\text{currentIndex} + \text{quantity} > 1.56\text{e}77 \ (2^{256}) - 1$



Line	Function - _transfer
971	<div><pre>971 _addressData[from].balance -= 1; 972 _addressData[to].balance += 1; 973 _ownerships[tokenId] = TokenOwnership(to, uint64(block.timestamp)); 974 975 // If the ownership slot of tokenId+1 is not explicitly set, that means the transfer initiator owns it. 976 // Set the slot of tokenId+1 explicitly in storage to maintain correctness for ownerOf(tokenId+1) calls. 977 uint256 nextTokenId = tokenId + 1; 978 if (_ownerships[nextTokenId].addr == address(0)) { 979 if (_exists(nextTokenId)) { 980 _ownerships[nextTokenId] = TokenOwnership(981 prevOwnership.addr, 982 prevOwnership.startTimestamp 983); 984 } 985 }</pre></div> <div><p>unchecked can be used from line 971 to line 985</p><p>At line 971,Underflow of the sender's balance is impossible because we check for ownership at line 951</p><p>At line 972,Overflow will occur if balance > 3.4e38 (2**128) - 1 which is unrealistic</p><p>At line 977, nextTokenId overflow is incredibly unrealistic as tokenId would have to be 2**256.</p></div>
<div><div>Status</div><div>Resolved</div></div>	



B3. Variables declared anywhere in between

Line	
1005	<div>1005uint256 public nextOwnerToExplicitlySet = 0;</div>
<div><div>Description</div><div>The variable nextOwnerToExplicitlySet should be declared at the top along with other variable declarations and not between functions</div><div>Remediation</div><div>Declaring nextOwnerToExplicitlySet at the top along with other functions</div><div>Status</div><div>Resolved</div></div>	

B4. Use of _setOwnersExplicit

Line	Function _setOwnersExplicit
1010	<div>1010function _setOwnersExplicit(uint256 quantity) internal { 1011 uint256 oldNextOwnerToSet = nextOwnerToExplicitlySet; 1012 require(quantity > 0, "quantity must be nonzero"); 1013 uint256 endIndex = oldNextOwnerToSet + quantity - 1; 1014 if (endIndex > collectionSize - 1) { 1015 endIndex = collectionSize - 1; 1016 } 1017 // We know if the last one in the group exists, all in the group exist, due to serial ordering. 1018 require(_exists(endIndex), "not enough minted yet for this cleanup"); 1019 for (uint256 i = oldNextOwnerToSet; i <= endIndex; i++) { 1020 if (_ownerships[i].addr == address(0)) { 1021 TokenOwnership memory ownership = ownershipOf(i); 1022 _ownerships[i] = TokenOwnership(1023 ownership.addr, 1024 ownership.startTimestamp 1025); 1026 } 1027 } 1028 nextOwnerToExplicitlySet = endIndex + 1; 1029 }</div>
<div><div>Description</div><div>_setOwnerExplicit function sets the owner in the _ownerships mapping. Need to confirm that this functionality is needed or not as this will cost the owner to update the _ownerships array which need not to be updated and is the whole point of using ERC721A over ERC721.</div><div>Status</div><div>Acknowledged</div></div>	

Functional Testing

Some of the tests performed are mentioned below

- ✓ Should be able to mint the token
- ✓ Should revert on minting token by sending less amount, minting when public sale isn't started, when maximum supply is reached or address minting more than allowed for one address
- ✓ Extra amount sent should be refunded after minting
- ✓ Owner should be able to airdrop NFTs to the entered addresses
- ✓ Should be able to transfer tokens
- ✓ Should be able to transfer ownership and revert for a zero address.
- ✓ Should be able to approve address for a specific token or all the tokens for an address
- ✓ Owner should be able to withdraw Ether from the contract
- ✓ Owner should be able to start and stop the minting functionality
- ✓ Owner should be set to 0 address after RenounceOwnership
- ✓ Should revert if token transferred to non ERC721Receiver Implementer
- ✓ Owner should get the contract's ether on using withdrawMoney
- ✓ Owner should be able to change the publicSalePrice, BaseURI, maxBatchSize, maxPerAddressDuringMint, teamNftStock

Automated Tests

No major issues were found after using Slither and Mythril. 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 TGK Nft contract. 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. At the end, The Gamble Kingdom Team Resolved most of the issues and Acknowledged others.

Disclaimer

Quillhash smart contract audit is not a security warranty, investment advice, or an endorsement of the The Gamble Kingdom 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 The Gamble Kingdom 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 July, 2022

For



THE GAMBLE KINGDOM



QuillAudits

📍 Canada, India, Singapore, United Kingdom

🌐 audits.quillhash.com

✉️ audits@quillhash.com