Pashov Audit Group

Gemint Security Review

March 18th 2026 - March 21st 2026

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out 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. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: 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** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

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

4. About Gemint

Gemint is a gacha-style NFT card game where users purchase randomized card draws and compete in player-versus-player battles using their collected cards. The protocol relies on a backend operator for off-chain battle resolution, purchase settlement, NFT minting, buyback payments, and refunds.

5. Executive Summary

A time-boxed security review of the **gemint-io/gemint-contracts** repository was done by Pashov Audit Group, during which **merlinboii**, **Nikethan**, **Said**, **ValvesSecurity** engaged to review **Gemint**. A total of **6** issues were uncovered.

Protocol Summary	
Project Name	Gemint
Protocol Type	NFT gacha card game
Timeline	March 18th 2026 - March 21st 2026

Review commit hash:
- ccc60df81fd952d16fbdd4c998b945178625efc2 (gemint-io/gemint-contracts)

Fixes review commit hash:
- 8260888d32551d0148d7bb3347eec454d8b671d7 (gemint-io/gemint-contracts)

Scope

`GachaController.sol` `GemintNFT.sol`

6. Findings

Findings count

Severity	Amount
Medium	3
Low	3
Total findings	**6**

Summary of findings

ID	Title	Severity	Status
[M-01]	Nft transfers remain active during pause due to lack of `ERC721PausableUpgradeable`	Medium	Resolved
[M-02]	`addBots` function lacks replay protection allowing signature replay until limits	Medium	Resolved
[M-03]	CancelBattle does not decrement `withdrawableFees` allowing potential fund theft	Medium	Resolved
[L-01]	Cancel battle and settlement functions missing `whenNotPaused`	Low	Resolved
[L-02]	No battle-level settlement tracking or finalization state for battles	Low	Resolved
[L-03]	Settle-then-cancel race already-settled players also receive full refund	Low	Resolved

Medium findings

[M-01] Nft transfers remain active during pause due to lack of `ERC721PausableUpgradeable`

Severity

Impact: Medium

Likelihood: Medium

Description

GemintNFT inherits `PausableUpgradeable` separately from `ERC721EnumerableUpgradeable` but does not inherit `ERC721PausableUpgradeable` and does not override the `_update` hook to enforce pause. The `whenNotPaused` modifier is applied only to mint/burn/redeem functions. All ERC721 transfer functions remain operational during the pause.

NatSpec states that the NFT must pause on minting, burning, redeeming, along with transfers:

```
    /// @notice Pause all token transfers and minting
    function pause() external onlyRole(DEFAULT_ADMIN_ROLE) {
        _pause();
    }

    /// @notice Unpause token transfers and minting
    function unpause() external onlyRole(DEFAULT_ADMIN_ROLE) {
        _unpause();
    }
```

This creates an inconsistency in `GachaController`: when `GemintNFT` is paused, `settleBattlePlayer()` is blocked (it calls the paused `mintToUser()`), but `cancelBattle()` can still proceed and successfully execute `nftContract.transferFrom()` to return the loser prize NFT to the host — because ERC721 transfers are not subject to the pause. Battles cannot be settled but can be partially canceled during a `GemintNFT` pause.

Recommendations

```
+ function _update(address to, uint256 tokenId, address auth)
+     internal override returns (address)
+ {
+     _requireNotPaused();
+     return super._update(to, tokenId, auth);
+ }
```

[M-02] `addBots` function lacks replay protection allowing signature replay until limits

Severity

Impact: Medium

Likelihood: Medium

Description

`addBots()` has no `processed` mapping or idempotency check. The same valid oracle signature can be replayed until the battle reaches `playerLimit`. Each replay pulls `paymentAmount` from the host via `safeTransferFrom`. Unlike `purchaseDraws` (which has `processed[purchaseIdHash]`), `joinBattle` (which has a duplicate participant check), and `createBattle` (which has `host != address(0)` check), `addBots` has zero replay protection.

```
function addBots(bytes calldata requestData, bytes calldata signature) external
whenNotPaused nonReentrant {
    _verifyOracleSignature(requestData, signature);
    AddBotsRequest memory request = abi.decode(requestData, (AddBotsRequest));
    // NO processed check -- compare with purchaseDraws:
    //   if (processed[purchaseIdHash]) revert PurchaseAlreadyProcessed(...);
    //   processed[purchaseIdHash] = true;
    IERC20(battle.paymentToken).safeTransferFrom(msg.sender, address(this),
request.paymentAmount);
    battlePayments[battleIdHash][msg.sender] += request.paymentAmount;
}
```

Exploit Scenario

1. Host creates a 4-player battle and pays 100 USDC.
2. Oracle signs `addBots(battleId, 1, 100_USDC, expiry)` for 1 bot.
3. Host submits `addBots` 2 times with the same signature. Each passes.
4. After 2 calls: 2 bots added, 200 USDC pulled. Battle full. `withdrawableFees += 300`
5. Now a legitimate participant comes and joins, paying 100 USDC, making `withdrawableFees += 300`.
6. Oracle intended for the host to pay 200 total. Host paid 300 and holds 3/4 participation.
7. Winning probability amplifies towards the host.

Recommendations

```
+ mapping(bytes32 => bool) public processedAddBots;

  function addBots(bytes calldata requestData, bytes calldata signature) external
whenNotPaused nonReentrant {
```

```
      _verifyOracleSignature(requestData, signature);
      AddBotsRequest memory request = abi.decode(requestData, (AddBotsRequest));
+     bytes32 requestHash = keccak256(requestData);
+     if (processedAddBots[requestHash]) revert AddBotsAlreadyProcessed();
+     processedAddBots[requestHash] = true;
      // ... rest of function
  }
```

Alternatively, add a unique `requestId` field to `AddBotsRequest` and reuse the existing `processed` mapping from `purchaseDraws()`: add `string requestId` as the first field of `AddBotsRequest`, then in `addBots()` after signature verification: `bytes32 requestIdHash = keccak256(bytes(request.requestId)); if (processed[requestIdHash]) revert PurchaseAlreadyProcessed(request.requestId); processed[requestIdHash] = true;` This mirrors the existing `purchaseDraws()` idempotency pattern and avoids introducing a separate mapping.

[M-03] CancelBattle does not decrement `withdrawableFees` allowing potential fund theft

Severity

Impact: Medium

Likelihood: Medium

Description

When a battle becomes full (via `joinBattle` or `addBots`), the total payment collected is immediately added to `withdrawableFees[battle.paymentToken]`. If the OPERATOR subsequently calls `cancelBattle`, all participants are refunded their `battlePayments`, but `withdrawableFees` is **never decremented**. This creates a phantom surplus: `withdrawableFees` records funds that have already left the contract.

There is a secondary dimension: `withdrawableFees` is incremented the instant a battle fills, before any settlement. The admin can call `withdrawFees()` at any time — even before the operator decides to settle or cancel. If the admin withdraws first and the battle is subsequently canceled, `cancelBattle` reverts on insufficient balance, permanently locking all participant funds.

```
// joinBattle (line 449-450) -- fee increment on fill
if (battle.playerCount == battle.playerLimit) {
    withdrawableFees[battle.paymentToken] += _battleTotalCollected(battleIdHash, battle);
}

// cancelBattle (lines 503-527) -- refunds WITHOUT decrement
function cancelBattle(string calldata battleId) external onlyRole(OPERATOR_ROLE) {
    battle.cancelled = true;
    for (uint256 i = 0; i < battle.participants.length;) {
        address participant = battle.participants[i];
        if (participant != address(0)) {
```

```
            IERC20(battle.paymentToken).safeTransfer(
                participant, battlePayments[battleIdHash][participant]
            );
        }
        unchecked { ++i; }
    }
    // NOTE: withdrawableFees is NEVER decremented
}
```

Exploit Scenario

1. Battle created: 2-player limit. Host pays 100 USDC. Player B joins, pays 100 USDC. Battle full. `withdrawableFees[USDC] += 200`.
2. User C calls `purchaseDraws`, pays 200 USDC. `withdrawableFees[USDC] = 400`. Contract balance = 400.
3. OPERATOR cancels battle. Host refunded 100, Player B refunded 100. Balance = 200. `withdrawableFees` is still 400.
4. Admin calls `withdrawFees(USDC, admin, 200)` -- succeeds. User C's 200 USDC drained.

Recommendations

```
  function cancelBattle(string calldata battleId) external onlyRole(OPERATOR_ROLE) {
      // ... existing checks ...
      battle.cancelled = true;

+     if (battle.playerCount == battle.playerLimit) {
+         withdrawableFees[battle.paymentToken] -= _battleTotalCollected(battleIdHash,
battle);
+     }

      for (uint256 i = 0; i < battle.participants.length;) {
          // ... refund loop ...
      }
  }
```

Additionally, consider not adding battle funds to `withdrawableFees` until the battle is fully settled. Track pending battle funds separately.

Low findings

[L-01] Cancel battle and settlement functions missing `whenNotPaused`

Description

Three operator-controlled functions, `cancelBattle`, `settleBattlePlayer`, and `transferLoserPrize`, lack the `whenNotPaused` modifier despite all other settlement-adjacent functions in the contract being guarded.

When the admin invokes `pause()` in an emergency, the intent is to halt all fund-moving operations. However, during the pause window, the operator can still cancel battles (triggering ERC20 refund transfers), settle players (triggering NFT mints and ERC20 buyback transfers), and transfer loser prizes (triggering NFT transfers).

The inconsistency is compounded by the fact that `purchaseDraws`, `createBattle`, `joinBattle`, `addBots`, `transferNFT`, `buyback`, `buybackMinted`, and `buybackMintedBattle` all carry `whenNotPaused`, while these three functions do not.

[L-02] No battle-level settlement tracking or finalization state for battles

Description

The battle struct tracks per-player settlement via `playerSettled[battleIdHash][player]`, but has no battle-level settlement counter (`settledCount`) or finalization flag (`battleSettled`). This means:

1. After all real players are settled, the battle remains in the same state as a partially settled battle. There is no transition to a finalized state.
2. Without a `settledCount > 0` check, `cancelBattle()` has no efficient way to know if settlement has begun.

Recommendation: Add a `settledCount` field to the `Battle` struct and increment it in `settleBattlePlayer()`:

```
// In Battle struct:
uint8 settledCount;

// In settleBattlePlayer(), after marking player settled:
if (battle.settledCount > 0) revert BattleAlreadyPartiallySettled(battleId);
playerSettled[battleIdHash][player] = true;
battle.settledCount++;
```

```
// In cancelBattle(), add O(1) check:
if (battle.settledCount > 0) revert BattleAlreadyPartiallySettled(battleId);
```

This is cheaper than looping through all participants to check `playerSettled` (O(n) → O(1)) and provides on-chain visibility into settlement progress.

[L-03] Settle-then-cancel race already-settled players also receive full refund

Description

`cancelBattle`'s refund loop does not check `playerSettled`. A player who was already settled (received NFTs + buyback) also receives a full refund when the battle is subsequently cancelled. Both `settleBattlePlayer` and `cancelBattle` check `!battle.cancelled`, but when called sequentially by the operator, both pass. The refund loop has no guard to skip already settled players.

This is a known issue: both functions are operator-only, so exploitation requires the operator to intentionally sequence settle-then-cancel. The operator is a trusted backend service that would not perform this sequence in normal operation.

```
  for (uint256 i = 0; i < battle.participants.length;) {
      address participant = battle.participants[i];

-     if (participant != address(0)) {
+     if (participant != address(0) && !playerSettled[battleIdHash][participant]) {
          IERC20(battle.paymentToken).safeTransfer(
              participant, battlePayments[battleIdHash][participant]
          );
      }
      unchecked { ++i; }
  }
```