QuillAudits

Audit Report
May, 2024

For

GANGSTER ARENA

Table of Content

Table of Content

Executive Summary

Project Name	Gangster Arena
Overview	Gangster Arena is a gamefi ecosystem that rewards users for gaming. GREED is the native token for Gangster Arena and it is an upgradable ERC20 contract. The NFT utility integrated into the gaming system is an ERC1155 token and this serves as the gangster for the arena. With the Gangster Arena contract, gangsters and assets (safehouse and goon) can be purchased, gangsters can be deposited and withdrawn. The Blast Points and Gas API was implemented with every of the contracts.
Timeline	20th May 2024
Method	Manual Review, Functional Testing, Automated Testing, etc. All the raised flags were manually reviewed and re-tested to identify any false positives.
Audit Scope	This audit aimed to analyze the Gangster Arena Codebase for quality, security, and correctness. 1. Gangster Arena.sol 2. GREED.sol 3. Gangster.sol 4. Minter.sol
Source Code	*https://github.com/wearedayone/GangsterArena2*
Branch	Main
Commit Hash	0a1c16096c6f5691ae107f0a903d76ee01b360c3
Fixed In	1657a147777a9501b802260c72d10b0b32828549

Number of Security Issues per Severity

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	0	0	0
Partially Resolved Issues	0	0	0	0
Resolved Issues	2	2	3	4

Checked Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- ERC20 API violation
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

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

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC-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 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

Hardhat, Foundry.

Types of Severity

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

High Severity Issues

A high severity issue or vulnerability means your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impacts and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

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

Resolved

These issues were identified in the initial audit and 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.

High Severity Issues

1. Loss of ETH after calling a function due to flawed implementation

Path

GangsterArena.sol

Function

devWithdraw

```
/**
*****************************
*DEV withdraw
*****************************
*/
//@audit The portion for the dev value will not be irredemable because of function implementation
ftrace | funcSig
function devWithdraw() public {
  require(devValue > 0, 'Nothing to withdraw');
  require(devAddr_ != address(0), 'Dev is not set');
  devValue = 0;
  address payable receiver = payable(devAddr_);
  //@audit - 0 eth will be sent
  receiver.transfer(devValue);
}
```

Description

The devValue state variable accumulates the portion of ether that is to be sent to the devAddr. The issue of losing ether when calling the devWithdraw function is caused by setting the devValue state variable to zero before it is passed as the value of ether to send. This implies that zero ether will be sent to the devAddr, leaving the portion of the devValue overtime in the smart contract.

POC

```
ftrace | funcSig
function testDevValue() external {
    // set game config
    vm.startPrank(owner);
    arena.setGameConfig(10, 60, 50_00, 50_00, 10);
    arena.setDevAddr(devAddr);

    vm.deal(owner, 100 ether);
    // owner sends to the arena contract
    (bool success,) = address(arena).call{value: 50 ether}("");

    uint256 devValue = (10 * 50 ether) / 10000;
    assertEq(devValue, arena.devValue());

    // call function to send to dev address
    arena.devWithdraw();

    assertEq(arena.devValue(), 0);
    assertEq(devAddr.balance, 0);
    assertEq(address(arena).balance, 50 ether);
}
```

Recommendation

The devValue should be cached with a local variable and used in place of the updated state variable.

Status

Resolved

2. Use of wrong values will cause retire function to always revert

Path

GangsterArena.sol

Function

retire

```
//@audit how is the payout calculated before the worker role passes the parameter to this function?
ftrace | funcSig
function retire(address _to↑, uint256 _payout↑, uint256 _nonce↑, bytes memory _sig↑) public onlyRole(WORKER_ROLE) {
    require(!gameClosed, "Game is closed");
    bytes32 message = SignerLib.prefixed(keccak256(abi.encodePacked(msg.sender, _to↑, _payout↑, _nonce↑)));
    require(signer.verifyAddressSigner(message, _sig↑), "Invalid signature");
    // @audit gangster[_to] is set to zero at this point
    gangster[_to↑] = 0;
    address payable receiver = payable(_to↑);
    reputationPrize -= _payout↑;
    //@audit value being passed to burn function will be zero since gangster[_to] is zero from above
    //@audit pass the tokenID constant istead of hardcoded value
    nft.burn(address(this), 1, gangster[_to↑]);
    receiver.transfer(_payout↑);
    emit Retire(_to↑, _payout↑, _nonce↑);      You, 2 hours ago • Uncommitted changes
}
```

Description

The retire function will revert anytime it is invoked. The gangster[_to] value passed into the nft burn function called within the retire function is 0 already because it was updated to 0 before it was passed to the burn function.

Recommendation

Cache the value of gangster[_to] with a local variable and use this in the burn function. Also use the state variable tokenID instead of hardcoded 1 value.

Status

Resolved

Medium Severity Issues

1. addReward function will always revert

Path

GangsterArena.sol

Function

addReward

```
ftrace | funcSig
function addReward(uint256 dev_, uint256 reputationPrize_, uint256 rankPrize_) public payable {
    require(msg.value == (dev_ + reputationPrize_ + rankPrize_), 'Need to send more ether'); //@audit
    devValue += dev_;
    reputationPrize += reputationPrize_;
    rankPrize += rankPrize_;
    emit AddReward(msg.sender, msg.value);
}
```

Description

The function addReward checks if the provided msg.value equals the sum of

1. dev
2. reputationPrize
3. rankPrize

However, it fails to consider a scenario where the user sends more value than the sum to compensate for high/volatile gas prices.

In that case, this function will revert due to strict equality being used in require function.

Recommendation

Instead of using “==” use “=>” and refund excess msg.value after the execution is completed.

Status

Resolved

2. Overflow will result in DOS

Path

GREED.sol

Function

batchMint

```
function batchMint(address[] calldata receivers↑, uint256[] calldata amounts↑) public onlyRole(MINTER_ROLE) {
    require(receivers↑.length == amounts↑.length);
    for (uint32 i = 0; i < receivers↑.length; i++) {
        _mint(receivers↑[i], amounts↑[i]);
    }
}
```

Description

The batchMint function in GREED.sol is designed to take a uint256 array of amounts. However, within this function, it uses a uint32 index for looping. This will revert when the loop forces the value of uint32 to overflow. This function is critical as it is used within the GangsterArena contract to batch mint tokens to players when the finalWarResult is called.

Recommendation

Change the index value to unit256.

Status

Resolved

Low Severity Issues

1. Missing address zero check

Path

GangsterArena.sol

Function

setWinner

```
/**
 * @notice update winners and payout
 */
//@audit missing address zero check
ftrace | funcSig
function setWinner(address[] memory to⬆, uint256[] memory points⬆) public onlyRole(WORKER_ROLE) {
    require(gameClosed && totalPoint > 0, "Game is not closed");
    require(address(this).balance > 0, "Nothing to withdraw");
    require(to⬆.length == points⬆.length, "Invalid input array length");
    uint256 totalPrize = reputationPrize + rankPrize;
    for (uint256 i = 0; i < to⬆.length; i++) {
        address payable receiver = payable(to⬆[i]);
        uint256 reward = (totalPrize * points⬆[i]) / totalPoint;
        require(address(this).balance >= reward, "Nothing to payout");
        if (reward > 0) receiver.transfer(reward);
    }
}
```

Description

There is no address zero check in the function to confirm that the caller does not provide parameters with null address. Ether will be sent to this address if there is an inclusion of address zero.

Recommendation

Add a zero address check.

Status

Resolved

2. Excessive gas usage when setting TokenMaxSupply

Path

GangsterArena.sol

Function

setTokenMaxSupply

```
/**
 * @notice set Base Price Gangster
 */
//@audit why set token max supply this way when there is only a single token id to mint?
ftrace | funcSig
function setTokenMaxSupply(uint256[] calldata _supply⬆) public onlyRole(ADMIN_ROLE) {
    for (uint256 i = 0; i < _supply⬆.length; i++) {
        tokenMaxSupply[i] = _supply⬆[i];
    }
}
```

Description

There is only one tokenId used within the Gangster Arena smart contracts which is token ID of 1. However, there is a function meant to set how much of the nft can be supplied into the Gangster Arena smart contract but this implementation will cost excessive gas usage to set the token ID of 1 to its maximum supply.

Recommendation

There is no need for a for-loop iteration since there is only one tokenID to set.

Status

Resolved

3. Unequal length of parameters will cause function to revert

Path

GangsterArena.sol

Function

finalWarResult

```
//@audit issue compare all three parameters length
ftrace | funcSig
function finalWarResult(address[] memory addr↑, uint256[] memory _lGang↑, uint256[] memory _wToken↑)
    public
    onlyRole(WORKER_ROLE)
{
    require(!gameClosed, "Game is closed");
    require(addr↑.length == _lGang↑.length, "Input array is not match");
    for (uint256 i = 0; i < addr↑.length; i++) {
        require(gangster[addr↑[i]] >= _lGang↑[i], "Invalid amount to burn");
    }

    uint256 total = reduce(_lGang↑);
    for (uint256 i = 0; i < addr↑.length; i++) {
        gangster[addr↑[i]] -= _lGang↑[i];
    }
    pointToken.batchMint(addr↑, _wToken↑);
    nft.burn(address(this), tokenId, total);

    emit WarResult(addr↑, _lGang↑, _wToken↑);
}
```

Description

Although the function checks the length of two parameters, if the _wToken is lesser than the length of others, the function reverts.

Recommendation

Check the length of all input parameters are equal.

Status

Resolved

Informational Issues

1. There is no referral system integrated into the arena

Path

GangsterArena.sol

Variable

refReward_

Description

refReward_ is a state variable that holds the percentage for referral. This variable also gets sets with other game configurations yet there is no implementation of the referral system with the smart contract.

Recommendation

Remove this variable and a save storage slot.

Status

Resolved

2. Rename function name to depict its implementation

Path

GangsterArena.sol

Function

reduce

Description

This function sums up each element in an array yet the name is reduce.

Recommendation

Rename function name to tally with the comment.

Status

Resolved

3. Variables only set at the constructor should be marked as immutable

Path

GangsterArena.sol, Minter.sol

Variables

Nft, pointToken, signer, pointsOperator, gangster, signer

Description

State variables that are only set at the constructor level should be marked as immutable to avoid taking a storage slot.

Recommendation

Add immutable to the variable declaration.

Status

Resolved

4. Use call over transfer to send ethers

Description

In the whole contract, it adopts the transfer function to send ethers. One associated problem with using the transfer function is that it consumes 2300 gas to perform its execution and when the operation fails it throws an error. And this is often not recommended to be used for complex functions performing ether transfer.

Recommendation

Use the call function to send ethers and handle when ether transfer is successful or not.

Status

Resolved

General Recommendation

Gangster Arena employs a backend system that calculates offchain the cost of purchase of gangsters and other parameters that make up the signature. This backend system also lifts the heavy weight of users interacting with the game smart contract. There are some functions that interact with the mint function from the Gangster contract, like mintNFT in GangsterArena.sol and mint in Minter.sol, and are likely to fail if not called by the address assigned as the MINTER_ROLE.

Gangster Arena Team's Comment: Because some information such as cost of purchase will be calculated offchain from our backend. so we use sign message to verify that these input parameters are valid. Depend on the functions, we will have different signer, but most-likely, signMessage will contains all input params to validate.

As current game logic, we do not allow user call directly to smart contract, they will need the signature from game pass validation.

Functional Tests Cases

Some of the tests performed are mentioned below:

- Should add reward to the Gangster Arena contract after every swapBack
- Should depositNFT only when game is open and increase the gangsters of the address “to”
- Should only withdraw if not locked and caller has one or more gangsters
- Should successfully retire a ganger by burning NFT and sending payout
- Should dev address get ether when devWithdraw is triggered
- Should successfully set game configurations before and after by appropriate callers
- Should call finalWarResult to end game by minting tokens to address and burning all gangsters in the contract
- Should buy gangsters with signature and revert if not with the minter role
- Should buy assets with different type
- Should trigger the spin function and burn point tokens from the caller

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 Gangster Arena 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.In the End, Gangster Team resolved all Issues.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in Gangster Arena smart contracts. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of Gangster Arena smart contracts. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services. It is the responsibility of the Gangster Arena to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

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.

1000+
Audits Completed

$30B
Secured

1M+
Lines of Code Audited

Follow Our Journey

Audit Report
May, 2024

For

GANGSTER ARENA

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com