Pashov Audit Group

Rip It Security Review

Conducted by:

Hals
merlinboii
Ch_301
saksham
IvanFitro
afriauditor
Opeyemi

May 10th 2025 - May 17th 2025

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 Rip It

Rip It is a trading card platform where users buy, open, and trade NFT packs, with a built-in marketplace and spin-to-win mechanics. It uses a randomized card distribution system, USDC for trading, and a role-based access model to manage minting, rewards, and marketplace operations.

5. Executive Summary

A time-boxed security review of the **ripdotfun/rip-it-core** repository was done by Pashov Audit Group, during which **Hals**, **merlinboii**, **Ch_301**, **saksham**, **IvanFitro**, **afriauditor**, **Opeyemi** engaged to review **Rip It**. A total of **27** issues were uncovered.

Protocol Summary	
Project Name	Rip It
Protocol Type	Game
Timeline	May 10th 2025 - May 17th 2025

Review commit hash:
- 2c22ac3f742b3f3a27161d39e762fa4449c4c1cd (ripdotfun/rip-it-core)

Fixes review commit hash:
- 6e3f3fa04e18275071ffd896288d8dbf2d7f7f70 (ripdotfun/rip-it-core)

Scope

`Card` `CardStorage` `CardAllocationPool` `CardAllocationPoolStorage` `Marketplace` `MarketplaceAdmin` `MarketplaceStorage` `Packet` `PacketStorage` `PacketStore` `RoleBasedAccessControl` `RoleBasedAccessControlConsumer` `SpinLottery` `SpinLotteryStorage` `WhitelistRegistry` `WhitelistRegistryStorage` `IMarketplace.sol` `IRandomAllocationPool.sol`

6. Findings

Findings count

Severity	Amount
Critical	2
High	2
Medium	2
Low	21
Total findings	**27**

Summary of findings

ID	Title	Severity	Status
[C-01]	Reentrancy attack in `authorizedOpenPacket()` allows duplicates	Critical	Resolved
[C-02]	Missing packet ID in `finalizeOpen()` causes NFT loss	Critical	Resolved
[H-01]	Missing index updates in `burnPacketFromInventory()` cause failures	High	Resolved
[H-02]	Packet burn state not reset in `finalizeOpen()` causes failure	High	Resolved
[M-01]	Failure in `sendNFTs()` leaves NFTs stuck and causes asset loss	Medium	Resolved
[M-02]	Missing nonce in `LISTING_TYPEHASH` affects EIP-712 and signatures	Medium	Resolved
[L-01]	Cleanup missing for `packetToBundle[packetId]` in `burnFromInventory()`	Low	Resolved
[L-02]	Stuck fees in `PacketStore` upon `paymentToken` change without withdrawal	Low	Resolved
[L-03]	Missing validation for burned `packetIds` in `addCardBundlesToPacketPool()`	Low	Resolved
[L-04]	Missing packet type validation in `addCardBundlesToPacketPool()`	Low	Resolved

ID	Title	Severity	Status
[L-05]	Missing state cleanup for burned packets	Low	Resolved
[L-06]	Role ID collision between `PacketStore` and `WhitelistRegistry` contracts	Low	Resolved
[L-07]	`onERC721Received` missing blocks safe NFT transfers and mints	Low	Resolved
[L-08]	Missing offer revocation mechanism in Marketplace contract	Low	Resolved
[L-09]	Missing `chargeDelegate` parameter in signed listing data	Low	Resolved
[L-10]	Silent error swallowing due to early event	Low	Resolved
[L-11]	`revertBurnRequest()` mismanages `INSTANT_OPEN_PACKET` burn type	Low	Resolved
[L-12]	Risk of packet loss in `authorizedOpenPacket()` with empty cards	Low	Resolved
[L-13]	`requestConfirmations` below Chainlink minimum allowed in `UpdateVRFConfig()`	Low	Acknowledged
[L-14]	Prizes need upper bound to prevent `configureRarity` going OOG	Low	Resolved
[L-15]	Sellers face fee volatility without safeguard in `MarketPlace.buy()`	Low	Resolved
[L-16]	Incorrect packet burning mechanism in bundle allocation	Low	Acknowledged
[L-17]	Missing refund in `SpinLottery.sol` when prize distribution fails	Low	Resolved
[L-18]	Rarity configuration in SpinLottery leads to `pendingCount` errors	Low	Resolved
[L-19]	Revocation may fail after currency update allowing exploits	Low	Acknowledged
[L-20]	Token collection failure during update leads to mixed accounting	Low	Resolved
[L-21]	DoS vulnerability in spin due to pending prize	Low	Resolved

Critical findings

[C-01] Reentrancy attack in `authorizedOpenPacket()` allows duplicates

Severity

Impact: High

Likelihood: High

Description

There is a critical reentrancy vulnerability in the interaction between `CardAllocationPool.sol` and `Packet.sol` when handling **OPEN_PACKET** burn types. This vulnerability enables attackers to receive multiple sets of NFT cards for a single packet.

The attack scenario follows this sequence:

- Attacker initiates a burn with **OPEN_PACKET** type using `initiateBurn()` in `Packet.sol` for a packet they own.
- A manager calls `authorizedOpenPacket()` in `CardAllocationPool.sol` to open the packet with specific cards.
- The `authorizedOpenPacket()` function transfers cards to the attacker using `safeTransferFrom()`, which triggers the `onERC721Received()` hook on the attacker's contract.
- Within this callback, the attacker can: 1) Call `cancelBurn()` to reset the burn state of the packet. 2) Immediately call `initiateBurn()` again with **INSTANT_OPEN_PACKET** type for the same packet
- If `packetTypeToCardBundles[packetType].length == 1` in `CardAllocationPool.sol`, the `instantOpenPacket()` function will: 1) Transfer a set of NFTs cards to the attacker. 2) Call `finalizeOpen()` in `Packet.sol` to transfer the packet to the inventory.
- When control returns to the original `authorizedOpenPacket()` function, it calls `finalizeOpen()` to burn the packet, but at this point, the packet is already in the inventory, causing a state inconsistency.

This attack leads to:

a. The attacker receives cards twice for a single packet.

b. The packet is added to inventory for resale but is also burned, causing future calls to the `PacketStore.sol` contract to fail when attempting to transfer the packet.

The root cause is that the `authorizedOpenPacket()` function doesn't protect against reentrancy and the burn type is not reset to **NONE**, allowing the packet's state to be manipulated during the callback from the `safeTransferFrom()` operation.

Recommendations

Implement reentrancy protection and use a checks-effects-interactions pattern to prevent this attack in `Packet.sol` and `CardAllocationPool.sol` functions.

[C-02] Missing packet ID in `finalizeOpen()` causes NFT loss

Severity

Impact: High

Likelihood: High

Description

In the `Packet.sol` contract, the `finalizeOpen()` function has a critical flaw that will cause all transactions to fail when the `burnType` is **INSTANT_OPEN_PACKET**. When processing an **INSTANT_OPEN_PACKET** burn type, the function attempts to add the packet to inventory by calling `packetStore.addPacketsToInventory(packetIds)`. However, it creates a new array to hold the packet ID but never actually assigns the packet ID to the array:

```
if (burnType == BurnType.INSTANT_OPEN_PACKET) {
    _transfer(ownerOf(packetId), address(this), packetId);
    this.approve(address(packetStore), packetId);

    uint256[] memory packetIds = new uint256[](1);
    // Missing: packetIds[0] = packetId;
    packetStore.addPacketsToInventory(packetIds);
}
```

As a result, the `packetIds` array contains only default values (zeros), not the actual packet ID, causing `addPacketsToInventory()` to attempt to process an invalid packet ID. This will lead to a revert in `addPacketsToInventory()` when it tries to validate a zero packet ID

```
if (!_validatePacketTypeId(packetTypeId)) revert InvalidPacketType(packetTypeId);
```

This issue results in the user losing their packet NFT (as it's transferred to the contract) but not receiving anything in return, as the transaction fails during the call to `packetStore.addPacketsToInventory()`.

Proof of Concept

Please copy the following POC in `CardAllocationPoolTest.t.sol`

```
function testPOC() public {
        uint256 packetType = 1;
        uint256 packetId = 1;
        uint256[] memory cardIds = _mintCardsToAdmin(3);

        // Add bundle to pool
        vm.startPrank(admin);
        bytes32 bundleProvenance = keccak256(abi.encode(cardIds));
        pool.addCardBundlesToPacketPool(packetType, cardIds, cardIds, bundleProvenance);
        vm.stopPrank();

        // Setup packet
        vm.startPrank(admin);
        packet.registerPacketType(
            PacketStorage.PacketTypeParams({packetTypeName: "Test Pack", packetTypeMetadata:
"Test Metadata"})
        );

        packet.mintTo(
            PacketStorage.PacketMintParams({
                packetTypeId: packetType,
                packetMetadata: "Test Packet",
                packetSerialNumber: "TEST-001"
            }),user
        );

        vm.stopPrank();

        // User initiates burn which will trigger instantOpenPacket

        vm.startPrank(user);
        vm.expectRevert(RipFunStore.InvalidPacketType.selector);
        packet.initiateBurn(
            PacketStorage.PacketBurnParams({packetId: packetId, burnType:
PacketStorage.BurnType.INSTANT_OPEN_PACKET})
        );
        vm.stopPrank();

        // Verify request was made
        //assertTrue(coordinator.lastRequestId() == 0);
    }
```

Recommendations

Add the missing assignment to store the packet ID in the array before calling `addPacketsToInventory()`

```
 function finalizeOpen(uint256 packetId, uint256[] memory selectedBundle, string memory
openMetadata)
        external
        onlyRole(ALLOCATION_MANAGER_ROLE)
    {
       //...
        if (burnType == BurnType.INSTANT_OPEN_PACKET) {
            _transfer(ownerOf(packetId), address(this), packetId);
            this.approve(address(packetStore), packetId);
```

```
            uint256[] memory packetIds = new uint256[](1);
+           packetIds[0] = packetId;
            packetStore.addPacketsToInventory(packetIds);
        } else {
          //...
        }
        //...
    }
```

This will ensure that the correct packet ID is passed to the `addPacketsToInventory()` function, allowing the transaction to complete successfully.

High findings

[H-01] Missing index updates in `burnPacketFromInventory()` cause failures

Severity

Impact: Medium

Likelihood: High

Description

In `PacketStore`, the `_packetInventoryIdx` mapping tracks the position of each packet in its type's inventory array, `_packetInventory`. When burning a packet, the function moves the last packet to replace the burned one **but fails to update the `_packetInventoryIdx` mapping of packets.**

```
function addPacketsToInventory(uint256[] calldata packetIds) external
onlyRole(INVENTORY_MANAGER_ROLE) {
    uint256 length = packetIds.length;
    for (uint256 i = 0; i < length; i++) {
        --- SNIPPED ---

@>      _packetInventoryIdx[packetId] = _packetInventory[packetTypeId].length;
@>      _packetInventory[packetTypeId].push(packetId);

        emit PacketAddedToInventory(packetId, packetTypeId);
    }
}

function burnPacketFromInventory(uint256 packetId) external onlyRole(INVENTORY_MANAGER_ROLE) {
    --- SNIPPED ---

    uint256 idx = _packetInventoryIdx[packetId];

    _packetInventory[packetTypeId][idx] = _packetInventory[packetTypeId]
[_packetInventory[packetTypeId].length - 1];
    _packetInventory[packetTypeId].pop();
}
```

This creates two issues:

1. The moved packet's index mapping still points to its old position (last index).
2. The burned packet's index mapping remains.

Consequently, this leads to a corrupted state where:

- Future operations on the moved packet will access the wrong array position: 1) If no new packet is added to the inventory, it will point to an out-of-bounds index, causing subsequent burns of the moved packet to revert. 2) Otherwise, there will be 2 packets pointing to the same index.
- The burned packet's index mapping becomes a dangling pointer.

Consider this illustrated sequence:

Initial state:

```
_packetInventory[typeId] = [P_1, P_2, P_3]
_packetInventoryIdx = {
  P_1 => 0,
  P_2 => 1,
  P_3 => 2
}
```

Burn packet `P_1`:

```
function burnPacketFromInventory(uint256 packetId) external {
    uint256 idx = _packetInventoryIdx[packetId]; // idx = 0
    // Move P_3 to index 0
    _packetInventory[packetTypeId][idx] = _packetInventory[packetTypeId][2];
    _packetInventory[packetTypeId].pop();
}
```

Resulting corrupted state:

```
_packetInventory[typeId] = [P_3, P_2] // length =2
_packetInventoryIdx = {
  P_1 => 0,  // Dangling
  P_2 => 1,
  P_3 => 2   // OOB, array length is 2 and actual position is at index 0
}
```

Attempting to burn `P_3` will revert due to index out of bounds or Adding a new packet creates an index collision.

```
_packetInventory[typeId] = [P_3, P_2, P_4] // length = 2
_packetInventoryIdx = {
  P_1 => 0,
  P_2 => 1,
  P_3 => 2  // Collision
  P_4 => 2  // Collision
}
```

Recommendation

Update the index mapping for the moved packet and delete the burned packet's mapping.

```
function burnPacketFromInventory(uint256 packetId) external {
    uint256 idx = _packetInventoryIdx[packetId];

    uint256 lastPacketId = _packetInventory[packetTypeId][_packetInventory[packetTypeId].length
- 1];
    _packetInventory[packetTypeId][idx] = lastPacketId;

+   _packetInventoryIdx[lastPacketId] = idx;
+   delete _packetInventoryIdx[packetId];
    _packetInventory[packetTypeId].pop();
}
```

[H-02] Packet burn state not reset in `finalizeOpen()` causes failure

Severity

Impact: Medium

Likelihood: High

Description

In the `Packet.sol` contract, when a user initiates a burn with **INSTANT_OPEN_PACKET** type, an issue occurs in the execution flow that leads to transaction failures. The sequence occurs as follows: - User calls `initiateBurn()` with **INSTANT_OPEN_PACKET** burn type in `Packet.sol`. - This sets the packet's burn state to **INSTANT_OPEN_PACKET** and calls `instantOpenPacket()` in `CardAllocationPool.sol`. - When only one card bundle is available, `CardAllocationPool` directly transfers NFTs and calls `finalizeOpen()` (the issue still exist if the contract request randomness from Chainlink VRF). - The `finalizeOpen()` function in `Packet.sol` attempts to transfer the packet to the contract sing directly the `_transfer()` function and then call `addPacketsToInventory()` on PacketStore.sol. - The `addPacketsToInventory()` function tries to transfer the packet using `safeTransferFrom()`. - However, this transfer fails because the packet's burn type is still **INSTANT_OPEN_PACKET**. The key issue is in the `transferFrom()` function which has a check:

```
function transferFrom(address from, address to, uint256 tokenId) public virtual override {
    if (_packetBurnType[tokenId] != BurnType.NONE) revert PacketFrozen();
    super.transferFrom(from, to, tokenId);
}
```

This function reverts with `PacketFrozen()` error when the burn type is not **NONE**, but the burn type is never reset to **NONE** during the `finalizeOpen()` process for **INSTANT_OPEN_PACKET** burn type.

Recommendations

Modify the `finalizeOpen()` function in `Packet.sol` contract to reset the burn state to **NONE** before transferring the packet:

```
if (burnType == BurnType.INSTANT_OPEN_PACKET) {
        // Reset burn state to NONE before transfer
        _packetBurnType[packetId] = BurnType.NONE;

        _transfer(ownerOf(packetId), address(this), packetId);
        this.approve(address(packetStore), packetId);

        uint256[] memory packetIds = new uint256[](1);
        packetStore.addPacketsToInventory(packetIds);
    } else {
```

Medium findings

[M-01] Failure in `sendNFTs()` leaves NFTs stuck and causes asset loss

Severity

Impact: Medium

Likelihood: Medium

Description

In the `CardAllocationPool` contract, when random card NFTs are being allocated to the owner (the owner of the `packetId`), if the `sendNFTs` function fails to transfer the selected cards, **these cards will be permanently stuck in the contract**:

```
function fulfillRandomWords(uint256 requestId, uint256[] memory randomWords) internal {
       //...

       // Select random cards using the provided randomness
       uint256[] memory selectedCards = selectRandomCards(cardBundles, randomWords[0]);

       // Transfer cards to owner
       try this.sendNFTS(selectedCards, request.owner) {
           RipFunPacks(packetNFTAddress).finalizeOpen(request.packetId, selectedCards, "");
           emit PacketOpenFulfilled(requestId, request.packetId, selectedCards);
       } catch {}

       //...
   }
```

The `sendNFTS()` invocation occurs within a `try/catch` block, where the catch does not revert any state changes or implement any functionality to reallocate the failed-to-send `selectedCards`. As a result, the cards remain **permanently stuck in the contract** and can not be re-utilized in future random selections. Additionally, there is no mechanism to allow the redeem manager to recover these cards.

Since the `cardBundles` are removed from the `packetTypeToCardBundles` array, the physical items associated with these cards NFTs **become permanently unredeemable**.

A similar issue in `SpinLottery.fulfillRandomness()` function when `_distributePrize()` fails.

Recommendations

Implement a mechanism to re-add the failed-to-send cards NFTs to the `packetTypeToCardBundles` array, or introduce a mechanism to enable the redeem manager to recover these cards NFTs.

[M-02] Missing nonce in `LISTING_TYPEHASH` affects EIP-712 and signatures

Severity

Impact: Medium

Likelihood: Medium

Description

In the `Marketplace.generateListingSignatureHash()` function, when the hash is generated for a listing request, it includes the `nonce` parameter, however, this nonce is not introduced in the `LISTING_TYPEHASH`, which breaks the `EIP-712` compliance, leading to the failure of the signature validation and causing the listing process to be non-compliant, thus; blocking listings from being correctly verified and processed:

```
    /**
     * @notice The EIP-712 type hash for listing requests.
     */
    bytes32 public constant LISTING_TYPEHASH = keccak256(
        "ListingRequest(address tokenContractAddress,uint256 tokenId,uint256 price,address
acceptedCurrency,uint256 deadline,address owner,uint256 chainId)"
    );
```

```
 function generateListingSignatureHash(ListingRequest calldata listing) public view returns
(bytes32) {
        return _hashTypedDataV4(
            keccak256(
                abi.encode(
                    LISTING_TYPEHASH,
                    listing.tokenContractAddress,
                    listing.tokenId,
                    listing.price,
                    listing.nonce,// @audit : not introduced in LISTING_TYPEHASH
                    acceptedCurrency,
                    listing.deadline,
                    listing.owner,
                    listing.chainId
                )
            )
        );
    }
```

Recommendations

Update `LISTING_TYPEHASH` to include nonce:

```
-    bytes32 public constant LISTING_TYPEHASH = keccak256("ListingRequest(address
tokenContractAddress,uint256 tokenId,uint256 price,address acceptedCurrency,uint256
deadline,address owner,uint256 chainId)");
+    bytes32 public constant LISTING_TYPEHASH = keccak256("ListingRequest(address
tokenContractAddress,uint256 tokenId,uint256 price,uint256 nonce,address
acceptedCurrency,uint256 deadline,address owner,uint256 chainId)");
```

Low findings

[L-01] Cleanup missing for `packetToBundle[packetId]` in `burnFromInventory()`

The `Packet.burnFromInventory()` function is called by the allocation manager to remove a packet from the packet store. However, it doesn't delete the associated `packetToBundle[packetId]` when the packet is burned.

```
function burnFromInventory(uint256 packetId) external onlyRole(ALLOCATION_MANAGER_ROLE) {
    if (_ownerOf(packetId) == address(packetStore)) {
        packetStore.burnPacketFromInventory(packetId);
        emit PacketBurnedFromInventory(packetId);
        _burn(packetId);
    }
}
```

Recommendation: Ensure that `packetToBundle[packetId]` is deleted when the `burnFromInventory()` function is called:

```
function burnFromInventory(uint256 packetId) external onlyRole(ALLOCATION_MANAGER_ROLE) {
    if (_ownerOf(packetId) == address(packetStore)) {
        packetStore.burnPacketFromInventory(packetId);
        emit PacketBurnedFromInventory(packetId);
        _burn(packetId);
+       delete packetToBundle[packetId];
    }
}
```

[L-02] Stuck fees in `PacketStore` upon `paymentToken` change without withdrawal

In the `PacketStore.setPaymentToken()` function, the accepted payment token is changed without first withdrawing any accumulated fees that have yet to be transferred, which would result in these fees of the old payment token being stuck in the contract:

```
  function setPaymentToken(
        address newTokenAddress
    ) external onlyRole(SUPER_ADMIN_ROLE) {
        if (newTokenAddress == address(0)) revert InvalidAddress();
        paymentToken = IERC20(newTokenAddress);
        emit PaymentTokenUpdated(newTokenAddress);
    }
```

Recommendation: Before changing the `paymentToken`, ensure that accumulated fees are collected via the `withdrawFunds()` function.

[L-03] Missing validation for burned `packetIds` in `addCardBundlesToPacketPool()`

In the `CardAllocationPool.addCardBundlesToPacketPool()` function, intended to be called by the redeemer manager to add cards to a specific packet type's pool as a bundle, there is no validation to ensure that the `packetIds` provided belong to the same `packetTypeId` being added.

This lack of validation can result in inconsistent or incorrect packet assignments. Additionally, it creates a potential security risk, as a malicious redeem manager could exploit this gap to burn any packet listed in the `PacketStore`, leading to the removal of unintended packets and causing possible disruptions in the system.

```
  function addCardBundlesToPacketPool(
        uint256 packetType,
        uint256[] memory cardBundles,
        uint256[] memory packetIds,
        bytes32 bundleProvenance
    ) external noPendingRandomness onlyRole(redeemManagerRoleId) {
        //...

        packetTypeToCardBundles[packetType].push(CardBundle({cardIds: cardBundles,
bundleProvenance: bundleProvenance}));
        for (uint256 i = 0; i < cardBundles.length; i++) {
            RipFunPacks(packetNFTAddress).burnFromInventory(packetIds[i]);
            IERC721(cardNFTAddress).safeTransferFrom(msg.sender, address(this), cardBundles[i]);
        }

        //...
    }
```

Recommendation: Implement a validation step to ensure that all `packetIds` provided are associated with the provided `packetTypeId`.

[L-04] Missing packet type validation in `addCardBundlesToPacketPool()`

In the `CardAllocationPool.addCardBundlesToPacketPool()` function, the function is designed to be called by the redeem manager to add cards to a specific packet type's pool as a bundle. However, it does not validate if the `packetType` is a registered type before assigning `cardBundles` to it. This could lead to incorrect data being added to the pool for unregistered or invalid packet types, resulting in inconsistent behavior or potential vulnerabilities.

```
function addCardBundlesToPacketPool(
        uint256 packetType,
        uint256[] memory cardBundles,
        uint256[] memory packetIds,
        bytes32 bundleProvenance
    ) external noPendingRandomness onlyRole(redeemManagerRoleId) {
        if (cardBundles.length > maxBundleSize) revert CannotExceedMaxBundleSize();
```

```
        if (cardBundles.length != packetIds.length) revert IncorrectLength();

        packetTypeToCardBundles[packetType].push(CardBundle({cardIds: cardBundles,
bundleProvenance: bundleProvenance}));
        //....
    }
```

Recommendation:
Implement validation to ensure that the `packetType` is registered before adding the `cardBundles` to the pool:

```
function addCardBundlesToPacketPool(
        uint256 packetType,
        uint256[] memory cardBundles,
        uint256[] memory packetIds,
        bytes32 bundleProvenance
    ) external noPendingRandomness onlyRole(redeemManagerRoleId) {
        if (cardBundles.length > maxBundleSize) revert CannotExceedMaxBundleSize();
        if (cardBundles.length != packetIds.length) revert IncorrectLength();
+       packetNFTAddress.getPacketTypeInfo(packetType); // @note: it will revert if the
packetType is not registerd
        packetTypeToCardBundles[packetType].push(CardBundle({cardIds: cardBundles,
bundleProvenance: bundleProvenance}));
        //....
    }
```

[L-05] Missing state cleanup for burned packets

In the `Packet` contract, when an NFT (Packet) is burned, the state variables `_packetUris[packetId]` and `_packetTypeIds[packetId]` are not deleted. While the `tokenURI()` getter function for the `_packetUris[packetId]` parameter will revert for a burned or non-existent NFT, **these states are public** and can still be accessed directly, which leads to invalid data being returned for burned or non-existent NFTs.

This issue is present in the `finalizeOpen()`, `finalizeRedeem()`, and `burnFromInventory()` functions, where the states related to the burned NFT are not properly cleaned up.

```
/// @notice Maps token IDs to their packet type IDs
/// @dev Used to determine which type a packet belongs to
mapping(uint256 => uint256) public _packetTypeIds;

/// @notice Maps token IDs to their packet metadata
/// @dev Used to store packet metadata for each token
mapping(uint256 => string) public _packetUris;
```

Recommendation:
Whenever an NFT (Packet) is burned, delete the `_packetUris[packetId]` and `_packetTypeIds[packetId]` states:

```
_burn(packetId);
+ delete _packetUris[packetId];
+ delete _packetTypeIds[packetId];
```

[L-06] Role ID collision between `PacketStore` and `WhitelistRegistry` contracts

Both the `PacketStore` and `WhitelistRegistry` contracts define role ID `6` for unrelated purposes:

```
//File: src/PacketStore.sol
uint256 private constant PRICE_MANAGER_ROLE = 6;

//File: src/WhitelistRegistry.sol
uint256 public constant WHITELIST_MANAGER_ROLE = 6;
```

Both contracts use `RoleBasedAccessControl`, which uses ERC1155 tokens to manage roles. Since they share the same role ID, any address granted one role effectively has both roles, as the access control system cannot distinguish between them.

It does violate the principle of least privilege and could lead to confusion in role management.

Consider assigning unique role IDs for each distinct role in the system.

[L-07] `onERC721Received` missing blocks safe NFT transfers and mints

The `SpinLottery` contract requires NFTs to be transferred or minted to it to be used as prizes, as verified by the ownership check.

```
function addPrize(address _nftAddress, uint256 _tokenId, uint8 _rarity) external
onlyRole(lotteryManagerRoleId) {
    IERC721 nft = IERC721(_nftAddress);
@>  if (nft.ownerOf(_tokenId) != address(this)) revert NFTNotOwnedByContract();

    uint256 packed = packPrize(_nftAddress, uint88(_tokenId), _rarity);
    _addPrizeToPool(_rarity, packed);

    emit PrizeAdded(_nftAddress, _tokenId, _rarity);
}
```

However, the contract lacks the `onERC721Received` implementation required by safe functions of ERC721. This prevents any external integrations that use safe methods from successfully sending NFTs to the contract.

Consider implementing the `onERC721Received` interface in the `SpinLottery` contract.

[L-08] Missing offer revocation mechanism in Marketplace contract

The `Marketplace.sol` lacks a mechanism for offer makers to revoke their offers before the deadline expires. While the contract implements `revokedListings` for seller listings through the `revokeListing()` function, there is no equivalent functionality for offers. This creates a security concern as requesters cannot cancel their offers if market conditions change rapidly or if they made a mistake, forcing them to rely solely on short deadlines, which is insufficient in fast-moving crypto markets.

[L-09] Missing `chargeDelegate` parameter in signed listing data

In the `Marketplace.sol` contract, the `buy()` function accepts a `chargeDelegate` parameter that determines who pays for the purchase (caller or receiver), but this parameter is not included in the signed `ListingRequest` data structure.

```
function buy(ListingParams[] calldata listings, bool chargeDelegate) external nonReentrant {
    // ...
    address payee = chargeDelegate ? address(msg.sender) : listings[i].receiver;
    // ...
    IERC20(acceptedCurrency).safeTransferFrom(payee, listings[i].request.owner, sellerAmount);
}
```

This creates a security risk because receivers who have previously approved the marketplace to spend their USDC could unexpectedly pay for purchases when returning to the marketplace if a malicious caller sets `chargeDelegate` to `false`. Since this parameter isn't part of the signed data, the payment arrangement can be manipulated without invalidating the signature.

To resolve this, Include the `chargeDelegate` parameter in the `ListingRequest` struct.

[L-10] Silent error swallowing due to early event

In `MarketplaceAdmin.sol`, multiple administrative functions use a try/catch pattern when calling functions on the Marketplace contract, but then silently ignore any errors that occur. More concerning, the contract emits success events before entering the try/catch block, meaning events are emitted even if the underlying operation fails. This occurs in multiple functions, including: - updateAcceptedCurrency(). - updateFeeReceiver(). - updateFees(). - updateTradeFee(). - collectFees(). - updateWhitelistRegistryAddress(). - emergencyShutdown().

This pattern create issue, because events are emitted indicating success even if the operation fails, so Incorrect event logs may mislead users, frontend applications, and monitoring systems. Admin functions appear to succeed when they actually fail.

To resolve this, move event emissions inside the try block or after successful execution to ensure events are only emitted upon success.

[L-11] `revertBurnRequest()` mismanages `INSTANT_OPEN_PACKET` burn type

The `revertBurnRequest()` function in `Packet.sol` allows reverting any burn type, including **INSTANT_OPEN_PACKET**. However, when a packet has burn type **INSTANT_OPEN_PACKET**, the packet opening process has already been initiated through `randomAllocationPool.instantOpenPacket()`, which starts an asynchronous VRF process that can't be simply canceled by changing the burn type back to NONE. This creates an inconsistent state where the burn type is reset to `NONE` in the Packet contract. And the VRF process continues in the `CardAllocationPool` contract.

To resolve this, set `fulfilled` to true.

[L-12] Risk of packet loss in `authorizedOpenPacket()` with empty cards

There is a critical vulnerability in the interaction between `CardAllocationPool.sol` and `Packet.sol` when handling **OPEN_PACKET** burn type. When a user initiates a burn with **OPEN_PACKET** type, a manager can call `authorizedOpenPacket()` in `CardAllocationPool.sol` to open packets with specific cards. The issue is that the `uthorizedOpenPacket()` function does not verify that the cards array contains at least one card. If a manager calls this function with an empty cards array, either maliciously or by mistake, it will still proceed to call `finalizeOpen()` in `Packet.sol`:

```
if (burnType == BurnType.INSTANT_OPEN_PACKET) {
    // ... INSTANT_OPEN_PACKET handling ...
} else {
    _burn(packetId);
}
```

This results in the packet being permanently burned without the user receiving any card in return, as the cards array was empty. This is a significant loss for the user who initiated the burn process.

To resolve this, modify the `authorizedOpenPacket()` function in `CardAllocationPool.sol` to require a non-empty cards array.

[L-13] `requestConfirmations` below Chainlink minimum allowed in `UpdateVRFConfig()`

In `CardAllocationPool.sol`, the `updateVRFConfig()` function allows setting the `requestConfirmations` parameter without enforcing Chainlink's minimum requirement. The function only checks that `_requestConfirmations` is not zero, but it doesn't enforce network-specific minimum values. For Polygon Mainnet, Chainlink requires a minimum of 3 confirmations as documented in their official documentation.

[L-14] Prizes need upper bound to prevent `configureRarity` going OOG

Prizes for a rarity pool can be added using `addPrize()` and currently there is no upper limit to the prizes that can be added for a rarity pool , due to this the function `configureRarity()` might go out of gas when it tries to transfer the prize NFTs in the rarity pool to the msg.sender , meaning rarity pools configuration would fail when needed , this functionality is crucial to the system to de-activate/activate rarities and to transfer out NFTs stuck in the contract due to failed transfers.

Recommendation: Have a safe upper limit on how many NFT prizes max can be added to a rarity pool.

[L-15] Sellers face fee volatility without safeguard in `MarketPlace.buy()`

In the `MarketPlace.buy()` function, a percentage fee is deducted from the request price for each purchase. However, the `fees` percentage can change between when the listing is added and when it is fulfilled via the `buy()` function. This creates a risk that the seller may receive a price that deviates significantly from the originally accepted price due to changing `fees`, resulting in financial loss for the seller.

```
 function buy(ListingParams[] calldata listings, bool chargeDelegate) external nonReentrant {
        for (uint256 i = 0; i < listings.length; i++) {
            //...
            uint256 feeAmount = Math.mulDiv(listings[i].request.price, fees, 10000,
Math.Rounding.Ceil);
            uint256 sellerAmount = listings[i].request.price - feeAmount;
            //...
            if (feeAmount > 0) IERC20(acceptedCurrency).safeTransferFrom(payee, address(this),
feeAmount);

            //...
        }
    }
```

A similar issue in `acceptOffer()` function.

Recommendations: Implement a mechanism that allows the seller to define an acceptable price deviation (`minPrice`):

```
 function buy(ListingParams[] calldata listings, bool chargeDelegate) external nonReentrant {
        for (uint256 i = 0; i < listings.length; i++) {
            //...
            uint256 feeAmount = Math.mulDiv(listings[i].request.price, fees, 10000,
Math.Rounding.Ceil);
            uint256 sellerAmount = listings[i].request.price - feeAmount;
+           require(sellerAmount > listings[i].request.minPrice)
            // External calls after state changes
            if (sellerAmount > 0) {
```

```
                IERC20(acceptedCurrency).safeTransferFrom(payee, listings[i].request.owner,
sellerAmount);
            }

            if (feeAmount > 0) IERC20(acceptedCurrency).safeTransferFrom(payee, address(this),
feeAmount);

            //...
        }
    }
```

This change requires updating the `generateListingSignatureHash()` function and `LISTING_TYPEHASH` to account for this `request.minPrice`.

[L-16] Incorrect packet burning mechanism in bundle allocation

The `CardAllocationPool.addCardBundlesToPacketPool()` function **burns 1 packet for each card in a bundle**.

This creates a mismatch between packets and card bundles, as the protocol's design shows that 1 packet should correspond to 1 card bundle (which may contain multiple cards).

Burning 1 packet for each card in a bundle can lead to incorrectly burning multiple packets or burning packets that are not of the same `packetType` as the added card bundle.

```
function addCardBundlesToPacketPool(
    ...
) external noPendingRandomness onlyRole(redeemManagerRoleId) {
    --- SNIPPED ---
    for (uint256 i = 0; i < cardBundles.length; i++) {
@>      RipFunPacks(packetNFTAddress).burnFromInventory(packetIds[i]);
        IERC721(cardNFTAddress).safeTransferFrom(msg.sender, address(this), cardBundles[i]);
    }

    emit CardsAddedToPool(packetType, cardBundles, bundleProvenance);
}
```

Recommendation: Modify the function to burn one packet per bundle rather than per card.

If the intention is to burn 1 packet per card (assuming each packet corresponds to 1 card), then the `packetType` of each packet should be validated to ensure it matches the bundle being added.

[L-17] Missing refund in `SpinLottery.sol` when prize distribution fails

In the `SpinLottery.sol` contract, the `fulfillRandomness()` function contains an issue related to prize distribution failures. When a user pays for a spin, their payment is processed immediately, but the actual prize is distributed asynchronously after Chainlink VRF provides randomness. The relevant code in `fulfillRandomness()` is:

```
// Step 3: Prize Selection
try this._distributePrize(req.player, requestId, rarity, prizeRandom) {
    // Prize distributed successfully
} catch {
    // If prize distribution fails, emit a no-win event
    emit NoWin(req.player, requestId);
}
```

When the prize distribution fails. The contract simply emits a NoWin event without refunding the user's payment. This scenario is particularly problematic in cases such as: When a lottery manager calls `configureRarity()` to add a new rarity but hasn't yet added any NFT prizes using `addPrize()`.

Recommendations: When prize distribution fails in the catch block, the contract should refund the user's payment, so you need to keep track of the exact amount paid by each user for each spin to ensure accurate refunds.

[L-18] Rarity configuration in SpinLottery leads to `pendingCount` errors

In the `SpinLottery.sol` contract, The issue occurs in the `fulfillRandomness()` function where the function relies on the current state of `rarityConfigs[i].active` to determine which rarities should have their pending counts decreased.

However, there's a critical flaw: the `rarityConfigs[i].active` state during randomness fulfillment may not match the state when the `spin()` function was originally called.

Here's the problematic scenario: - User calls spin(), increasing `prizePools[rarity1].pendingCount` for an active rarity (rarity1). - Before Chainlink VRF responds with randomness, the lottery manager calls `configureRarity()` and adds a new rarity (rarity2). - Another user calls `spin()`, which increases the pendingCount for the newly added rarity2. - When `fulfillRandomness()` is called for the first user's request, it checks `rarityConfigs[i].active` for all rarities. - It finds rarity2 is now active, so it decrease `pendingCount` for rarity2.

The function will incorrectly decrease `pendingCount` for rarity2, even though this pending count corresponds to the second user's spin request, not the first user's request that's currently being processed. This creates a mismatch between actual pending requests and the tracked `pendingCount`, which could lead to prizes being incorrectly distributed or requests being incorrectly accounted for.

Recommendations: Store the active rarities at the time of the spin request alongside the `SpinRequest` struct:

```
struct SpinRequest {
    address player;
    uint256 totalSlots;
```

```
    uint256 prizeCount;
    uint256[] activeRarities; // Store array of active rarity IDs when spin was initiated
}
```

[L-19] Revocation may fail after currency update allowing exploits

The `revokeListing()` function in `Marketplace.sol` has a critical issue that prevents users from revoking their listings if the `acceptedCurrency` has been updated since the listing was created.

This occurs because the function uses the current `acceptedCurrency` value when generating the signature hash to validate the revocation request:

```
bytes32 listingHash = generateListingSignatureHash(listing.request);
```

Looking at `generateListingSignatureHash()`:

```
function generateListingSignatureHash(ListingRequest calldata listing) public view returns
(bytes32) {
    return _hashTypedDataV4(
        keccak256(
            abi.encode(
                //...
                acceptedCurrency,  // Current value used here
                //...
```

When a user creates a listing, the signature is generated with the then-current `acceptedCurrency` address. If an admin later updates the `acceptedCurrency` through `setAcceptedCurrency()`, any attempt to revoke a listing will fail because the calculated `listingHash` will be different. The impact is: * Users cannot revoke outdated listings that might have long deadlines (e.g., 1 month). * If the `acceptedCurrency` is ever switched back to the original token address, outdated listings suddenly become valid again. * Attackers could wait for such a situation and then immediately execute the outdated listings at potentially disadvantageous prices for the seller.

When the marketplace admin reverts to a previously used token, there's a race condition where users must quickly revoke their old listings, but many will likely fail to do so in time, creating a window of opportunity for exploitation.

Recommendations: This can be resolved by, Track historical currencies, Maintain a mapping of historical currency addresses and allow users to specify which currency was used when revoking.

[L-20] Token collection failure during update leads to mixed accounting

In the `MarketplaceAdmin.sol` contract, the `updateAcceptedCurrency()` function attempts to collect pending fees before updating the payment token.

The issue occurs when the fee collection silently fails (e.g., if the `feeReceiver` is blacklisted or the token contract is paused), but the currency update succeeds. In this case:

- The `totalPendingFees` in `Marketplace.sol` is not reset to zero.
- New fees collected in the new token will be added to this same counter.
- This results in mixing accounting for two different tokens in a single variable.

This becomes particularly problematic when the two tokens have different decimals (e.g., USDC with 6 decimals and WETH with 18 decimals), the admin doesn't notice the issue early, and users continue paying fees in the new token. When `collectFees()` is called again, it will attempt to transfer the combined amount in only the new token, eventually leading to incorrect fee withdrawals or failed transactions.

Recommendations: Modify the `updateAcceptedCurrency()` function to use a more robust pattern for fee collection.

[L-21] DoS vulnerability in spin due to pending prize

Consider a scenario with three rarity pools:

Rarity 1: 1 prize available. Rarity 2: 10 prizes available. Rarity 3: 10 prizes available.

User1 calls `spin()` For rarity 1, pendingCount becomes 1 Any subsequent user calling spin() will trigger:

```
// For rarity 1:
// available=1, pending=1, pendingCounts[1]=1
// available - pending = 0, which is < pendingCounts[1]
if (available - pending < pendingCounts[i]) {
    revert InsufficientPrizes();
}
```

The function reverts, blocking access until the VRF callback completes. An attacker can leverage this by calling spin N number of times where N is the number of prizes of the rarityId with the least number of prizes.

Though this is done this way to prevent scenarios where prize was of the same rarity for two spins and if one spin consumed the final prize nft then the next spin's prize wouldnt have a prize nft to be transferred , there should be safeguards to ensure there are enough prizes so that such griefing can not be done.

Recommendations: A possible solution can be if a rarity prize pool is consumed then increment the pending count of the next available rarity.