Gacha Security Review

Pashov Audit Group

Conducted by: piken, Udsen, ZeroTrust01, Dan Ogurtsov

January 27th 2025 - January 29th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **sleekcore/gacha** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Gacha

Gacha is a game with purchasing and claiming randomized rewards using an onchain ticketing system, where users buy tickets, and winners are determined through an entropy-based mechanism. It integrates Uniswap for token swaps, supports referral rewards, and executes automated payout calculations and claim validations.

5. Risk Classification

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. 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.

5.2. 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.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 3e352cadd3780cfaef3c65fbdb81e709a9a85d91

fixes review commit hash - fae67e9381765e7774ad14921579354846b5e444

Scope

The following smart contracts were in scope of the audit:

- `Gacha`
- `errors/Gacha`
- `events/Gacha`
- `GachaConfig`
- `GachaPools`
- `GachaStorage`
- `GachaTickets`
- `Pick`
- `Config`
- `Pool`
- `Referral`
- `Ticket`
- `GachaStorage`

7. Executive Summary

Over the course of the security review, piken, Udsen, ZeroTrust01, Dan Ogurtsov engaged with Gacha to review Gacha. In this period of time a total of **11** issues were uncovered.

Protocol Summary

Protocol Name	Gacha
Repository	https://github.com/sleekcore/gacha
Date	January 27th 2025 - January 29th 2025
Protocol Type	Game

Findings Count

Severity	Amount
Critical	1
High	1
Medium	1
Low	8
Total Findings	**11**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Incorrect reserve order in _swap()	Critical	Resolved
[H-01]	_swap() is vulnerable to sandwich attacks	High	Resolved
[M-01]	Improper handling of ERC20 transfer return value	Medium	Resolved
[L-01]	The excess fee has not been returned to the user	Low	Resolved
[L-02]	deadline in swapExactTokensForTokens() does not work	Low	Resolved
[L-03]	Modulo operator introduces bias during oddsBPS ratio selection	Low	Acknowledged
[L-04]	The redundant call to pool.oddsBPS.indexOf(ratio)	Low	Resolved
[L-05]	Missing referralClaimThreshold validation	Low	Resolved
[L-06]	Lack of referral fee tracking	Low	Acknowledged
[L-07]	Reducing protocol fees when purchasing tickets	Low	Acknowledged
[L-08]	The paymentToken should only be allowed to be set once	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Incorrect reserve order in `_swap()`

Severity

Impact: High

Likelihood: Medium

Description

In the `GachaTickets::_swap` function, reserves are retrieved from the UniswapV2Pair contract as follows:

```
address pair = factory.getPair($.paymentToken, token);
(uint256 wethReserve, uint256 tokenReserve, ) = IUniswapV2Pair(pair)
    .getReserves();
if (wethReserve == 0 || tokenReserve == 0) revert InvalidPair();
```

However, `there is no validation` to ensure that `wethReserve` and `tokenReserve` are correctly assigned based on the actual token pair order.

Uniswap V2 reserves `must be retrieved in a consistent order`:

```
(token0, token1) = (pair.token0(), pair.token1())
```

where `getReserves()` always returns `(reserve0, reserve1)`, and `token0 is always the smaller address`.

Due to the above issue following consequences can occur :

1. Incorrect Swap Calculation:

   - If the wrong reserves are used in:

```
uint256 maxTokens = uni.getAmountOut(cost, wethReserve, tokenReserve);
```

- The function may return a `wrong actualTokens value`, leading to incorrect token amounts received.

2. Potential Denial of Service (DoS):

- If the `minTokens` calculation is incorrect (is very high), a swap may `always revert`, making purchases impossible.

3. Silent Loss of Funds:

- The protocol may receive `fewer pool tokens than expected` from the swap if the `minToken` (slippage parameter) is extremely less (than the correct value), thus reducing the `lottery reward pool size`.

Recommendations

It is recommended to retrieve and validate token pair order before using reserves for the `minTokens` calculation.

Modify `_swap` to `always fetch reserves in the correct order`:

```
(address token0, ) = IUniswapV2Pair(pair).token0();
(uint256 reserve0, uint256 reserve1, ) = IUniswapV2Pair(pair).getReserves();

 uint256 wethReserve = token0 == $.paymentToken ? reserve0 : reserve1;
 uint256 tokenReserve = token0 == $.paymentToken ? reserve1 : reserve0;
```

By implementing the recommendation, the contract will `prevent incorrect token swaps and avoid potential DoS vulnerabilities`.

8.2. High Findings

[H-01] `_swap()` is vulnerable to sandwich attacks

Severity

Impact: High

Likelihood: High

Description

Anyone can buy a ticket from a specified pool, a certain payment token will be used to swap for meme tokens. The swapping process is implemented as below:

```
function _swap(
        address token,
        uint256 cost
    ) private returns (uint256 actualTokens) {
        Storage storage $ = _getOwnStorage();
        IUniswapV2Router01 uni = IUniswapV2Router01($.uniswapRouter);
        IUniswapV2Factory factory = IUniswapV2Factory($.uniswapFactory);

        address pair = factory.getPair($.paymentToken, token);
        (uint256 wethReserve, uint256 tokenReserve, ) = IUniswapV2Pair(pair)
            .getReserves();
        if (wethReserve == 0 || tokenReserve == 0) revert InvalidPair();

@>      uint256 maxTokens = uni.getAmountOut
//(cost, wethReserve, tokenReserve); // includes 0.3%
@>      uint256 minTokens = Math.mulDiv(maxTokens, 95, 100); // 5% slippage

        address[] memory path = new address[](2);
        path[0] = $.paymentToken;
        path[1] = token;

        IERC20($.paymentToken).approve($.uniswapRouter, cost);
        uint256[] memory amounts = uni.swapExactTokensForTokens(
            cost,
@>          minTokens,
            path,
            address(this),
            block.timestamp + 1
        );
        actualTokens = amounts[amounts.length - 1];
    }
```

Before swapping for meme tokens using `uni.swapExactTokensForTokens()`, a `minTokens` amount is calculated. This value serves as a slippage protection measure. However it is calculated dynamically, attackers can manipulate the price by front-running the meme token purchase, buying the token before the ticket purchase, and then selling it immediately after, profiting from the price movement.

Recommendations

`minTokens` should be calculated on the frontend and passed as an input parameter to `GachaTickets#purchase()`

8.3. Medium Findings

[M-01] Improper handling of ERC20 `transfer` return value

Severity

Impact: High

Likelihood: Low

Description

In the `GachaTickets::entropyCallback` function, the contract transfers, winnings to the recipient as follows:

```
if (token == pool.token) pool.tokenBalance -= amount;
if (amount > 0) IERC20(token).transfer(receiver, amount);
```

This function call has `two major issues`:

1. Non-Compliant ERC20 tokens can revert :

The contract assumes that all ERC20 tokens implement the standard `transfer` function with the following signature:

```
function transfer(address to, uint256 value) external returns (bool);
```

However, `some tokens do not return a bool value`, which means:

- If the transferred token does `not return a boolean value` in its `transfer function`, the `transfer` function call `will revert unexpectedly` because the `IERC20` expects a `return boolean` value, causing potential DoS risks.

2. Tokens that return `false` on failure are ignored :

Certain ERC20 tokens (e.g., `USDT`) do not revert on failure but `return false` instead.

Since the return value of `transfer` is `not checked`, if a transfer fails:

- The `pool token balance is updated`, but the user `never receives their winnings`.
- The function `continues execution`, assuming the transfer was successful.
- The value `pool.totalRedeemed += 1;` is updated incorrectly, leading to a `loss of funds` for the user.

The same issue occurs in multiple places where `IERC20` functions such as `approve` and `transferFrom` `do not check return values`, leading to potential security risks.

Recommendations

Hence it is recommended to use OpenZeppelin's SafeERC20 for transfers Replace `all direct ERC20 calls` with `SafeERC20` functions from OpenZeppelin:

```
import {SafeERC20} from "oz/token/ERC20/utils/SafeERC20.sol";
using SafeERC20 for IERC20;
```

Then modify the transfer logic:

```
if (amount > 0) IERC20(token).safeTransfer(receiver, amount);
```

SafeERC20 functions of the `OpenZeppelin` correctly handle the `return value check` as well as `non-standard-compliant tokens`.

8.4. Low Findings

[L-01] The excess fee has not been returned to the user

```
function claim(uint256 poolId) external payable nonReentrant {
        Storage storage $ = _getOwnStorage();

        Ticket storage ticket = $.tickets[poolId][msg.sender];
        if (ticket.purchased <= ticket.claimed) revert InsufficientBalance();
        IEntropy entropy = IEntropy($.entropy);
        address provider = entropy.getDefaultProvider();
        uint256 fee = entropy.getFee(provider);
@>        uint64 seqNo = entropy.requestWithCallback{value: fee}(
            provider,
            bytes32(poolId)
        );

        $.callbackReceiver[seqNo] = msg.sender;
        $.callbackPool[seqNo] = poolId;
        ticket.claimed += 1;

        emit GachaLib.ClaimStarted(poolId, msg.sender, seqNo);
    }
```

The contract does not provide an accurate fee for users to query. As a result, the msg.value sent by users during the claim() function may exceed the actual fee. This excess amount needs to be refunded to the user.

[L-02] `deadline` in `swapExactTokensForTokens()` does not work

`block.timestamp+1` is used as deadline for `uni.swapExactTokensForTokens()` when swapping for meme tokens:

```
uint256[] memory amounts = uni.swapExactTokensForTokens(
            cost,
            minTokens,
            path,
            address(this),
            block.timestamp + 1
        );
```

This is always true because the deadline verification in `UniswapV2Router01` is:

```
require(deadline >= block.timestamp, 'UniswapV2Router: EXPIRED');
```

`deadline` should be passed as an input parameter.

[L-03] Modulo operator introduces bias during `oddsBPS ratio` selection

The `GachaTickets::getPayout` function uses `modulo` operation with the `recieved random number` to generate randomness for the lottery draw. But the `pyth documentation` states the following regarding the usage of modulo operation :

> Notice that using the modulo operator can distort the distribution of random numbers if it's not a power of 2. This is negligible for small and medium ranges, but it can be noticeable for large ranges. For example, if you want to generate a random number between 1 and 52, the probability of having a value of 5 is approximately 10^-77 higher than the probability of having a value of 50 which is infinitesimal.

As per the documentation, the probability of having 5 has a drastic difference with the probability of having 50. Hence when the `pool.oddsBPS.pick("pt", entropy)` is called to select an `oddsBPS ratio` randomly, the `oddsBPS ratio` may be skewed towards a particular value due to uneven distribution of selection probabilities. In addition, the `oddsBPS array` range could be large enough to cause the `modulo` operation to be `skewed` towards a particular `oddsBPS ratios`.

Hence this could affect the randomness of the `payout` calculation for the winner.

```
function pick(
    uint16[] memory self,
    string memory prefix,
    bytes32 entropy
  )
    internal
    pure
    returns (uint16)
  {
    return self[uint256(keccak256(abi.encode(prefix, entropy))) % self.length];
  }
```

Hence when determining the `oddsBPS ratio` randomly, it is recommended to consider using a different method to mitigate the bias caused by `modulo operation`.

[L-04] The redundant call to `pool.oddsBPS.indexOf(ratio)`

The `GachaTickets::getPayout` function selects the `oddsBPS ratio` randomly by calling the `pool.oddsBPS.pick` function. The implementation is as follows:

```
function pick(
    uint16[] memory self,
    string memory prefix,
    bytes32 entropy
  )
    internal
    pure
    returns (uint16)
  {
    return self[uint256(keccak256(abi.encode(prefix, entropy))) % self.length];
  }
```

Once the `oddsBPS ratio` is selected the `getPayout` function calls the `pool.oddsBPS.indexOf(ratio)` to get the index of the `ratio value` in the `oddsBPS array`.

```
function indexOf(uint16[] memory self, uint16 value) internal pure returns
    (uint16) {
    for (uint16 i = 0; i < self.length; i++) {
      if (self[i] == value) return i;
    }
    revert("not found");
  }
```

As shown above, the array starting from the first element of the `oddsBPS array` iterates till the `ratio` matches the specific element of the array and then the index is returned as the `tier`. But the issue with this implementation is if there are two elements with the same `ratio` in the `oddsBPS array` (let's assume index 2 and 3) then the `indexOf()` function will always return the first occurrence which is index 2 whereas the `uint256(keccak256(abi.encode(prefix, entropy))) % self.length` computation would have returned index 3. Hence the wrong tier is returned in this scenario.

Hence it is recommended to return the `uint256(keccak256(abi.encode(prefix, entropy))) % self.length` values as the `tier` during the `pool.oddsBPS.pick` function call instead of calling the `pool.oddsBPS.indexOf(ratio)` function redundantly to retrieve the same index.

[L-05] Missing `referralClaimThreshold` validation

Users can claim their `referral rewards` accrued in the contract by calling the `GachaTickets::claimReferralFees` function :

```
function claimReferralFees() external nonReentrant {
    Storage storage $ = _getOwnStorage();
    Referral storage ref = $.referrals[msg.sender];

    uint256 amount = ref.awardedAmount - ref.claimedAmount;
    if (amount == 0) revert InsufficientBalance();

    ref.claimedAmount += amount;

    IERC20($.paymentToken).transferFrom($.feeWallet, msg.sender, amount);

    emit GachaLib.ClaimReferral(msg.sender, amount);
}
```

However, the `documented behavior states` that `referrals should only be able to claim their rewards once they hit a referralClaimThreshold`.

Despite this, the `referralClaimThreshold` parameter is set in the `GachaConfig::setConfig` function, but `it is never enforced` in the `GachaTickets::claimReferralFees` function during `referral reward` withdrawal.

Due to the above missing threshold implementation following consequences can occur :

1. Users can claim referral fees below the intended threshold**

   - Users can `repeatedly` claim small amounts, leading to `high transaction costs and inefficiencies`.

2. Increased On-Chain activity & gas costs :

- The `protocol may experience relatively higher gas costs` due to frequent, small-value claims.

3. Deviation from the documented behavior :

   - The `contract does not align with its intended and documented functionality`, which may lead to `user confusion or misaligned behavior`.

It is recommended to enforce the `referralClaimThreshold` before processing a claim.

Modify `claimReferralFees` function to `only allow claims if the amount exceeds referralClaimThreshold`:

```
function claimReferralFees() external nonReentrant {
    Storage storage $ = _getOwnStorage();
    Referral storage ref = $.referrals[msg.sender];

    uint256 amount = ref.awardedAmount - ref.claimedAmount;
    if (amount == 0) revert InsufficientBalance();

    // Enforce referral claim threshold
    require(amount >= $.referralClaimThreshold, "Claim amount below threshold");

    ref.claimedAmount += amount;

    IERC20($.paymentToken).transferFrom($.feeWallet, msg.sender, amount);

    emit GachaLib.ClaimReferral(msg.sender, amount);
}
```

[L-06] Lack of referral fee tracking

In the `GachaTickets::_purchase` function, `referral fees` are allocated if the `referralFee > 0`:

```
if (referralFee > 0) {
    Referral storage ref = $.referrals[referral];
    ref.tickets += amount;
    ref.awardedAmount += referralFee;
}
```

At the same time, `all collected fees (including referral fees)` are transferred to the `treasury wallet (feeWallet)` as shown below :

```
IERC20($.paymentToken).transferFrom(
    msg.sender,
    address($.feeWallet),
    totalFees
);
```

But this implementation has the following issues :

1. Total referral fees are not tracked :

   - The contract `does not store the total referral fee amount separately` from the `protocol fee`.
   - This means `there is no way to differentiate referral fees from protocol fees` inside the `feeWallet`.

2. Governance cannot accurately withdraw protocol fees :

   - Since `referral fees` are stored `per address` in the `referrals mapping` but `not as a total value`, governance cannot determine the actual `protocol fee balance` in the treasury.
   - If governance `withdraws all funds`, it may `accidentally withdraw referral fees` that should be reserved for referral rewards.

3. Lack of referral address tracking :

   - Referral addresses are stored in a `mapping`, making it `impossible to iterate over them`.
   - If governance wants to `calculate the total referral payments`, they cannot retrieve a list of addresses with referral claims.

Recommendations:

Hence it is recommended to introduce a `totalReferralFees` state variable. And modify the contract to `keep track of the total referral fees` accumulated in the treasury.

```
uint256 public totalReferralFees;
```

Whenever a `referral fee is added`, update `totalReferralFees`:

```
if (referralFee > 0) {
    Referral storage ref = $.referrals[referral];
    ref.tickets += amount;
    ref.awardedAmount += referralFee;

    totalReferralFees += referralFee; // Track total referral fees separately
}
```

[L-07] Reducing protocol fees when purchasing tickets

Users need to pay protocol fees when purchasing tickets. The protocol fees will be transferred to `feeWallet`:

```
IERC20($.paymentToken).transferFrom(
            msg.sender,
            address($.feeWallet),
            totalFees
        );
```

The referral fees is calculated as below:

```
uint256 referralFee = referral == address(0)
            ? 0
            : Math.mulDiv(totalFees, $.referralBPS, BPS);
```

It will be distributed to the specified referral as rewards:

```
if (referralFee > 0) {
            Referral storage ref = $.referrals[referral];
            ref.tickets += amount;
            ref.awardedAmount += referralFee;
        }
```

The referral can claim their rewards lately by calling `claimReferralFees()`:

```
function claimReferralFees() external nonReentrant {
        Storage storage $ = _getOwnStorage();
        Referral storage ref = $.referrals[msg.sender];

        uint256 amount = ref.awardedAmount - ref.claimedAmount;
        if (amount == 0) revert InsufficientBalance();

        ref.claimedAmount += amount;

        IERC20($.paymentToken).transferFrom($.feeWallet, msg.sender, amount);

        emit GachaLib.ClaimReferral(msg.sender, amount);
    }
```

As we can see, The same amount of protocol fees should be paid when buying a ticket from a specified pool. However, a portion of fees will be distributed to the referral as a reward when the referral is not `address(0)`.

Buyers can set the referral address to their own address (`msg.sender`) or any account they control, enabling them to claim the resulting referral rewards.

Recommendations:

Redesign the referral fees mechanism. E.g. Referral fees should be paid directly by buyers instead of being deducted from protocol fees:

```
function _purchase(
        uint256 poolId,
        uint16 amount,
        address referral
    ) private returns (uint256 actualTokens) {
        Storage storage $ = _getOwnStorage();
        Pool storage pool = $.pools[poolId];

        uint256 totalCost = amount * pool.ticketPrice;
        uint256 tokenCost = Math.mulDiv(totalCost, pool.memeRatioBPS, BPS);
        uint256 totalFees = Math.mulDiv(totalCost, $.feeBPS, BPS);
        uint256 referralFee = referral == address(0)
            ? 0
            : Math.mulDiv(totalFees, $.referralBPS, BPS);

        IERC20($.paymentToken).transferFrom(
            msg.sender,
            address(this),
            totalCost
        );
        IERC20($.paymentToken).transferFrom(
            msg.sender,
            address($.feeWallet),
-           totalFees
+           totalFees + referralFee
        );

        if (referralFee > 0) {
            Referral storage ref = $.referrals[referral];
            ref.tickets += amount;
            ref.awardedAmount += referralFee;
        }
```

[L-08] The paymentToken should only be allowed to be set once

```
function setConfig(Config calldata config) public {
        Storage storage $ = _getOwnStorage();
        if (msg.sender != $.owner) revert Unauthorized();

        if (config.uniswapRouter != address(0))
            $.uniswapRouter = config.uniswapRouter;
        if (config.uniswapFactory != address(0))
            $.uniswapFactory = config.uniswapFactory;
        if (config.owner != address(0)) $.owner = config.owner;
        if (config.feeWallet != address(0)) $.feeWallet = config.feeWallet;
        if (config.feeBPS != 0) $.feeBPS = config.feeBPS;
        if (config.paymentToken != address(0))
@>              $.paymentToken = config.paymentToken;
        if (config.entropy != address(0)) $.entropy = config.entropy;
        if (config.referralBPS != 0) $.referralBPS = config.referralBPS;
        if (config.referralClaimThreshold != 0) {
            $.referralClaimThreshold = config.referralClaimThreshold;
        }
}
```

Allowing the paymentToken to be set to different tokens could lead to financial losses. For instance, if the previous paymentToken was WETH and it is subsequently changed to USDC, the WETH in the pool would become irretrievable.

Consider a scenario where the unclaimed referral amount is 1e18. Initially, this would entitle the user to claim 1 WETH. However, after the change, the user could claim 1e12 USDC instead. This discrepancy arises because the value of the tokens is not equivalent, and the change in paymentToken could significantly affect the actual value of the claims.

Recommendation:

```
- if (config.paymentToken != address(0))
+ if (config.paymentToken != address(0)&&$.paymentToken==address(0))
            $.paymentToken = config.paymentToken;
```