QuillAudits

AUDIT REPORT

June 2025

For

gigablocks

Table of Content

Table of Content

Executive Summary

Project Name Gigablocks

Protocol Type NFTs

Project URL https://beta-giga.rumsan.net/

Overview Gigablocks NFT Protocol for school connectivity projects, where users can mint and trade NFTs representing schools worldwide. The GigaMinter contract handles both free admin minting and paid public minting, collecting donations for school connectivity initiatives. The Escrow contract securely holds NFTs, managing reservations by email and facilitating transfers to school addresses and collectors. The GigaSeller contract implements a dynamic pricing mechanism with linear price increases based on demand for secondary market trading.

The Nft contract is an upgradeable ERC721 implementation that generates unique tokens for each school, while NftContent manages comprehensive metadata including school details like location, connectivity status, and electricity availability.

The ImageContent contract stores chunked base64 image data organized by geographic regions, enabling dynamic visual generation.

Finally, QOSGiga serves as a quality-of-service oracle system, allowing authorized managers to store Arweave hashes by date for tracking school connectivity metrics.

The entire Protocol operates under a two-tier NFT model: school NFTs for institutional ownership and collector NFTs for public trading, with robust access controls and pause mechanisms for secure operations.

Audit Scope The scope of this Audit was to analyze the Gigablocks Smart Contracts for quality, security, and correctness.

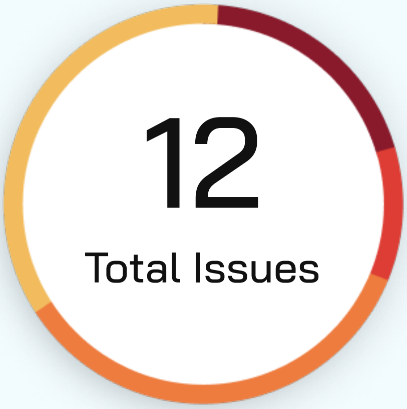
Source Code link https://github.com/giga-nft2-0/Giga_NFT_2.0-contracts/tree/main

Branch	Main
Contracts in Scope	-Escrow -GigaMinter -Image -Nft -NftContent -QOSGiga -GigaSeller
Commit Hash	9897c77ba9b064a6e8f43791897571934770f49d
Language	Polygon
Blockchain	Ethereum , Base
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	5th June 2025 - 16th June 2025
Updated Code Received	26th June 2025
Review 2	26th June 2025 - 1st July 2025
Fixed In	295e35f2551455977d2ffd51c5b5f9a5b03672ea

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



🟥	Critical	2 (16.7%)
🟥	High	1 (8.4%)
🟧	Medium	4 (33.3%)
🟨	Low	5 (41.6%)
🟪	Informational	0 (0%)

Severity

Issues

	🟥 Critical	🟥 High	🟧 Medium	🟨 Low	🟪 Informational
Open	0	0	0	0	0
Acknowledged	0	1	0	1	0
Partially Resolved	0	0	0	0	0
Resolved	2	0	4	4	0

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Users can lose previous donations through data override	Critical	Resolved
2	Buyers can bypass the fee escalation mechanism affecting protocol revenue	Critical	Resolved
3	Buyers can lose excess ETH due to the missing refund mechanism in buyNFT function	High	Acknowledged
4	Owner Can Override NFT Reservations for Email Holders.	Medium	Resolved
5	Incompatible Recipient Can Permanently Lock Reserved NFTs	Medium	Resolved
6	Owner Can Mint NFTs Without Paying Fees	Medium	Resolved
7	Initializer Vulnerability in UUPS Upgradeable Contract	Medium	Resolved
8	Donation Receiver Can Suffer Silent Transfer Failures Due to send() Usage	Low	Resolved
9	Unnecessary Multicall Inheritance in GigaMinter Contract	Low	Acknowledged

Issue No.	Issue Title	Severity	Status
10	Use Ownable2Step version rather than Ownable version	Low	Resolved
10	Outdated Pragma	Low	Resolved
12	Unused Event in Escrow Contract	Low	Resolved

Checked Vulnerabilities

- ☑ Access Management
- ☑ Arbitrary write to storage
- ☑ Centralization of control
- ☑ Ether theft
- ☑ Improper or missing events
- ☑ Logical issues and flaws
- ☑ Arithmetic Computations Correctness
- ☑ Race conditions/front running
- ☑ SWC Registry
- ☑ Re-entrancy
- ☑ Timestamp Dependence
- ☑ Gas Limit and Loops
- ☑ Exception Disorder
- ☑ Gasless Send
- ☑ Use of tx.origin
- ☑ Malicious libraries
- ☑ Compiler version not fixed
- ☑ Address hardcoded
- ☑ Divide before multiply
- ☑ Integer overflow/underflow
- ☑ ERC's conformance
- ☑ Dangerous strict equalities
- ☑ Tautology or contradiction
- ☑ Return values of low-level calls

- ☑ **Missing Zero Address Validation**
- ☑ **Private modifier**
- ☑ **Revert/require functions**
- ☑ **Multiple Sends**
- ☑ **Using suicide**
- ☑ **Using delegatecall**
- ☑ **Upgradeable safety**
- ☑ **Using throw**
- ☑ **Using inline assembly**
- ☑ **Style guide violation**
- ☑ **Unsafe type inference**
- ☑ **Implicit visibility level**

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code
- Use of best practices
- Code documentation and comments, match logic and expected behavior
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper
- Implementation of ERC standards
- Efficient use of gas
- Code is safe from re-entrancy and other vulnerabilities

The following techniques, methods, and tools were used to review all the smart contracts:

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms Used for Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity Static Analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

🟥 Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

🟥 High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

🟧 Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

🟨 Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

🟪 Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

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.

Severity Matrix

Impact

Likelihood	🟥 High	🟧 Medium	🟨 Low
🟥 High	Critical	High	Medium
🟧 Medium	High	Medium	Low
🟨 Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

Critical Severity Issues

Users can lose previous donations through data override when calling sendDonation multiple times

Resolved

Path

src/contracts/GigaMinter.sol

Function Name

sendDonation()

Description

The sendDonation() function in the GigaMinter contract contains a critical vulnerability that allows users' previous donation data to be completely overwritten when they call the function multiple times. This occurs due to the unconditional assignment of the bulkMinters mapping without any checks for existing data or accumulation logic.
The vulnerable code is located in the sendDonation() function:

```
function sendDonation(uint256 _schools) public payable nonReentrant {
    require(msg.value >= baseFee * _schools, “GigaMinter: Insufficient funds”);
    bulkMinters[msg.sender] = BulkMinter({
        totalDonation: msg.value,
        totalSchoolsToMint: _schools,
        minter: msg.sender,
        remainingSchoolsToMint: _schools
    });
    _transferDonation(msg.value);
}
```

The critical flaw lies in the direct assignment bulkMinters[msg.sender] = BulkMinter({...}) which completely replaces any existing BulkMinter struct for the same user address. This assignment operation does not check whether the user already has an existing donation record, nor does it accumulate the new donation with the previous one.

The vulnerability manifests in the following scenario: when a user calls sendDonation() for the first time, their donation is recorded correctly. However, if the same user calls sendDonation() again before all their entitled NFTs from the first donation have been minted, the second call overwrites the entire BulkMinter struct. This results in the loss of tracking for any remaining NFTs from the first donation, effectively causing the user to lose access to NFTs they have already paid for.

The mintForDonor() function, which is responsible for minting NFTs for donors, relies on the remainingSchoolsToMint field:

```
function mintForDonor(
    string memory _schoolId,
    address _schoolNftTo,
    address _collectorNftTo,
    string[9] memory _values
) public onlyOwner nonReentrant {
    require(bulkMinters[_collectorNftTo].remainingSchoolsToMint > 0,
“GigaMinter: No schools to mint”);
    uint256 schoolNftTokenId = _mintSchools(_schoolId, _schoolNftTo,
_collectorNftTo, _values);
    bulkMinters[_collectorNftTo].remainingSchoolsToMint -= 1;
    emit NFTMinted(schoolNftTokenId, _schoolNftTo, _collectorNftTo,0);
}
```

When the bulkMinters mapping is overwritten, the remainingSchoolsToMint count is reset to the new donation amount, completely losing track of the previous donation’s remaining entitlements.

Impact

Impact Level: High
Direct Financial Loss: Users who make multiple donations lose their entitlement to NFTs from previous donations that haven’t been fully minted yet. Since each NFT costs baseFee, users can lose significant amounts.

Likelihood

Likelihood Level: High
Users often make multiple donations over time, especially in cause-driven platforms.

Recommendation

Implement an accumulative donation system that preserves existing entitlements when users make additional donations

Specific Fixed In Commit

60d87a8cd5bd8e7cbf039fd9a009e29e627e8cd0

QuilAudits’s Response

This Issue has been fixed properly.

Buyers can bypass fee escalation mechanism affecting protocol revenue

Resolved

Path

src/contracts/GigaMinter.sol , src/contracts/GigaSeller.sol

Function Name

purchaseNft() , buyNft()

Description

The vulnerability stems from a fundamental conflict in the pricing mechanisms between the GigaMinter and GigaSeller contracts. While GigaSeller is designed with an escalating price model where NFT costs increase with each purchase, GigaMinter offers a fixed-price alternative that completely undermines this economic structure. This creates a loophole where buyers can bypass the intended fee escalation once prices in GigaSeller exceed GigaMinter's base fee.

The issue manifests as a classic arbitrage opportunity that worsens with protocol adoption. Initially, both contracts have comparable pricing, but as more NFTs are sold through GigaSeller, its prices rise according to the increment formula while GigaMinter's prices remain static. This growing price disparity creates an inevitable economic incentive for buyers to choose the cheaper GigaMinter option, effectively rendering GigaSeller's pricing model obsolete.

This architectural flaw has severe consequences for protocol economics. The escalating price mechanism in GigaSeller is meant to serve multiple purposes: generating increasing revenue for the protocol, creating artificial scarcity, and rewarding early adopters. By allowing buyers to circumvent this through GigaMinter, the protocol loses its primary value capture mechanism and long-term sustainability model.

Impact

The vulnerability has a high impact on the protocol's economic sustainability and fairness. Since GigaMinter allows users to mint NFTs at a fixed price while GigaSeller enforces an increasing price model, buyers will naturally prefer the cheaper option once the secondary market price exceeds the minting cost. This undermines the protocol's intended revenue generation, as fees from GigaSeller—which should grow with demand—are effectively bypassed. Over time, this leads to significant financial losses for the protocol, as most transactions will shift to GigaMinter, depriving the system of its designed fee escalation mechanism.

Likelihood

The likelihood of this issue being exploited is high, as it arises from basic economic incentives rather than a complex attack vector. Once the price difference between GigaMinter and GigaSeller becomes noticeable, rational buyers will automatically choose the cheaper option without requiring any malicious intent.

Recommendation

To ensure a consistent and sustainable economic model, the pricing logic between GigaMinter and GigaSeller must be fully aligned. Both contracts should enforce the same price mechanism.

Specific Fixed In Commit

295e35f2551455977d2ffd51c5b5f9a5b03672ea

QuillAudits team’s Response

This Issue has been fixed properly.

High Severity Issues

Buyers can lose excess ETH due to missing refund mechanism in buyNft function

Acknowledged

Path

src/contracts/GigaMinter.sol

Function

buyNft()

Description

The buyNft() function in the GigaMinter contract contains a critical flaw in its payment handling mechanism that results in the permanent loss of excess ETH sent by users. The vulnerability exists in the following code section:

```
function buyNft(
    string memory _schoolId,
    address _schoolNftTo,
    address _collectorNftTo,
    string[9] memory _values
) public payable nonReentrant {
    // @audit extra eth not transferred back confirm high
    require(msg.value >= baseFee, “GigaMinter: Insufficient funds”);
    uint256 schoolNftTokenId = _mintSchools(_schoolId, _schoolNftTo,
_collectorNftTo, _values);
    _transferDonation(msg.value); // This transfers ALL msg.value, not just
baseFee
    emit NFTMinted(schoolNftTokenId, _schoolNftTo, _collectorNftTo,msg.value);
}
```

The problematic logic lies in the _transferDonation(msg.value) call, which transfers the entire payment amount to the donation receiver without calculating or refunding any excess. The function only validates that the payment meets the minimum requirement using require(msg.value >= baseFee) but completely ignores the scenario where msg.value > baseFee.
The _transferDonation() internal function compounds this issue:

```
function _transferDonation(uint256 _amount) internal {
    totalDonations += _amount;
    bool transfer = donationReceiver.send(_amount);
    require(transfer, “GigaMinter: Donation Transfer failed”);
    emit DonationTransferred(msg.sender, _amount);
}
```

This function blindly transfers the full _amount parameter (which equals msg.value) to the donation receiver, treating any overpayment as an intended donation. The same vulnerability exists in the batchBuyNft() function, where users can overpay for multiple NFTs and lose even larger amounts.

Impact

Impact Level: High
Financial Loss: Users face immediate and irreversible financial loss when they accidentally send more ETH than required.

Likelihood

Likelihood Level: Medium
Common User Behavior: Overpayments are incredibly common in web3 interactions.

Recommendation

Implement an automatic refund mechanism in the buyNft() function to return excess ETH to users

QuillAudits team’s Response

This Issue has been acknowledged

Medium Severity Issues

Owner Can Override NFT Reservations for Email Holders

Resolved

Path

src/contracts/Escrow.sol

Function

reserveNft()

Description

The reserveNft function in the Escrow contract allows the contract owner to assign an NFT to a specific email address by updating the reservedNft mapping. However, the function does not enforce any checks to prevent the owner from overwriting an existing reservation. If an NFT has already been reserved for a given email, calling reserveNft again with the same email but a different tokenId will silently override the previous reservation without any warning or validation.

This issue arises because the function lacks a check to verify whether the email already has a reserved NFT. Additionally, the reservedNft mapping does not track historical reservations, making it impossible to detect or revert unauthorized overrides.

Incompatible Recipient Can Permanently Lock Reserved NFTs

Resolved

Path

src/contracts/Escrow.sol

Function

transfeReservedNft() , bulkTransferReservedNft()

Description

The transfeReservedNft and bulkTransferReservedNft functions in the Escrow contract use the standard transferFrom method from the ERC721 specification to send NFTs to recipient addresses. However, this method does not verify whether the recipient address is a contract that can properly handle ERC721 tokens. If the recipient is a non-receiving contract, the NFT transfer will succeed at the token level but the NFT may become permanently locked in the recipient's address.

This vulnerability arises because transferFrom does not check for onERC721Received callbacks, unlike safeTransferFrom which is specifically designed to prevent transfers to contracts that cannot manage NFTs. The issue affects all NFT transfer functions in the contract, including school NFT transfers, though the impact is most severe for reserved NFTs since they are tied to specific email claims.

Owner Can Mint NFTs Without Paying Fees

Resolved

Path

src/contracts/GigaMinter.sol

Function

batchMintNft()

Description

The batchMintNft function in the GigaMinter contract allows the owner to mint NFTs in bulk without paying the required baseFee, despite the function's NatSpec comment explicitly stating:

"The collector will pay the baseFee to mint the NFT"

This creates a critical inconsistency between the contract's documented behavior and its actual implementation. While regular users must pay baseFee when calling buyNft or batchBuyNft, the owner can bypass this requirement entirely via batchMintNft. The issue stems from:

- Missing fee validation in batchMintNft (unlike batchBuyNft, which enforces msg.value >= baseFee * _schoolIds.length)
- Misleading documentation that incorrectly suggests fees apply to all mints.

Initializer Vulnerability in UUPS Upgradeable Contract

Resolved

Path

src/contracts/NFT.sol

Function

constructor()

Description

The Nft contract inherits from OpenZeppelin's UUPS upgradeable pattern but fails to call _disableInitializers() in its constructor. This allows attackers to:

- Directly initialize the implementation contract – Since initialize() is public, anyone can call it on the implementation contract (not just through the proxy), setting themselves as the owner.

- Create a second owner – The proxy and implementation contract will have separate ownership states, allowing an attacker to bypass proxy-based access control.

- Bypass upgrade restrictions – The attacker could then upgrade the implementation contract maliciously, as _authorizeUpgrade checks onlyOwner but does not distinguish between proxy and implementation owners.

This violates the intended security model of UUPS proxies, where the implementation contract should be uninitializable except through the proxy.

Low Severity Issues

Donation Receiver Can Suffer Silent Transfer Failures Due to send() Usage

Resolved

Path

src/contracts/GigaMinter.sol

Function

_transferDonation()

Description

The _transferDonation function in the GigaMinter contract uses the deprecated send() method to transfer ETH to the donationReceiver. This is problematic because send():

- Has a fixed 2300 gas stipend, which may be insufficient if the receiver is a contract requiring more gas (e.g., for logic in receive() or fallback()).

The issue arises specifically in the donation handling logic, where send() is used despite modern best practices favoring call(). This affects all functions that process payments (buyNft, batchBuyNft, sendDonation).

Unnecessary Multicall Inheritance in GigaMinter Contract

Acknowledged

Path

src/contracts/GigaMinter.sol

Function

No function related to multicall

Description

The GigaMinter contract inherits from OpenZeppelin's Multicall utility, but this functionality is never used in the contract. Multicall is designed to allow batching multiple function calls into a single transaction, but:

- No Internal Usage: None of the contract's functions (buyNft, batchBuyNft, sendDonation, etc.) utilize multicall functionality.
- No External Integration: There is no clear indication that external systems (e.g., frontends or other contracts) require multicall support for this contract.

Inheriting unnecessary contracts increases deployment costs and could introduce complexity without providing value.

Use Ownable2Step version rather than Ownable version

Resolved

Path

src/contracts/Gigaminter , Gigaseller, Nftcontent

Function

constructor()

Description

Ownable2Step prevent the contract ownership from mistakenly being transferred to an address that cannot handle it (e.g. due to a typo in the address), by requiring that the recipient of the owner's permissions actively accept via a contract call of its own.

Consider using Ownable2Step from OpenZeppelin Contracts to enhance the security of your contract ownership management. This contract prevents the accidental transfer of ownership to an address that cannot handle it, such as due to a typo, by requiring the recipient of owner permissions to actively accept ownership via a contract call.

Outdated Pragma

Resolved

Description

The project uses pragma solidity 0.8.17. 0.8.17 is an outdated pragma version of Solidity.
Use pragma solidity 0.8.28.

Unused Event in Escrow Contract

Resolved

Path

src/contracts/Escrow.sol

Function

event NftReserved

Description

The Escrow contract declares an event NftReserved that is never emitted anywhere in the contract:

- `event NftReserved(string email, uint256 tokenId);`

While this event appears intended to log when NFTs are reserved via the reserveNft function, it is not actually used in the codebase. Instead, the contract only updates the reservedNft mapping without emitting this event.

Functional Tests

Some of the tests performed are mentioned below:

- ✓ Initial ownership should be set correctly
- ✓ Initial NFT address should be stored correctly
- ✓ Contract should accept ETH and emit Deposit event
- ✓ onERC721Received should emit ERC721Received event
- ✓ Only owner can call withdraw
- ✓ ETH withdrawal should transfer the correct amount
- ✓ Only owner can call reserveNft
- ✓ NFT reservation should store correct mapping
- ✓ Only owner can call transfeReservedNft
- ✓ Reserved NFT transfer should fail if token is not reservedOnly owner can call bulkTransferReservedNft
- ✓ Bulk transfer fails on length mismatch between _tos and _holderemails
- ✓ Successful bulk transfer updates claimed state and emits correct events
- ✓ Only owner can call transferSchoolNft
- ✓ School NFT transfer emits correct SchoolNftTransferred event
- ✓ Only owner can call bulkTransferSchoolNftOnly owner can call mintNft
- ✓ mintNft should mint school and collector NFTs with matching tokenId
- ✓ mintNft should emit NFTMinted event with correct parameters and zero donation
- ✓ Only owner can call batchMintNft
- ✓ batchMintNft should mint multiple NFTs correctlybuyNft should transfer donation to donationReceiver
- ✓ buyNft should fail if insufficient ETH is sent
- ✓ Excess ETH sent in buyNft is not refunded
- ✓ sendDonation should fail if donation < required baseFee * schools
- ✓ sendDonation should override previous donation from same user

- ✓ purchaseNft should transfer ETH to escrowContract if price is correct
- ✓ purchaseNft should fail if msg.value != calculated price
- ✓ purchaseNft should increment number of NFTs sold
- ✓ Only owner can call updateInitialPrice
- ✓ Only owner can call updateIncrement
- ✓ addImageChunk should store chunk data correctly
- ✓ addRegionImage should append image names to the region correctly
- ✓ Should mint the NFT properly
- ✓ Should updateNftContent properly
- ✓ Should updateNftImageHash properly

Threat Model

Contract	Function	Threats
Escrow	constructor	Zero-address parameters, Centralised ownership risk
	onERC721Received	Unrestricted receiver
	withdraw	Full ETH drain if owner compromised , No zero-address / balance checks
	reserveNft	Silent overwrite of existing reservation
	transfeReservedNft	transferFrom (not safeTransferFrom) bypasses receiver checks , No delete from reservedNft
	bulkTransferReservedNft	Gas exhaution
GigaMinter	mintNft	Owner-only unlimited minting , No donation check
	buyNft	Over-payment kept , No anti-bot / caps
	batchBuyNft	Gas-limit exhaustion
	sendDonation	Donation overwrite
	_transferDonation	Uses .send (2300 gas limit)—may fail for smart-contract receivers
	_mintSchools	Assumes identical tokenId from two different NFT contracts

Contract	Function	Threats
GigaSeller	purchaseNft	Static pricing logic
Nft	safeMint	External dependency (nftContent) for ID/hash generation
NftContent	updateNftContent	Content managers can update any NFT metadata arbitrarily
QOSGiga	setOracleManager	Can grant data push privileges to malicious actors
	addHashes	Data injection risk; malicious or spammy content could be stored

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 GigaBlock. We performed our audit according to the procedure described above.

Issues of Critical,High , Medium, and low severity were found. GigaBlock Team acknowledged two and resolved the others

Disclaimer

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

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

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

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With six years of expertise, we've secured over 1000 projects globally, averting over $30 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

QuillAudits

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

Follow Our Journey

AUDIT REPORT

June 2025

For

gigablocks

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com