Pashov Audit Group

Rip It Security Review

Conducted by:

0x37

Arnie

Said

jesjupyter

km

0xTheBlackPanther

dobrevaleri

April 25th 2025 - May 2nd 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 **0x37**, **Arnie**, **Said**, **jesjupyter**, **km**, **0xTheBlackPanther**, **dobrevaleri** engaged to review **Rip It**. A total of **54** issues were uncovered.

Protocol Summary

Project Name	Rip It
Protocol Type	Game
Timeline	April 25th 2025 - May 2nd 2025

Review commit hash:

- a8f2ee2544ef7ae42397f8d75915416bfeb7cb81 (ripdotfun/rip-it-core)

Fixes review commit hash:

- 2c22ac3f742b3f3a27161d39e762fa4449c4c1cd (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
High	7
Medium	14
Low	33
Total findings	**54**

Summary of findings

ID	Title	Severity	Status
[H-01]	Funds in subscription may be drained	High	**Resolved**
[H-02]	Insufficient restrictions in `instantOpenPacket()` risk DOS	High	**Resolved**
[H-03]	Prize locking mechanism inconsistency with weight proportions	High	**Resolved**
[H-04]	DoS risk from pending prize locking mechanism	High	**Resolved**
[H-05]	Currency validation missing in listing and offer requests	High	**Resolved**
[H-06]	Incorrect `pendingCounts` calculation	High	**Resolved**
[H-07]	`SpinLottery` will break if the first `prizeId` is selected	High	**Resolved**
[M-01]	Missing sanity check in `addCardBundlesToPacketPool`	Medium	**Resolved**
[M-02]	Lack of nonce mechanism in signature hash generation	Medium	**Resolved**
[M-03]	Inconsistent trade fees from `tradeFee` and `acceptedCurrency` mods	Medium	**Resolved**
[M-04]	EIP-712 compliance Issue in `Marketplace`	Medium	**Resolved**
[M-05]	Potential out-of-bounds in `removeCardBundlesFromPacketPool`	Medium	**Resolved**

ID	Title	Severity	Status
[M-06]	Improper randomness requests in `CardAllocationPool` and `SpinLottery`	Medium	**Resolved**
[M-07]	Non-compliance with EIP-721 in `tokenURI()` function	Medium	**Resolved**
[M-08]	Inconsistent logic in `initiateBurn()` and `initiateBurnBatch()` functions	Medium	**Resolved**
[M-09]	Role management vulnerability in `assignRole()` function	Medium	**Resolved**
[M-10]	Spin cost fees are stuck in the SpinLottery	Medium	**Resolved**
[M-11]	Incorrect initial `tradeFee` in `marketPrice`	Medium	**Resolved**
[M-12]	Inconsistent `pendingCounts` in `spin()` and `fulfillRandomness()` when 0	Medium	**Resolved**
[M-13]	Buy request executed by providing `request.owner`	Medium	**Resolved**
[M-14]	Upgrade permissions inappropriate for `ROLE_MANAGER_ROLE`	Medium	**Resolved**
[L-01]	No NFT retrieval mechanism in `SpinLottery` for inactive rarities	Low	**Resolved**
[L-02]	Zero prize locking for single prizes due to implicit condition	Low	**Resolved**
[L-03]	Precision loss from `Divide-Before-Multiply` in price calculation	Low	Acknowledged
[L-04]	`getAvailablePrizes` may return incorrect result	Low	**Resolved**
[L-05]	No duplicate prize check in prize pool addition	Low	Acknowledged
[L-06]	Risks in modifying critical parameters in marketplace	Low	Acknowledged
[L-07]	DoS risk with zero-amount transfers	Low	**Resolved**
[L-08]	Compatibility issues with `transferFrom()` in ERC20	Low	**Resolved**
[L-09]	Inconsistent validation in `isValidListing()` and `isValidTrade()`	Low	Acknowledged

ID	Title	Severity	Status
[L-10]	Data inconsistency risk from modifying critical addresses in `cardAllocationPool`	Low	Acknowledged
[L-11]	Deterministic selection with one available card	Low	**Resolved**
[L-12]	Validation issues in `authorizedOpenPacket()`	Low	Acknowledged
[L-13]	Inconsistent role ID in `cardAllocationPool` initialization	Low	Acknowledged
[L-14]	Risks of modifying `randomAllocationPool` during a transaction	Low	**Resolved**
[L-15]	Shared privilege concerns in role-based access control cards	Low	Acknowledged
[L-16]	Data integrity issue with `revertBurnRequest` in `RipFunPacks`	Low	Acknowledged
[L-17]	Unconventional role transfer timing in `RoleBasedAccessControl`	Low	Acknowledged
[L-18]	No storage gap in `RoleBasedAccessControl` contract	Low	**Resolved**
[L-19]	Unnecessary modifier for `getInventoryPackets`	Low	**Resolved**
[L-20]	Low `requestConfirmations` number prone to frequent chain reorgs	Low	**Resolved**
[L-21]	`CardAllocationPool` should use `ERC1155HolderUpgradeable`	Low	**Resolved**
[L-22]	Missing cancel burn functionality in `Card` and `Packet`	Low	**Resolved**
[L-23]	There is no slippage control in `spin`	Low	**Resolved**
[L-24]	There is no slippage protection when purchasing packets	Low	**Resolved**
[L-25]	Insufficient error handling	Low	Acknowledged
[L-26]	Inconsistent prize locking/unlocking from dynamic weight changes	Low	Acknowledged
[L-27]	Redundant state management and inconsistency risk in `MarketplaceAdmin`	Low	**Resolved**

ID	Title	Severity	Status
[L-28]	Potential revenue loss due to fee calculation method	Low	Resolved
[L-29]	Indexing issues in `removeCardBundlesFromPacketPool()`	Low	Resolved
[L-30]	Use of `transferFrom` over `safeTransferFrom()` in NFT transfers	Low	Resolved
[L-31]	Use of `_mint` instead of `_safeMint()` in minting	Low	Resolved
[L-32]	Potential incomplete role revocation in `revokeRole()` function	Low	Resolved
[L-33]	Trade fee calculation mismatch in documentation and implementation	Low	Resolved

High findings

[H-01] Funds in subscription may be drained

Severity

Impact: High

Likelihood: Medium

Description

In SpinLottery, users can participate in the spin activity to win one prize card. Users can choose their win possibility. If users choose one higher win possibility, users have to pay one higher payment for this spin.

When users trigger one spin, we will request one randomness from Chainlink VRF. In RIP, we choose the Subscription mode. Chainlink VRF will charge the related fees from our subscription balance. Each time, we trigger one spin, Chainlink VRF will charge the related fees from our subscription balance.

According to Chainlink VRF billing formula, the fee for one random is `gasPrice * (callback gas used + Verification gas used) * ((100 + premium percentage)/100)`.

The problem here is that some malicious users can start one grief attack. The hacker can repeat triggering `spin` to consume our subscription balance. In a normal scenario, one spin's cost should be `gas used + spin cost`. However, the hacker can increase the `_totalSlots` to one very huge value to let the spit cost round down to 0. The only cost for this attack vector is the `gas used`.

This will cause Chainlink VRF will consume all balance in our subscription account.

```
    function spin(uint256 _totalSlots, uint256 _prizeCount) external whenNotPaused returns
(uint256) {
        uint256 spinCost = calculateSpinCost(_totalSlots, _prizeCount);

        // Process payment
        if (!usdcToken.transferFrom(msg.sender, address(this), spinCost)) {
            revert USDCTransferFailed();
        }
        uint256 requestId = requestRandomness();
        spinRequests[requestId] = SpinRequest({
            player: msg.sender,
            totalSlots: _totalSlots,
            prizeCount: _prizeCount
        });
    }
    function calculateSpinCost(uint256 _totalSlots, uint256 _prizeCount) public view returns
```

```
(uint256) {
        return (avgBasePrice * _prizeCount * 1000) / (_totalSlots * 1000);
    }
```

Recommendations

Add one minimum spin cost.

[H-02] Insufficient restrictions in `instantOpenPacket()` risk DOS

Severity

Impact: Medium

Likelihood: High

Description

In `Packet.sol` a user can call `initiateBurn` with `params.BurnType` set to `BurnType.INSTANT_OPEN_PACKET`.

```
        if (params.burnType == BurnType.INSTANT_OPEN_PACKET) {
            randomAllocationPool.instantOpenPacket(params.packetId,
_packetTypeIds[params.packetId], msg.sender);
        }
```

This will result in the function calling `instantOpenPacket` on the `CardAllocationPool` contract.

```
    function instantOpenPacket(uint256 packetId, uint256 packetType, address owner) external {
        if (msg.sender != packetNFTAddress) revert UnauthorizedCaller();

        if (packetTypeToCardBundles[packetType].length == 0) revert InsufficientCardBundles();

        // Request randomness from Chainlink VRF
        uint256 requestId = requestRandomWords(packetType);

        requestIdToPacketOpen[requestId] =
            PacketOpenRequest({packetId: packetId, packetType: packetType, owner: owner,
fulfilled: false});
```

The function will first check that the msg.sender is in fact the `packetNFTAdress` next the logic ensures there is enough cardBundles of the specific type to fulfill the request otherwise the function will revert with `insufficientCardBundles`. Finally a call to `requestRandomWords` is made and the `requestId` assigned.

When the chainlink VRF finally delivers the random values, it will call `selectRandomCards` using the random values.

```
function fulfillRandomWords(uint256 requestId, uint256[] memory randomWords) internal {
    PacketOpenRequest storage request = requestIdToPacketOpen[requestId];
    if (request.fulfilled) revert("Already fulfilled");

    // fetch the available card bundles
    CardBundle[] storage cardBundles = packetTypeToCardBundles[request.packetType];

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

This is where the issue begins...

```
function selectRandomCards(CardBundle[] storage availableCardBundles, uint256 randomSeed)
    private
    returns (uint256[] memory selectedCardBundle)
{
    if (availableCardBundles.length == 0) {
        revert NoAvailableCardBundles();
    }
```

If `availableCardBundles.length` == 0 we will revert, this a problem, let me explain below.

1. 2 users call `initiateBurn` on `Packet.sol` with `BurnType` == `INSTANT_OPEN_PACKET` in the same block, both requests are of the same `packetType`.
2. Assume there is only 1 bundle left of the specific `packetType`, the checks in `instantOpenPacket` will pass for both users even if there is only 1 bundle left for the specific `packetType`.
3. Next both users will be assigned a `requestId` and will be waiting for chainlink to `fulfillRandomWords`
4. When `fulfillRandomWords` is called there is an internal call to `selectRandomCards` this function will pop the only value from `availableCardBundles`, user 1 will receive his cards as intended.
5. when `fulfillRandomWords` is called for user 2, there will be a revert in `fulfillRandomWords` subcall to `selectRandomCards` because `availableCardsBundles.length` will == 0 and thus revert the entire tx.
6. Since `fulfillRandomWords` reverts, this will cause the request to be dosed according to chainlink docs

> If your fulfillRandomWords() implementation reverts, the VRF service will not attempt to call it a second time. Make sure your contract logic does not revert. Consider simply storing the randomness and taking more complex follow-on actions in separate contract calls made by you, your users, or an Automation Node.

Recommendations

Consider adding checks in `instantOpenPacket` to ensure more requests cannot be made than the amount of `cardBundles` available for a specific `packetType`.

[H-03] Prize locking mechanism inconsistency with weight proportions

Severity

Impact: High

Likelihood: Medium

Description

In the SpinLottery contract, users pay for `spin`s based on the weight distribution of the rarities, which determines the cost of participation.

```
    function calculateSpinCost(uint256 _totalSlots, uint256 _prizeCount) public view returns
(uint256) {
        if (_totalSlots == 0 || _prizeCount == 0 || _prizeCount > _totalSlots) {
            revert InvalidSlotConfiguration();
        }

        uint256 totalWeightedPrice = 0;

        // Calculate weighted average price across all active rarities
        for (uint8 i = 1; i <= maxRarityId; i++) {
            RarityConfig memory config = rarityConfigs[i];
            if (config.active) {
                totalWeightedPrice += uint256(config.basePrice) * config.weight;
            }
        }

        if (totalRarityWeight == 0) revert InvalidWeightConfiguration();
        uint256 avgBasePrice = totalWeightedPrice / totalRarityWeight;

        return (avgBasePrice * _prizeCount * 1000) / (_totalSlots * 1000);
    }
```

However, the actual locking of prizes does not adhere to this weight distribution. **For example, in a scenario where the weights are distributed as `80%`, `10%`, and `10%`, if `prizeCount` is set to 2, the prizes may be locked in a `1:1:1` ratio instead of reflecting the intended weight distribution. Conversely, if `prizeCount` is set to `1`, it may result in a locking of `1:0:0`, disregarding the weights entirely.**

Especially, when `1:0:0` is locked, the prize could still fall into the `higher rarity`: without protection of `lock mechanism`, the `fulfillRandomness` could fail unexpectedly.

```
    function determineRarity(uint256 randomValue) public view returns (uint8) {
        uint256 value = randomValue % totalRarityWeight;
        uint256 cumulativeWeight = 0;

        // Iterate through rarities to find where the random value lands
        for (uint8 i = 1; i <= maxRarityId; i++) {
            RarityConfig memory config = rarityConfigs[i];
            if (config.active) {
                cumulativeWeight += config.weight; // <= could still fall into higher rarity
                if (value < cumulativeWeight) {
                    return i;
                }
            }
        }

        // Fallback to highest rarity (should not happen unless weights are misconfigured)
        return maxRarityId;
    }
```

This inconsistency can lead to unfair outcomes, as the allocation of prizes does not align with their contributions.

Recommendations

Revise the logic of lock mechanism.

[H-04] DoS risk from pending prize locking mechanism

Severity

Impact: High

Likelihood: Medium

Description

The `calculatePendingPrizesForRarity` function in the `SpinLottery` contract implements a minimum guarantee mechanism where, if `_prizeCount` is greater than `1` and the `weight` is greater than `0`, the `pendingCount` is conservatively set to 1.

```
        // Ensure at least some prizes are allocated if weights allow it
        if (_prizeCount > 1 && pendingCount == 0 && weight > 0) {
            pendingCount = 1;
        }
```

In the `spin` function, the `pendingCounts` are locked, and if the total available prizes are insufficient, it can lead to an `InsufficientPrizes` error, resulting in a Denial of Service (DoS).

```
        for (uint8 i = 1; i <= maxRarityId; i++) {
            if (rarityConfigs[i].active) {
                pendingCounts[i] = calculatePendingPrizesForRarity(_prizeCount, i);
                totalPending += pendingCounts[i];
            }
        }
        ...
        // Check if enough prizes are available considering pending ones
        for (uint8 i = 1; i <= maxRarityId; i++) {
```

```
        if (pendingCounts[i] > 0) {
            uint256 available = getAvailablePrizes(i);
            uint256 pending = prizePools[i].pendingCount;

            if (available - pending < pendingCounts[i]) {
                revert InsufficientPrizes();
            }
        }
    }
```

This situation is exacerbated when **a high-value prize (`rarity`) has a low number of `available` prizes.**

An attacker or normal user could initiate multiple `spin` transactions with `_prizeCount = 2`, effectively locking the limited prizes and temporarily preventing other users from participating in the lottery, leading to frequent DoS scenarios for a certain period of time, which leads to bad user experience.

Recommendations

The relevant locking logic should be refined to fix this issue.

[H-05] Currency validation missing in listing and offer requests

Severity

Impact: High

Likelihood: Medium

Description

The `Marketplace` contract defines the currency in both `ListingRequest` and `OfferRequest` :

```
struct ListingRequest {
    address tokenContractAddress;
    uint256 tokenId;
    uint256 price;
    **address acceptedCurrency;**
    uint256 deadline;
    address owner;
    uint256 chainId;
}

struct OfferRequest {
    address tokenContractAddress;
    uint256 tokenId;
    uint256 price;
    **address offerCurrency;**
    uint256 deadline;
```

```
        address requester;
        uint256 chainId;
    }
```

However, in the relevant check `isValidListing` and `isValidBid`, there is no validation to ensure that the currency specified in these requests matches the `acceptedCurrency` stored in the contract.

```
    function isValidListing(ListingParams calldata listing) public view returns (bool) {
        if (listing.request.price == 0) revert InvalidPrice();
        if (block.timestamp > listing.request.deadline) revert ListingExpired();
        if (listing.request.chainId != block.chainid) revert InvalidChainId();
        bytes32 listingHash = generateListingSignatureHash(listing.request);
        if (usedListings[listingHash]) revert ListingAlreadyUsed();
        if (!
IWhitelistRegistry(whitelistRegistry).isWhitelisted(listing.request.tokenContractAddress)) {
            revert NFTNotWhitelisted();
        }
        if (IERC721(listing.request.tokenContractAddress).ownerOf(listing.request.tokenId) !=
listing.request.owner) {
            revert NotOwner();
        }

        if (!SignatureChecker.isValidSignatureNow(listing.request.owner, listingHash,
listing.sig)) {
            revert InvalidSignature();
        }

        if (msg.sender != listing.receiver && !
SignatureChecker.isValidSignatureNow(listing.receiver, listingHash, listing.receiverSig)) {
            revert InvalidSignature();
        }

        return true;
    }
   ....
```

This oversight can lead to significant issues:

- In the `buy` function, the function directly uses `acceptedCurrency` to pay. If the `acceptedCurrency` has changed between the time of signing and execution, it could result in incorrect charges (if the payee happens to have approved enough allowance) or a DOS.

```
            // External calls after state changes
            IERC20(acceptedCurrency).transferFrom(payee, listings[i].request.owner,
sellerAmount);
            IERC20(acceptedCurrency).transferFrom(payee, address(this), feeAmount);
```

- More significantly, In the `acceptOffer` function, the function uses `offers[i].request.offerCurrency` to pay. This bypasses the `acceptedCurrency` check, potentially leading to incorrect calculations of `totalPendingFees` and making it impossible to retrieve the actual fee.

```
            // Update state before external calls
            totalPendingFees += feeAmount;
            // External calls after state changes
            IERC20(offers[i].request.offerCurrency).transferFrom(offers[i].request.requester,
offers[i].receiver, sellerAmount);
            IERC20(offers[i].request.offerCurrency).transferFrom(offers[i].request.requester,
address(this), feeAmount);
```

- Furthermore, if `acceptedCurrency` is set to zero during an `emergencyShutdown`, transactions of `acceptOffer` could still be completed, undermining the purpose of the emergency shutdown mechanism.

```
    function emergencyShutdown() external {
        require(msg.sender == marketplaceAdmin, "Only marketplaceAdmin can call");
        // First collect any pending fees
        uint256 amount = totalPendingFees;
        if (amount > 0 && acceptedCurrency != address(0)) {
            totalPendingFees = 0;
            IERC20(acceptedCurrency).transfer(feeReceiver, amount);
        }
        // Set currency to address(0) to prevent any new trades
        acceptedCurrency = address(0);
    }
```

Recommendations

To enhance the integrity of the contract and prevent these issues, implement validation checks to ensure that the currency used in both `ListingRequest` and `OfferRequest` matches the `acceptedCurrency` before executing any transactions.

[H-06] Incorrect `pendingCounts` calculation

Severity

Impact: High

Likelihood: Medium

Description

In SpinLottery contract, users can spin to win one prize NFT. The spin result is based on the randomness from Chainlink VRF. The spin result is asynchronous. When users trigger `spin` function, we will calculate the `pendingCounts`. Because there are some previous `spin` transactions, these transactions' results are unknown, and these `spin` transactions may win and claim some prize NFTs from the contract. So when we trigger `spin`, we need to check current prize in the pool and also the potential claimed prize.

The `pendingCounts` is used to record the potential claimed prize NFT before this `spin`.

According to current implementation, if the user wins in one spin, the user will gain one prize NFT. The problem here is that in function `calculatePendingPrizesForRarity`, when we calculate the `pendingCount`, the `pendingCount` is related to `_prizeCount`, `weight`, `totalRarityWeight`. The `_prizeCount` can be controlled by users. When users trigger `spin` with one large `_prizeCount`, we will get one large `pendingCount`. The `pendingCount` will be updated into `prizePools[i].pendingCount`.

This will cause the later `spin` may be blocked.

In function `calculatePendingPrizesForRarity`, the parameter `_prizeCount` is expected to be the prize count that we will distribute according to the commnet. However, this is incorrect. The `_prizeCount` can be controlled by the user, and no matter which value `_prizeCount` is, the distribute prize count for one spin will always be `1`. This will cause some logic related to `_prizeCount` in `spin` are incorrect.

```
    function spin(uint256 _totalSlots, uint256 _prizeCount) external whenNotPaused returns
(uint256) {
        for (uint8 i = 1; i <= maxRarityId; i++) {
            if (rarityConfigs[i].active) {
                pendingCounts[i] = calculatePendingPrizesForRarity(_prizeCount, i);
                totalPending += pendingCounts[i];
            }
        }
        if (totalPending == 0) {
            for (uint8 i = 1; i <= maxRarityId; i++) {
                if (rarityConfigs[i].active) {
                    pendingCounts[i] = 1;
                    totalPending = 1;
                    break;
                }
            }
        }
        for (uint8 i = 1; i <= maxRarityId; i++) {
            if (pendingCounts[i] > 0) {
                uint256 available = getAvailablePrizes(i);
                uint256 pending = prizePools[i].pendingCount;
                if (available - pending < pendingCounts[i]) {
                    revert InsufficientPrizes();
                }
            }
        }
        for (uint8 i = 1; i <= maxRarityId; i++) {
            if (pendingCounts[i] > 0) {
                prizePools[i].pendingCount += uint96(pendingCounts[i]);
            }
        }
    }
    function calculatePendingPrizesForRarity(uint256 _prizeCount, uint8 rarity) internal view
returns (uint256) {
        if (_prizeCount == 1) {
            // For single prizes, prioritize first active rarity
            return rarity == 1 && rarityConfigs[1].active ? 1 : 0;
        }
        if (!rarityConfigs[rarity].active) return 0;

        uint256 weight = rarityConfigs[rarity].weight;
@>        uint256 pendingCount = (_prizeCount * weight) / totalRarityWeight;

        if (_prizeCount > 1 && pendingCount == 0 && weight > 0) {
            pendingCount = 1;
        }

        return pendingCount;
    }
```

Recommendations

Revisit the `pendingCount`'s calculation.

[H-07] `SpinLottery` will break if the first `prizeId` is selected

Severity

Impact: High

Likelihood: Medium

Description

Inside the `_claimPrize` logic, if the randomly selected `prizeId` is equal to `firstPrizeId`, `firstPrizeId` will be incremented. However, `nextPrizeId` is also decremented.

```
    function _claimPrize(uint8 rarity, uint256 randomValue) internal returns (address
nftAddress, uint88 tokenId) {
        PrizePool storage pool = prizePools[rarity];
        if (pool.nextPrizeId <= pool.firstPrizeId) revert NoPrizesAvailable();

        // Calculate prize count and random index
        uint256 prizeCount = pool.nextPrizeId - pool.firstPrizeId;
        uint256 index = randomValue % prizeCount;
        uint256 prizeId = pool.firstPrizeId + index;

        // Get prize data
        uint256 packed = pool.prizeData[prizeId];

        // Remove prize using unordered removal pattern - more gas efficient than array
management
        // When a prize is removed, we move the last prize to its place if it's not already the
last
        if (prizeId != pool.nextPrizeId - 1) {
            uint256 lastPrizeId = pool.nextPrizeId - 1;
            pool.prizeData[prizeId] = pool.prizeData[lastPrizeId];
            delete pool.prizeData[lastPrizeId];
        } else {
            delete pool.prizeData[prizeId];
        }

        // If we've removed the first prize, increment firstPrizeId
        if (prizeId == pool.firstPrizeId) {
>>>         pool.firstPrizeId++;
        }

        // Decrement nextPrizeId
>>>     pool.nextPrizeId--;

        uint8 unpackedRarity;
        (nftAddress, tokenId, unpackedRarity) = unpackPrize(packed);
        assert(unpackedRarity == rarity);
    }
```

This will cause the prize selection function to break for the corresponding rarity. Consider the following scenario:

- `pool.firstPrizeId` = 1 and `pool.nextPrizeId` = 3, meaning there are 2 possible prizes.
- A user spins and wins the first prize (`prizeId` = 1).
- `pool.firstPrizeId++` and `pool.nextPrizeId--` are performed, resulting in `pool.firstPrizeId` = 2 and `pool.nextPrizeId` = 2, meaning the other prize is lost and can no longer be played and claimed by users.

Recommendations

Remove the `pool.firstPrizeId++` logic, as it is unnecessary because the `lastPrizeId` is moved to the first position.

Medium findings

[M-01] Missing sanity check in `addCardBundlesToPacketPool`

Severity

Impact: Medium

Likelihood: Medium

Description

In CardAllocationPool contract, we will generate one randomness via Chainlink VRF for the instant redeem. In initialize() function, we will initialize `callbackGasLimit` to 100_000. This gas limit is used for the Chainlink VRF callback gas limit.

In the callback function, there is one loop for transferring the selected cards. After the foundry's test, if we have 10 cards in this bundle, the gas limit will be insufficient. This will cause the callback function will fail, and users cannot get the card NFT.

Function `addCardBundlesToPacketPool` is used to add one card bundle. The problem here is that we don't add any sanity check about the card bundle's length.

```
    function initialize(
        address packetAddress,
        address cardAddress,
        address _rbac,
        uint256 _redeemManagerRoleId,
        address vrfCoordinator,
        uint256 subscriptionId,
        bytes32 _keyHash
    ) external initializer {
        callbackGasLimit = 100000;
    }
    function fulfillRandomWords(uint256 requestId, uint256[] memory randomWords) internal {
        PacketOpenRequest storage request = requestIdToPacketOpen[requestId];
        if (request.fulfilled) revert("Already fulfilled");

        CardBundle[] storage cardBundles = packetTypeToCardBundles[request.packetType];

        uint256[] memory selectedCards = selectRandomCards(cardBundles, randomWords[0]);

@>        for (uint256 i = 0; i < selectedCards.length; i++) {
            IERC721(cardNFTAddress).transferFrom(address(this), request.owner,
selectedCards[i]);
        }

        request.fulfilled = true;

        RipFunPacks(packetNFTAddress).finalizeOpen(request.packetId, "");
        emit PacketOpenFulfilled(requestId, request.packetId, selectedCards);
    }
    function addCardBundlesToPacketPool(uint256 packetType, uint256[] memory cardBundles,
bytes32 bundleProvenance)
        external
        onlyRole(redeemManagerRoleId)
    {
        packetTypeToCardBundles[packetType].push(CardBundle({cardIds: cardBundles,
bundleProvenance: bundleProvenance}));
        for (uint256 i = 0; i < cardBundles.length; i++) {
            IERC721(cardNFTAddress).transferFrom(msg.sender, address(this), cardBundles[i]);
        }

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

Recommendations

Add a sanity check for the card bundle's length.

[M-02] Lack of nonce mechanism in signature hash generation

Severity

Impact: Medium

Likelihood: Medium

Description

The `generateListingSignatureHash` function in the Marketplace contract creates a hash for a listing request that does not include a nonce mechanism.

```
    function generateListingSignatureHash(ListingRequest calldata listing) public view returns
(bytes32) {
        return _hashTypedDataV4(
            keccak256(
                abi.encode(
                    LISTING_TYPEHASH,
                    listing.tokenContractAddress,
                    listing.tokenId,
                    listing.price,
                    listing.acceptedCurrency,
                    listing.deadline,
                    listing.owner,
                    listing.chainId
                )
            )
        );
    }
```

This omission means that once the owner signs the listing, the signature remains valid (until `deadline`) even if the owner subsequently modifies the price. **And once a signature is signed, there is no way to revoke it.**

This creates a vulnerability where a malicious attacker could withhold the owner's signature until the `deadline` . If the price of the asset increases (a common occurrence with volatile assets), the attacker could use the original signed price to execute a purchase, undermining the integrity of the listing process and potentially leading to financial losses for the seller.

```
        bytes32 listingHash = generateListingSignatureHash(listing.request);
        if (usedListings[listingHash]) revert ListingAlreadyUsed();
        if (!
IWhitelistRegistry(whitelistRegistry).isWhitelisted(listing.request.tokenContractAddress)) {
            revert NFTNotWhitelisted();
        }
        if (IERC721(listing.request.tokenContractAddress).ownerOf(listing.request.tokenId) !=
listing.request.owner) {
            revert NotOwner();
        }

        if (!SignatureChecker.isValidSignatureNow(listing.request.owner, listingHash,
listing.sig)) {
            revert InvalidSignature();
        }
```

This is the same for other signature generation in `isValidTrade` and `isValidBid` .

Recommendations

To enhance the security of the listing process, implement a nonce mechanism that is included in the signature hash generation. This nonce should be unique for each listing and should be incremented with each new listing request from the owner. **More specifically, the owner should be able to disable a signature once it becomes invalid.**

[M-03] Inconsistent trade fees from `tradeFee` and `acceptedCurrency` mods

Severity

Impact: Medium

Likelihood: Medium

Description

In the Marketplace contract, the `tradeFee` is charged in the `acceptedCurrency` , which can be modified independently of the fee itself.

```
            // Update state before external calls
            totalPendingFees += tradeFee;

            // External calls after state changes
            // take fee from requester
            IERC20(acceptedCurrency).transferFrom(trades[i].request.requester, address(this),
tradeFee);
```

```
    /**
     * @notice Updates the trade fee (called by MarketplaceAdmin)
     * @dev Only callable by the MarketplaceAdmin contract
     * @param newTradeFee New trade fee amount
     */
    function setTradeFee(uint256 newTradeFee) external {
        require(msg.sender == marketplaceAdmin, "Only marketplaceAdmin can call");
        tradeFee = newTradeFee;
    }


    /**
     * @notice Updates the accepted currency (called by MarketplaceAdmin)
     * @dev Only callable by the MarketplaceAdmin contract
     * @param newERC20TokenAddress Address of the new ERC20 token to be accepted
     */
    function setAcceptedCurrency(address newERC20TokenAddress) external {
        require(msg.sender == marketplaceAdmin, "Only marketplaceAdmin can call");
        if (newERC20TokenAddress == address(0)) revert InvalidCurrencyAddress();
        acceptedCurrency = newERC20TokenAddress;
    }
```

This creates a potential risk when both the `tradeFee` and `acceptedCurrency` are changed in separate transactions. If a user initiates a payment between these two modifications, it could lead to unintended consequences, such as overpayment or underpayment.

For example, if the decimals of the `acceptedCurrency` change from `6` (as with `USDC`) to `18` (as with `DAI`), a `tradeFee` of 1e6 would effectively become negligible, resulting in a significant loss of revenue for the marketplace.

Recommendations

Require that any changes to `acceptedCurrency` should also change `acceptedCurrency` in the same transaction to ensure that both values are updated simultaneously.

[M-04] EIP-712 compliance Issue in `Marketplace`

Severity

Impact: Medium

Likelihood: Medium

Description

The `TRADE_TYPEHASH` in the Marketplace contract is computed using a struct that references `TradeGive` and `TradeTake`. However, the current implementation does not include the definitions or type hashes for these referenced structs.

```
    /**
     * @notice The EIP-712 type hash for trade requests.
     */
    bytes32 public constant TRADE_TYPEHASH = keccak256(
        "TradeRequest(TradeGive[] give,TradeTake[] take,uint256 deadline,address
requester,uint256 chainId)"
    );
```

According to EIP-712:

> If the struct type references other struct types (and these in turn reference even more struct types), then the set of referenced struct types is collected, sorted by name and appended to the encoding. An example encoding is `Transaction(Person from,Person to,Asset tx)Asset(address token,uint256 amount)Person(address wallet,string name)`.

Also, when calculating `generateTradeSignatureHash`, the `trade.give` and `trade.take` should be `hashed` values since they are dynamic types.

```
    function generateTradeSignatureHash(TradeRequest calldata trade) public view returns
(bytes32) {
        return _hashTypedDataV4(
            keccak256(
                abi.encode(
                    TRADE_TYPEHASH,
                    trade.give,
                    trade.take,
                    trade.deadline,
                    trade.requester,
                    trade.chainId
                )
            )
        );
    }
```

This omission violates the `EIP-712` specification, which could lead to incorrect encoding of potential vulnerabilities in the signing and verification process of requests.

Recommendations

It is recommended to ensure compliance with the `EIP-712` standard.

[M-05] Potential out-of-bounds in `removeCardBundlesFromPacketPool`

Severity

Impact: Medium

Likelihood: Medium

Description

The `removeCardBundlesFromPacketPool` function in the `CardAllocationPool` contract has a critical flaw that could lead to `out-of-bounds` access errors.

When removing card bundles, the function **swaps the selected bundle with the last unselected bundle and then pops the last element from the array.**

```
        // Swap selected card bundle with last unselected card bundle and pop
        cardBundles[index] = cardBundles[cardBundles.length - 1];
        cardBundles.pop();
```

However, if the function attempts to remove bundles in a sequential order (e.g., removing `bundle0` and `bundle2` from an array of three bundles), the indices of the remaining bundles will change after the first removal and the length will decrease by `1`.

For instance, if the initial array is `[bundle0, bundle1, bundle2]` and `bundle0` is removed first, the array becomes `[bundle2, bundle1]`, but the function may still attempt to access `cardBundles[2]` in order to remove `bundle2`, which no longer exists in that index, leading to a potential runtime error.

Recommendations

Modify the logic or the input `cardBundleIndexes` should be in descending order.

[M-06] Improper randomness requests in `CardAllocationPool` and `SpinLottery`

Severity

Impact: Medium

Likelihood: Medium

Description

According to best practices outlined by Chainlink, **once a contract has submitted a randomness request, it should not accept any additional user-supplied inputs that could affect the outcome. This is to ensure fairness and prevent potential griefing attacks.**

> Generally speaking, whenever an outcome in your contract depends on some user-supplied inputs and randomness, the contract should not accept any additional user-supplied inputs after it submits the randomness request.

In the `CardAllocationPool` contract, the `selectedCards` are determined based on the `packetTypeToCardBundles` mapping.

```
    function fulfillRandomWords(uint256 requestId, uint256[] memory randomWords) internal {
        PacketOpenRequest storage request = requestIdToPacketOpen[requestId];
        if (request.fulfilled) revert("Already fulfilled");

        // fetch the available card bundles
        CardBundle[] storage cardBundles = packetTypeToCardBundles[request.packetType];

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

        // Transfer cards to owner
        for (uint256 i = 0; i < selectedCards.length; i++) {
            IERC721(cardNFTAddress).transferFrom(address(this), request.owner,
selectedCards[i]);
        }

        // Mark request as fulfilled
        request.fulfilled = true;

        // Call finalize open on packet contract
        RipFunPacks(packetNFTAddress).finalizeOpen(request.packetId, "");

        emit PacketOpenFulfilled(requestId, request.packetId, selectedCards);
    }
```

However, the `addCardBundlesToPacketPool` function allows modifications to this input, which can be called by the user with the `redeemManagerRole`.

```
    function addCardBundlesToPacketPool(uint256 packetType, uint256[] memory cardBundles,
bytes32 bundleProvenance)
        external
        onlyRole(redeemManagerRoleId)
    {
        packetTypeToCardBundles[packetType].push(CardBundle({cardIds: cardBundles,
bundleProvenance: bundleProvenance})); // <= changed here
        for (uint256 i = 0; i < cardBundles.length; i++) {
            IERC721(cardNFTAddress).transferFrom(msg.sender, address(this), cardBundles[i]);
        }
        emit CardsAddedToPool(packetType, cardBundles, bundleProvenance);
    }
```

At the same time, the process of `fulfillRandomWords` for `requestA` could also impact `requestB` since `selectedCardBundle` is popped out from `availableCardBundles`.

```
        selectedCardBundle = availableCardBundles[index].cardIds;

        // Swap selected card bundle with last unselected card bundle and pop
        availableCardBundles[index] = availableCardBundles[availableCardBundles.length - 1];
        availableCardBundles.pop();
```

Additionally, `removeCardBundlesFromPacketPool` could also change the corresponding `packetTypeToCardBundles[packetType]`:

```
    function removeCardBundlesFromPacketPool(uint256 packetType, uint256[] memory
cardBundleIndexes)
        external
        onlyRole(redeemManagerRoleId)
    {
        CardBundle[] storage cardBundles = packetTypeToCardBundles[packetType];

        for (uint256 i = 0; i < cardBundleIndexes.length; i++) {
            uint256 index = cardBundleIndexes[i];

            // transfer the cards to the caller
            for (uint256 j = 0; j < cardBundles[index].cardIds.length; j++) {
                IERC721(cardNFTAddress).transferFrom(address(this), msg.sender,
cardBundles[index].cardIds[j]);
            }

            // Swap selected card bundle with last unselected card bundle and pop
            cardBundles[index] = cardBundles[cardBundles.length - 1];
            cardBundles.pop();
        }

        emit CardBundlesRemovedFromPool(packetType, cardBundleIndexes);
    }
```

This design violates the recommended practice, as it permits changes to the card bundles after a randomness request has been made, potentially skewing the selection of cards and compromising the integrity of the outcome.

Similarly, this also applies to `SpinLottery`, while after a `Randomness Requests` is being made, `rarity weight` and `card` could also be changed via `configureRarity` and `addPrize`, even after the user has paid the `spin` fee. However, the situation could be completely different for the user.

```
    function configureRarity(uint8 _rarityId, uint8 _weight, uint160 _basePrice, bool _active)
        external
        onlyRole(lotteryManagerRoleId)
    {
        if (_rarityId == 0) revert InvalidRarity();
        if (_rarityId > MAX_RARITIES) revert MaxRaritiesExceeded();

        // If this is a new rarity, track the highest rarity ID
        if (_rarityId > maxRarityId) {
            maxRarityId = _rarityId;
        }

        // Update total weight (subtract old weight if exists, add new weight)
        RarityConfig memory oldConfig = rarityConfigs[_rarityId];
        if (oldConfig.active) {
            totalRarityWeight -= oldConfig.weight;
        }

        // Only add new weight if rarity will be active
        if (_active) {
            totalRarityWeight += _weight;
        }

        // Update configuration
        rarityConfigs[_rarityId] = RarityConfig({
            weight: _weight,
            basePrice: _basePrice,
            active: _active
        });

        emit RarityConfigUpdated(_rarityId, _weight, _basePrice, _active);
    }
```

```
    function addPrize(address _nftAddress, uint256 _tokenId, uint8 _rarity) external
onlyRole(lotteryManagerRoleId) {
        if (_nftAddress == address(0)) revert ZeroAddress();
        if (_tokenId > type(uint88).max) revert TokenIdTooLarge();
        if (_rarity == 0 || _rarity > maxRarityId || !rarityConfigs[_rarity].active) revert
InvalidRarity();

        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);
    }
```

Recommendations

Evaluate the Impact of Randomness Requests and **Decide on whether following Best Practice Or Not.**

[M-07] Non-compliance with EIP-721 in `tokenURI()` function

Severity

Impact: Low

Likelihood: High

Description

Similar Finding: Link

The `tokenURI` function in the `RipFunPacks` contract does not strictly adhere to the EIP-721 standard.

```
function tokenURI(uint256 tokenId) public view override returns (string memory) {
    return _packetUris[tokenId];
}
```

According to the EIP-721 specification, the function should throw an error if the provided _tokenId does not correspond to a valid NFT.

```
/// @notice A distinct Uniform Resource Identifier (URI) for a given asset.
/// @dev Throws if `_tokenId` is not a valid NFT. URIs are defined in RFC
///  3986. The URI may point to a JSON file that conforms to the "ERC721
///  Metadata JSON Schema".
function tokenURI(uint256 _tokenId) external view returns (string);
```

However, the current implementation simply returns an empty string for non-existent tokenId values. This behavior can lead to confusion and makes it difficult for clients and applications to determine the validity of a token.

Note: this is the same for `Card` contract.

```
/**
 * @dev Returns the metadata URI for a given card
 * @param tokenId ID of the card
 * @return Metadata URI for the card
 */
function tokenURI(uint256 tokenId) public view override returns (string memory) {
    return _cardUris[tokenId];
}
```

Recommendations

To ensure compliance with the EIP-721 standard, modify the `tokenURI` function to revert when an invalid `tokenId` is provided.

[M-08] Inconsistent logic in `initiateBurn()` and `initiateBurnBatch()` functions

Severity

Impact: Medium

Likelihood: Medium

Description

The `initiateBurnBatch` function in the Packet contract exhibits inconsistent logic compared to the `initiateBurn` function.

In `initiateBurn`, if the `burnType` is `BurnType.INSTANT_OPEN_PACKET`, the function calls `instantOpenPacket` to handle the burn operation.

```
        if (params.burnType == BurnType.INSTANT_OPEN_PACKET) {
            randomAllocationPool.instantOpenPacket(params.packetId,
_packetTypeIds[params.packetId], msg.sender);
        }
```

However, the `initiateBurnBatch` function lacks similar logic to process `BurnType.INSTANT_OPEN_PACKET`, which means that batch burning does not trigger the same behavior as individual burning. This inconsistency could lead to unexpected outcomes, **where packets intended for instant opening are not handled correctly when burned in a batch.**

```
    function initiateBurnBatch(PacketBurnParams[] calldata params) external {
        for (uint256 i = 0; i < params.length; i++) {
            if (params[i].burnType == BurnType.NONE) {
                revert InvalidBurnType();
            }
            if (ownerOf(params[i].packetId) != msg.sender) {
                revert NotPacketOwner();
            }
            if (_packetBurnType[params[i].packetId] != BurnType.NONE) {
                revert PacketAlreadyInBurnState();
            }

            _packetBurnType[params[i].packetId] = params[i].burnType;
            emit BurnInitiated(params[i].packetId, params[i].burnType, msg.sender);
        }
    }
```

Recommendations

To ensure consistency between the two functions, the `initiateBurnBatch` function should include logic to handle the `BurnType.INSTANT_OPEN_PACKET` case. Or simply prohibit the usage of `BurnType.INSTANT_OPEN_PACKET` in `initiateBurnBatch`.

[M-09] Role management vulnerability in `assignRole()` function

Severity

Impact: High

Likelihood: Low

Description

Reference Finding: [Link]()

The `assignRole` function in the `RoleBasedAccessControl` contract is protected by the `onlySuperAdminOrRoleManager` modifier, which allows both `SUPER_ADMIN_ROLE` and `ROLE_MANAGER_ROLE` to invoke it.

```
  function assignRole(uint256 roleId, address receiver) external onlySuperAdminOrRoleManager {
     ...
  }
```

While the function correctly prevents the assignment of the `SUPER_ADMIN_ROLE`, it does not restrict a user with `ROLE_MANAGER_ROLE` from assigning the `ROLE_MANAGER_ROLE` to others.

```
        if (roleId == SUPER_ADMIN_ROLE) {
            revert SuperAdminRoleReserved();
        }
        if (roleId == 0) {
            revert InvalidRoleId();
        }
```

This creates a potential scenario where a malicious `ROLE_MANAGER_ROLE` user could exploit this function to maintain or enhance their privileges, leading to administrative confusion and potential abuse of power.

Consider the following scenario: 1. A `SUPER_ADMIN_ROLE` attempts to revoke permissions from a user holding the `ROLE_MANAGER_ROLE` using the `revokeRole` function. 2. A malicious admin anticipates this action and executes a front-run by calling the `assignRole` function to grant `ROLE_MANAGER_ROLE` privileges to another user under their control. 3. The revocation transaction from step 1 is processed successfully, but the malicious admin retains access to the system through the newly empowered user, undermining the intended security measures.

Note: Same applies for `revokeRole`, as a `ROLE_MANAGER_ROLE` user could revoke the `ROLE_MANAGER_ROLE` role from others.

Recommendations

To mitigate this risk, it is essential to implement additional checks within the `assignRole` function to prevent a `ROLE_MANAGER_ROLE` from assigning the same role to themselves or others.

[M-10] Spin cost fees are stuck in the SpinLottery

Severity

Impact: Medium

Likelihood: Medium

Description

In SpinLottery, users can trigger `spin` to win one prize NFT. In order to join the activity, users need to transfer some spin cost to SpinLottery contract.

The problem here is that we miss one interface to withdraw the spin costs fees. These spin cost fees will be locked in the contract.

```
    function spin(uint256 _totalSlots, uint256 _prizeCount) external whenNotPaused returns
(uint256) {
        uint256 spinCost = calculateSpinCost(_totalSlots, _prizeCount);

        // Process payment
        if (!usdcToken.transferFrom(msg.sender, address(this), spinCost)) {
            revert USDCTransferFailed();
        }
    }
```

Recommendations

Add one interface to withdraw spin cost fees.

[M-11] Incorrect initial `tradeFee` in `marketPrice`

Severity

Impact: Medium

Likelihood: Medium

Description

In marketPrice contract, when users trade their NFTs via function `executeTrade`, the protocol will charge some trade fees from this trade.

When we initialize the marketPrice contract, we will initialize the `tradeFee`. According to the comment, the initial trade fee is 1 USDC. The problem here is that we use the incorrect value `10e6` here. `10e6` USDC means 10 USDC. So traders have to pay 10X trade fees than expected.

```
    function initialize(
        address _rbac,
        uint256 _marketplaceManagerRoleId,
        address _whitelistRegistry,
        address _acceptedCurrency,
        address _feeReceiver
    ) external initializer {
        ...
        tradeFee = 10e6; // 1 USDC
    }
```

Recommendations

Initialize the trade fee with the correct value `1e6`.

[M-12] Inconsistent `pendingCounts` in `spin()` and `fulfillRandomness()` when 0

Severity

Impact: High

Likelihood: Low

Description

Inside `spin`, when `totalPending` equals 0, it will set the first active rarity `pendingCounts` to 1.

```
    function spin(uint256 _totalSlots, uint256 _prizeCount) external whenNotPaused returns
(uint256) {
        // Validate slot configuration
        if (_totalSlots == 0 || _prizeCount == 0 || _prizeCount > _totalSlots) {
            revert InvalidSlotConfiguration();
        }

        // Calculate and verify prize distribution
        uint256[] memory pendingCounts = new uint256[](maxRarityId + 1);
        uint256 totalPending = 0;

        for (uint8 i = 1; i <= maxRarityId; i++) {
            if (rarityConfigs[i].active) {
                // @audit - each loop, price count should be decreased?
                pendingCounts[i] = calculatePendingPrizesForRarity(_prizeCount, i);
                totalPending += pendingCounts[i];
            }
        }

        // Ensure at least one prize will be allocated
        if (totalPending == 0) {
            // Find first active rarity and allocate one prize
>>>         for (uint8 i = 1; i <= maxRarityId; i++) {
                if (rarityConfigs[i].active) {
                    pendingCounts[i] = 1;
                    totalPending = 1;
                    break;
                }
            }
        }
        // Check if enough prizes are available considering pending ones
        for (uint8 i = 1; i <= maxRarityId; i++) {
            if (pendingCounts[i] > 0) {
                uint256 available = getAvailablePrizes(i);
                uint256 pending = prizePools[i].pendingCount;

                if (available - pending < pendingCounts[i]) {
                    revert InsufficientPrizes();
                }
            }
        }

       // ...

        // Increment pending counts
        for (uint8 i = 1; i <= maxRarityId; i++) {
            if (pendingCounts[i] > 0) {
                prizePools[i].pendingCount += uint96(pendingCounts[i]);
            }
        }

       // ...
    }
```

However, the same case is not considered when `fulfillRandomness` is triggered, which could cause an incorrect `pendingCount` for the corresponding rarity.

Recommendations

Consider also considering `totalPending` equals 0 inside `fulfillRandomness`.

[M-13] Buy request executed by providing `request.owner`

Severity

Impact: Medium

Likelihood: Medium

Description

When `buy` is executed, it first validates `isValidListing`, ensuring that `request.owner` has indeed signed the listing, and that listing.receiver has also signed the listing if the caller is not the `listing.receiver`.

```
function isValidListing(ListingParams calldata listing) public view returns (bool) {
        if (listing.request.price == 0) revert InvalidPrice();
        if (block.timestamp > listing.request.deadline) revert ListingExpired();
        if (listing.request.chainId != block.chainid) revert InvalidChainId();
        bytes32 listingHash = generateListingSignatureHash(listing.request);
        if (usedListings[listingHash]) revert ListingAlreadyUsed();
        if (!
IWhitelistRegistry(whitelistRegistry).isWhitelisted(listing.request.tokenContractAddress)) {
            revert NFTNotWhitelisted();
        }
        if (IERC721(listing.request.tokenContractAddress).ownerOf(listing.request.tokenId) !=
listing.request.owner) {
            revert NotOwner();
        }

>>>     if (!SignatureChecker.isValidSignatureNow(listing.request.owner, listingHash,
listing.sig)) {
            revert InvalidSignature();
        }

>>>     if (msg.sender != listing.receiver && !
SignatureChecker.isValidSignatureNow(listing.receiver, listingHash, listing.receiverSig)) {
            revert InvalidSignature();
        }

        return true;
    }
```

However, if the request owner has `acceptedCurrency` allowance to the contract, this could allow an attacker to trigger `buy`, designate the owner as the receiver, and provide a valid signature, causing the `buy` to be executed. This would result in the listing owner's request being fulfilled and the fee being paid to the contract.

Recommendations

Prevent `buy` operation from being executed if the provided receiver is the owner.

[M-14] Upgrade permissions inappropriate for `ROLE_MANAGER_ROLE`

Severity

Impact: Medium

Likelihood: Medium

Description

The `_authorizeUpgrade` function in the `RoleBasedAccessControl` contract allows both `SUPER_ADMIN_ROLE` and `ROLE_MANAGER_ROLE` to authorize contract upgrades.

```
    function _authorizeUpgrade(address newImplementation) internal override
onlySuperAdminOrRoleManager {}
```

This design violates the best practice of **maintaining a single point of truth** for upgrade authority. Contract upgrades are a highly sensitive operation and should be restricted to a single, clearly defined, and trusted role to avoid ambiguity and reduce the risk of misconfiguration or abuse. Granting this power to multiple roles introduces unnecessary complexity and increases the attack surface.

In this context, the `ROLE_MANAGER_ROLE` should not possess the authority to perform upgrades, as it is typically not assumed to have the same level of trust or oversight as the `SUPER_ADMIN_ROLE`. A compromised or malicious `ROLE_MANAGER_ROLE` user could abuse this capability to introduce malicious logic or backdoors.

Note that other contracts in the codebase follow a more appropriate pattern where only one specific role is allowed to perform upgrades. For example:

```
    /// @notice Role ID for super admin, used for contract upgrades
    /// @dev Constant value that defines the role identifier for upgrade authorization
    uint256 public constant SUPER_ADMIN_ROLE = 1;
```

The same goes for `MINT_MANAGER_ROLE` in the `Card` contract and `redeemManagerRoleId` in `CardAllocationPool`, as it is used as the role for upgrade. Under this situation, even `SUPER_ADMIN_ROLE` couldn't upgrade the contract.

```
    function _authorizeUpgrade(address newImplementation) internal override
onlyRole(MINT_MANAGER_ROLE) {}
```

```
    function _authorizeUpgrade(address newImplementation) internal override
onlyRole(redeemManagerRoleId) {}
```

Recommendations

To enhance the security of the upgrade mechanism, restrict the upgrade authorization solely to the `SUPER_ADMIN_ROLE`.

Low findings

[L-01] No NFT retrieval mechanism in `SpinLottery` for inactive rarities

In the `SpinLottery` contract, once an NFT is added to the prize pool, there is no mechanism for retrieving it other than through the lottery process.

```
    function _distributePrize(address player, uint256 requestId, uint8 rarity, uint256
prizeRandom) external {
        // Only allow this contract to call this function
        if (msg.sender != address(this)) revert("Unauthorized");

        (address nftAddress, uint88 tokenId) = _claimPrize(rarity, prizeRandom);
        IERC721(nftAddress).transferFrom(address(this), player, tokenId);
        emit PrizeWon(player, requestId, nftAddress, tokenId, rarity);
    }
```

If a rarity is marked as inactive via `configureRarity`, any NFTs associated with that rarity become effectively locked and cannot be retrieved.

```
        // Update configuration
        rarityConfigs[_rarityId] = RarityConfig({
            weight: _weight,
            basePrice: _basePrice,
            active: _active
        });
```

To address this issue, implement a mechanism that allows admins to retrieve their NFTs if the associated `rarity` becomes inactive.

[L-02] Zero prize locking for single prizes due to implicit condition

In the SpinLottery contract, the logic for handling single prize requests contains an implicit condition that could lead to unintended outcomes.

```
    function calculatePendingPrizesForRarity(uint256 _prizeCount, uint8 rarity) internal view
returns (uint256) {
        // Handle special case for single prize
        if (_prizeCount == 1) {
            // For single prizes, prioritize first active rarity
            return rarity == 1 && rarityConfigs[1].active ? 1 : 0;
        }

        if (!rarityConfigs[rarity].active) return 0;

        // Calculate based on weight distribution
        uint256 weight = rarityConfigs[rarity].weight;
        uint256 pendingCount = (_prizeCount * weight) / totalRarityWeight;
        // Ensure at least some prizes are allocated if weights allow it
        if (_prizeCount > 1 && pendingCount == 0 && weight > 0) {
            pendingCount = 1;
        }

        return pendingCount;
    }
```

Specifically, when `_prizeCount` is set to `1`, the function returns `rarity == 1 && rarityConfigs[1].active ? 1 : 0`. This means that if the first `rarity` is `inactive` (`rarityConfigs[1]`), users will be unable to lock any prizes, even if he has already paid the spin fee.

[L-03] Precision loss from `Divide-Before-Multiply` in price calculation

In the `SpinLottery` contract, the calculation of `avgBasePrice` in the function `calculateSpinCost` involves a division operation before a multiplication operation.

```
        // Calculate weighted average price across all active rarities
        for (uint8 i = 1; i <= maxRarityId; i++) {
            RarityConfig memory config = rarityConfigs[i];
            if (config.active) {
                totalWeightedPrice += uint256(config.basePrice) * config.weight;
            }
        }
```

This approach can lead to precision loss, which is not in favor of the protocol. To mitigate the risk of precision loss, adjust the calculation to perform the division after the multiplication.

[L-04] `getAvailablePrizes` may return incorrect result

The `getAvailablePrizes` function in the `SpinLottery` contract retrieves the number of available prizes for a specified `rarity` level.

```
    function getAvailablePrizes(uint8 rarity) public view returns (uint256) {
        if (rarity == 0 || rarity > maxRarityId) revert InvalidRarity();
        PrizePool storage pool = prizePools[rarity];
        return pool.nextPrizeId > pool.firstPrizeId ? pool.nextPrizeId - pool.firstPrizeId : 0;
    }
```

However, **it does not check whether the** `rarity` **is currently** `active`. If a rarity is marked as inactive, any prizes associated with that rarity should be considered unavailable and not counted towards the total available prizes.

```
    function calculatePendingPrizesForRarity(uint256 _prizeCount, uint8 rarity) internal view
returns (uint256) {
        // Handle special case for single prize
        if (_prizeCount == 1) {
            // For single prizes, prioritize first active rarity
            return rarity == 1 && rarityConfigs[1].active ? 1 : 0;
        }

        if (!rarityConfigs[rarity].active) return 0;
        ...
    }
```

This oversight could lead to misleading information being presented to users. To enhance the accuracy of the prize availability check, implement a validation step in the `getAvailablePrizes` function to verify the active status of the specified rarity.

[L-05] No duplicate prize check in prize pool addition

In the `SpinLottery` contract, the function responsible for adding prizes checks whether the contract currently owns the specified NFT by verifying `nft.ownerOf(_tokenId) != address(this)`. However, it does not include a check to determine if the `packed` prize (which combines the NFT address, token ID, and rarity) has already been added to the prize pool.

```
    function addPrize(address _nftAddress, uint256 _tokenId, uint8 _rarity) external
onlyRole(lotteryManagerRoleId) {
        if (_nftAddress == address(0)) revert ZeroAddress();
        if (_tokenId > type(uint88).max) revert TokenIdTooLarge();
        if (_rarity == 0 || _rarity > maxRarityId || !rarityConfigs[_rarity].active) revert
InvalidRarity();

        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);
    }
```

This oversight could lead to potential duplicate entries in the prize pool, affecting the `spin` process.

To prevent the addition of duplicate prizes, implement a check to verify whether the `packed` prize already exists in the prize pool before adding it.

[L-06] Risks in modifying critical parameters in marketplace

The `Marketplace` contract allows for the modification of several critical parameters, including `acceptedCurrency`, `tradeFee`, `fees`, and `whitelistRegistry`.

```
    function setFees(uint256 newFees) external {
        require(msg.sender == marketplaceAdmin, "Only marketplaceAdmin can call");
        if (newFees > 1000) revert InvalidFees();
        fees = newFees;
    }
    /**
     * @notice Updates the trade fee (called by MarketplaceAdmin)
     * @dev Only callable by the MarketplaceAdmin contract
     * @param newTradeFee New trade fee amount
     */
    function setTradeFee(uint256 newTradeFee) external {
        require(msg.sender == marketplaceAdmin, "Only marketplaceAdmin can call");
        tradeFee = newTradeFee;
    }

    /**
     * @notice Updates the fee receiver address (called by MarketplaceAdmin)
     * @dev Only callable by the MarketplaceAdmin contract
     * @param newFeeReceiver Address of the new fee receiver
     */
    function setFeeReceiver(address newFeeReceiver) external {
        require(msg.sender == marketplaceAdmin, "Only marketplaceAdmin can call");
        if (newFeeReceiver == address(0)) revert InvalidFeeReceiver();
        feeReceiver = newFeeReceiver;
    }

    /**
     * @notice Updates the accepted currency (called by MarketplaceAdmin)
     * @dev Only callable by the MarketplaceAdmin contract
     * @param newERC20TokenAddress Address of the new ERC20 token to be accepted
     */
    function setAcceptedCurrency(address newERC20TokenAddress) external {
        require(msg.sender == marketplaceAdmin, "Only marketplaceAdmin can call");
        if (newERC20TokenAddress == address(0)) revert InvalidCurrencyAddress();
        acceptedCurrency = newERC20TokenAddress;
    }

    /**
     * @notice Updates the whitelist registry (called by MarketplaceAdmin)
     * @dev Only callable by the MarketplaceAdmin contract
     * @param newWhitelistRegistryAddress Address of the new whitelist registry contract
     */
    function setWhitelistRegistry(address newWhitelistRegistryAddress) external {
        require(msg.sender == marketplaceAdmin, "Only marketplaceAdmin can call");
        if (newWhitelistRegistryAddress == address(0)) revert InvalidRegistryAddress();
        whitelistRegistry = newWhitelistRegistryAddress;
    }
```

Changing these parameters mid-operation can lead to unintended consequences. Modifying the tradeFee or fees could affect ongoing trades, leading to disputes or financial losses for users.

[L-07] DoS risk with zero-amount transfers

The `Marketplace` contract currently performs transfers using the `transferFrom` method for both the `sellerAmount` and the `feeAmount` without checking if these amounts are greater than zero.

```
            // Update state before external calls
            totalPendingFees += feeAmount;

            // External calls after state changes
            IERC20(offers[i].request.offerCurrency).transferFrom(offers[i].request.requester,
offers[i].receiver, sellerAmount);
            IERC20(offers[i].request.offerCurrency).transferFrom(offers[i].request.requester,
address(this), feeAmount);
```

This can lead to Denial of Service (DoS) vulnerabilities, particularly with ERC20 tokens that do not support zero-amount transfers.

To mitigate the risk of DoS attacks due to `zero-amount transfers`, implement a check to ensure that the amounts being transferred are greater than zero before executing the transfer.

[L-08] Compatibility issues with `transferFrom()` in ERC20

The `Marketplace` contract uses the `transferFrom` method to handle payments in the `acceptedCurrency`:

```
            address payee = chargeDelegate ? address(msg.sender) : listings[i].receiver;

            // External calls after state changes
            IERC20(acceptedCurrency).transferFrom(payee, listings[i].request.owner,
sellerAmount);
            IERC20(acceptedCurrency).transferFrom(payee, address(this), feeAmount);
```

While the project currently states that only `USDC` is used, the implementation does not account for potential compatibility issues with other ERC20 tokens, such as `USDT`, which may have different transfer mechanisms.

Furthermore, the function allows for the modification of `acceptedCurrency`, which could lead to unexpected behavior if a non-compliant token is set as the accepted currency in the future.

```
    function setAcceptedCurrency(address newERC20TokenAddress) external {
        require(msg.sender == marketplaceAdmin, "Only marketplaceAdmin can call");
        if (newERC20TokenAddress == address(0)) revert InvalidCurrencyAddress();
        acceptedCurrency = newERC20TokenAddress;
    }
```

Instead of using `transferFrom`, use `safeTransferFrom` for better handling of the case.

[L-09] Inconsistent validation in `isValidListing()` and `isValidTrade()`

The `isValidListing` function in the Marketplace contract performs **comprehensive checks to ensure that the NFT contract is whitelisted and that the requester is the actual owner of the NFT.** This proactive validation helps identify issues early and prevents errors to save gas costs.

```
        if (!
IWhitelistRegistry(whitelistRegistry).isWhitelisted(listing.request.tokenContractAddress)) {
            revert NFTNotWhitelisted();
        }
        if (IERC721(listing.request.tokenContractAddress).ownerOf(listing.request.tokenId) !=
listing.request.owner) {
            revert NotOwner();
        }
```

In contrast, the `isValidTrade` function only checks if the NFTs involved in the trade are whitelisted, neglecting to verify the ownership of the NFTs being traded. It is recommended to do the check earlier in order to save gas if the transaction is to be reverted.

```
        // check if all the give NFTs and take NFTs are whitelisted
        for (uint256 i = 0; i < trade.request.give.length; i++) {
            if (!
IWhitelistRegistry(whitelistRegistry).isWhitelisted(trade.request.give[i].tokenContractAddress))
{
                revert NFTNotWhitelisted();
            }
        }

        for (uint256 i = 0; i < trade.request.take.length; i++) {
            if (!
IWhitelistRegistry(whitelistRegistry).isWhitelisted(trade.request.take[i].tokenContractAddress))
{
                revert NFTNotWhitelisted();
            }
        }
```

Note: the same issue exists in `isValidBid`.

[L-10] Data inconsistency risk from modifying critical addresses in `cardAllocationPool`

The `CardAllocationPool` contract contains critical parameters such as `packetNFTAddress`, `cardNFTAddress`, and `s_vrfCoordinator`.

```
    function updateCardAddress(address newCardAddress) external onlyRole(redeemManagerRoleId) {
        if (newCardAddress == address(0)) revert InvalidCardContract();
        address oldAddress = cardNFTAddress;
        cardNFTAddress = newCardAddress;
        emit CardAddressUpdated(oldAddress, newCardAddress);
    }

    function updatePacketAddress(address newPacketAddress) external
onlyRole(redeemManagerRoleId) {
        if (newPacketAddress == address(0)) revert InvalidPacketContract();
        address oldAddress = packetNFTAddress;
        packetNFTAddress = newPacketAddress;
        emit PacketAddressUpdated(oldAddress, newPacketAddress);
    }

    function setCoordinator(address _vrfCoordinator) external
```

```
onlyRoleOrCoordinator(redeemManagerRoleId) {
        if (_vrfCoordinator == address(0)) {
            revert ZeroAddress();
        }
        s_vrfCoordinator = IVRFCoordinatorV2Plus(_vrfCoordinator);

        emit CoordinatorSet(_vrfCoordinator);
    }
```

Modifying these addresses after they have been set can lead to significant issues, including data inconsistency.

For instance, if the addresses are changed while there are ongoing operations, such as packet open requests that rely on the original addresses, the `fulfillRandomWords` may fail. This could result in unexpected behavior, errors, or even the locking of NFTs, as the contract may attempt to interact with the wrong addresses, leading to a breakdown in functionality.

[L-11] Deterministic selection with one available card

In the `CardAllocationPool` contract, the logic for selecting a card from `availableCardBundles` only checks that the length of the array is greater than zero.

```
    function selectRandomCards(CardBundle[] storage availableCardBundles, uint256 randomSeed)
        private
        returns (uint256[] memory selectedCardBundle)
    {
        if (availableCardBundles.length == 0) {
            revert NoAvailableCardBundles();
        }
        uint256 index = uint256(keccak256(abi.encode(randomSeed, 1))) %
availableCardBundles.length;
        selectedCardBundle = availableCardBundles[index].cardIds;

        // Swap selected card bundle with last unselected card bundle and pop
        availableCardBundles[index] = availableCardBundles[availableCardBundles.length - 1];
        availableCardBundles.pop();

        return selectedCardBundle;
    }
```

However, if the length is exactly one, the selection becomes deterministic, as there is only one card available to choose from. In this scenario, the randomness introduced by the random seed is irrelevant, and the outcome is predetermined.

This undermines the purpose of using randomness in the selection process, as it does not provide any variability or uncertainty when only one option exists.

Note: This finding also applies for `SpinLottery`

```
    function _claimPrize(uint8 rarity, uint256 randomValue) internal returns (address
nftAddress, uint88 tokenId) {
        PrizePool storage pool = prizePools[rarity];
        if (pool.nextPrizeId <= pool.firstPrizeId) revert NoPrizesAvailable();
        // Calculate prize count and random index
        uint256 prizeCount = pool.nextPrizeId - pool.firstPrizeId;
        uint256 index = randomValue % prizeCount;
        uint256 prizeId = pool.firstPrizeId + index;
        ...
    }
```

To address this issue, it is important to acknowledge that when the length of `availableCardBundles` is one, the selection should be treated as deterministic. Maybe some additional checks could be added.

[L-12] Validation issues in `authorizedOpenPacket()`

In the `CardAllocationPool` contract, the `authorizedOpenPacket` function allows for the opening of packets using specified cards.

```
    /**
     * @notice Allows authorized managers to open packets with specific cards
     * @dev Used for physical packet redemption or administrative purposes
     * @param packetId The ID of the packet to open
     * @param packetType The type of packet being opened
     * @param cards Array of specific card IDs to be included in the packet
     * @param openMetadata metadata for card opening
     * @custom:access Redeem Manager Role
     * @custom:example
     * ```typescript
     * const openPacket = await walletClient.writeContract({
     *   address: cardAllocationPoolAddress,
     *   abi: cardAllocationPoolABI,
     *   functionName: 'authorizedOpenPacket',
     *   args: [
     *     packetId,
     *     packetType,
     *     [1, 2, 3], // Array of card IDs
     *     "https://example.com/metadata" // metadata for card opening
     *   ]
     * })
     *
     * // Wait for transaction
     * const receipt = await publicClient.waitForTransactionReceipt({
     *   hash: openPacket
     * })
     * ```
     */
    function authorizedOpenPacket(
        uint256 packetId,
        uint256 packetType,
        uint256[] memory cards,
        string memory openMetadata
    ) external onlyRole(redeemManagerRoleId) {
        address packetOwner = RipFunPacks(packetNFTAddress).ownerOf(packetId);
        // Transfer cards to owner
        for (uint256 i = 0; i < cards.length; i++) {
            IERC721(cardNFTAddress).transferFrom(msg.sender, packetOwner, cards[i]);
```

```
        }

        // Finalize packet opening
        RipFunPacks(packetNFTAddress).finalizeOpen(packetId, openMetadata);

        emit PhysicalPacketOpened(packetId, packetType, packetOwner, cards);
    }
```

However, there are critical validation issues that need to be addressed: 1. First, the cards used in `authorizedOpenPacket` should not include any cards that have already been added through `addCardBundlesToPacketPool`. If they do, it could lead to failures for other users attempting to execute `instantOpenPacket` operations, as the state of the card bundles would be altered unexpectedly without any notice. 2. Secondly, there should be a validation check to ensure that the `burnType` associated with the corresponding `packetId` is specifically `BurnType.OPEN_PACKET` but not `BurnType.INSTANT_OPEN_PACKET`.

To enhance the integrity of the `authorizedOpenPacket` function, it is recommended to implement corresponding checks.

[L-13] Inconsistent role ID in `cardAllocationPool` initialization

In the `CardAllocationPool` contract, the assignment of the `redeemManagerRoleId` is done during the initialization phase, requiring the role ID to be specified explicitly.

```
       packetNFTAddress = packetAddress;
        cardNFTAddress = cardAddress;
        redeemManagerRoleId = _redeemManagerRoleId;
        __RoleBasedAccessControlConsumer_init(_rbac);
        __UUPSUpgradeable_init();
```

This approach differs from other contracts, such as the `Card` contract, where role IDs like `REDEEM_MANAGER_ROLE` are directly defined as constants.

```
    /// @notice Role identifier for accounts that can manage card redemptions
    /// @dev Used in access control checks for redemption operations
    uint256 public constant REDEEM_MANAGER_ROLE = 4;
```

This inconsistency introduces redundancy and increases the risk of errors. If the `redeemManagerRoleId` overlaps with other role IDs, such as `PRICE_MANAGER_ROLE`, it could inadvertently grant excessive privileges to unrelated roles.

Note: the same applies for `marketplaceManagerRoleId` used in `Marketplace` and `lotteryManagerRoleId` used in `SpinLottery`.

To enhance consistency and reduce the risk of errors, consider defining all role IDs as constants within the contract.

[L-14] Risks of modifying `randomAllocationPool` during a transaction

The `setCardAllocationPool` function in the Packet contract allows the `SUPER_ADMIN_ROLE` to change the `randomAllocationPool` address at any time.

```
    function setCardAllocationPool(address newCardAllocationPool) external
onlyRole(SUPER_ADMIN_ROLE) {
        randomAllocationPool = CardAllocationPool(newCardAllocationPool);
    }
```

This mid-transaction modification can lead to significant issues: - First, users may see outdated pool information on the front end that does not match the actual state during transaction execution, potentially leading to disputes and confusion. - Second, if the previous pool address's `ALLOCATION_MANAGER_ROLE` has also been revoked, any subsequent calls to `finalizeOpen` by the previous pool could fail. This failure would result in the packet being temporarily locked, requiring an administrator to manually call `revertBurnRequest` to resolve the situation.

To mitigate these risks, consider implementing a mechanism that prevents changes to the `randomAllocationPool` during active transactions or requires a confirmation period before the change takes effect.

[L-15] Shared privilege concerns in role-based access control cards

In the Card contract, the MINT_MANAGER_ROLE and REDEEM_MANAGER_ROLE are defined as privileged roles for managing card minting and redemption operations, respectively.

However, multiple `Cards` could refer to the same `rbac` (Role-Based Access Control) instance via `RoleBasedAccessControlConsumer`. This design implies that if a user is granted minting or redemption privileges in one `Card`, they inherently possess the same privileges in all other `Card` instances that share the same `rbac`.

This raises concerns about the independence of access control across different `Card` contracts, as it could lead to unintended privilege escalation or management issues if the roles are not intended to be universally applicable.

[L-16] Data integrity issue with `revertBurnRequest` in `RipFunPacks`

In the `RipFunPacks` function, if a packet of type `INSTANT_OPEN` undergoes a `revertBurnRequest`, it can lead to a failure when the `fulfillRandomWords` function is later called by the `Chainlink`.

```
    function revertBurnRequest(uint256 packetId, string calldata revertBurnRequestMetadata)
        external
        onlyRole(REDEEM_MANAGER_ROLE)
    {
        if (_packetBurnType[packetId] == BurnType.NONE) {
            revert PacketNotInBurnState();
        }

        BurnType previousBurnType = _packetBurnType[packetId];
        _packetBurnType[packetId] = BurnType.NONE;

        emit BurnRequestReverted(packetId, revertBurnRequestMetadata, previousBurnType,
msg.sender);
    }
```

However, in `CardAllocationPool` will call `finalizeOpen` in the function `fulfillRandomWords`.

```
    function fulfillRandomWords(uint256 requestId, uint256[] memory randomWords) internal {
        PacketOpenRequest storage request = requestIdToPacketOpen[requestId];
        if (request.fulfilled) revert("Already fulfilled");

        // fetch the available card bundles
        CardBundle[] storage cardBundles = packetTypeToCardBundles[request.packetType];

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

        // Transfer cards to owner
        for (uint256 i = 0; i < selectedCards.length; i++) {
            IERC721(cardNFTAddress).transferFrom(address(this), request.owner,
selectedCards[i]);
        }

        // Mark request as fulfilled
        request.fulfilled = true;

        // Call finalize open on packet contract
        RipFunPacks(packetNFTAddress).finalizeOpen(request.packetId, "");

        emit PacketOpenFulfilled(requestId, request.packetId, selectedCards);
    }
```

This failure occurs because the state of the packet is not properly managed when the burn request is reverted.

```
    function finalizeOpen(uint256 packetId, string memory openMetadata) external
onlyRole(ALLOCATION_MANAGER_ROLE) {
        BurnType burnType = _packetBurnType[packetId];
        if (burnType != BurnType.OPEN_PACKET && burnType != BurnType.INSTANT_OPEN_PACKET) {
            revert InvalidBurnStateForOpen();
        } // <= revert here, due to status has been reset

        _burn(packetId);
        emit OpenFinalized(packetId, burnType, msg.sender, openMetadata);
    }
```

As a result, the `CardAllocationPool` may end up with inconsistent or "dirty" data, where the status of the packet does not accurately reflect its intended state. - The request will never be fulfilled. - The VRF subscription is still charged for the work done to generate your requested random values.

To mitigate this issue, it is crucial to implement proper state management to keep track of the status.

[L-17] Unconventional role transfer timing in `RoleBasedAccessControl`

The `RoleBasedAccessControl` contract employs a mechanism similar to the `Ownable2Step` pattern for role transfers, utilizing a `roleTransferExpiry` timestamp to enforce a time constraint on the transfer process.

```
        if (block.timestamp > roleTransferExpiry) {
            revert RoleTransferExpired();
        }
```

However, this approach is unconventional, as it requires the transfer to be completed within a specific time window rather than **allowing for a timelock period.**

Typically, a `timelock` buffer is implemented to provide a buffer for community review and to enhance decentralization, allowing stakeholders to react to potential changes. The current implementation may inadvertently limit the ability of the community to assess and respond to role transfers, which could undermine the intended governance model.

To align with best practices for decentralized governance, **consider implementing a `timelock` mechanism that allows for a delay before the role transfer becomes effective.**

[L-18] No storage gap in `RoleBasedAccessControl` contract

The `RoleBasedAccessControl` contract, which inherits from `Initializable`, `ERC1155Upgradeable`, and `UUPSUpgradeable`, defines state variables such as `pendingSuperAdmin` and `roleTransferExpiry`.

```
contract RoleBasedAccessControl is Initializable, ERC1155Upgradeable, UUPSUpgradeable {
    /// @dev ID for the super admin role that has full system access
    uint256 public constant SUPER_ADMIN_ROLE = 1;
    /// @dev ID for the role manager role that can assign/revoke other roles
    uint256 public constant ROLE_MANAGER_ROLE = 2;
    /// @dev Time window during which a super admin role transfer must be accepted
    uint256 public constant ROLE_TRANSFER_EXPIRY_DEADLINE = 2 days;

    /// @dev Address that is proposed to become the new super admin
    address public pendingSuperAdmin;
    /// @dev Timestamp when the current super admin role transfer expires
    uint256 public roleTransferExpiry;
```

However, it does not implement a storage gap or namespace layout as recommended by best practices for upgradeable contracts.

While this omission does not cause immediate issues, since no contracts currently inherit from `RoleBasedAccessControl`, it is still strongly recommended to include a storage gap. Adding a storage gap proactively helps prevent potential storage collisions in future development, especially if the contract is later inherited.

[L-19] Unnecessary modifier for `getInventoryPackets`

`getInventoryPackets` is a external view function, but it unnecessary restricted to only callable by `INVENTORY_MANAGER_ROLE`.

```
    function getInventoryPackets(uint256 packetTypeId) external view
onlyRole(INVENTORY_MANAGER_ROLE)
        returns (uint256[] memory) {
        // Validate packet type ID
        if (!_validatePacketTypeId(packetTypeId)) {
            revert InvalidPacketType(packetTypeId);
        }
        return _packetInventory[packetTypeId];
    }
```

Consider removing the modifier.

[L-20] Low `requestConfirmations` number prone to frequent chain reorgs

In both `CardAllocationPool` and `SpinLottery`, the initial `requestConfirmations` value is set to 3, which is insufficient for the selected chains (Base and Polygon). Miners or validators on these chains could potentially reorganize the chain's history to move a randomness request from the contracts into a different block, resulting in a different VRF output. It is recommended to set `requestConfirmations` to 5 or higher to enhance security.

[L-21] `CardAllocationPool` should use `ERC1155HolderUpgradeable`

`CardAllocationPool` is an upgradable contract, but currently uses `ERC1155Holder` instead of `ERC1155HolderUpgradeable`. While this does not impact functionality, it is considered best practice to use the upgradeable contract version.

[L-22] Missing cancel burn functionality in `Card` and `Packet`

When a `Card` and `Packet` holder calls `initiateBurn`, they must wait for the `REDEEM_MANAGER_ROLE` to execute `finalizeRedeem`. However, there is no functionality that allows holders to cancel the redemption after a certain period if the

`REDEEM_MANAGER_ROLE` does not process the request. Considering that during this state the `Card` and `Packet` cannot be transferred, it is important to allow users to cancel the request if the managers fail to execute `finalizeRedeem` within a specified time threshold.

[L-23] There is no slippage control in `spin`

The `spin` function does not allow a user to specify the maximum amount of usdc they wish to spend when calling the function. Since the cost of a spin can change from block to block due to state changes, it is important to let users set a max amount they are willing to pay for calling `spin` otherwise, they may be overcharged.

```
        // Calculate spin cost
        uint256 spinCost = calculateSpinCost(_totalSlots, _prizeCount);

        // Process payment
        if (!usdcToken.transferFrom(msg.sender, address(this), spinCost)) {
            revert USDCTransferFailed();
        }
```

Recommendations:

Allow users to specify a maximum amount they are willing to spend when calling the `spin` function.

[L-24] There is no slippage protection when purchasing packets

In `PacketStore.sol` the contract allows users to buy packets at a certain price.

```
    function _processPurchase(uint256 packetTypeId, address recipient) internal returns
(uint256) {
        uint256 price = packetTypePrices[packetTypeId];
        if (price == 0) revert InvalidPacketType(packetTypeId);
```

This price is set by the price manager, however the problem occurs when a user is buying a packet while the manager is changing the price for the packet, this can happen by accident or due to a malicious manager who is frontrunning the users tx. Since the user cannot specify a price, he will be forced to pay the price even if it changes when his tx was in the mempool. This happens because purchasing packets does not use any slippage protections.

```
    function purchasePacket(uint256 packetTypeId) external nonReentrant returns (uint256) {
        if (salesFrozen) revert SalesCurrentlyFrozen();

        // Validate packet type ID
        if (!_validatePacketTypeId(packetTypeId)) {
            revert InvalidPacketType(packetTypeId);
        }

        return _processPurchase(packetTypeId, msg.sender);
    }
```

Recommendations:

Allow users to specify the price they are paying and revert if the price is higher than what the user has specified.

[L-25] Insufficient error handling

In the `SpinLottery` contract, if there are insufficient prizes available, the `_claimPrize` function will revert with the error `NoPrizesAvailable`.

```
        PrizePool storage pool = prizePools[rarity];
        if (pool.nextPrizeId <= pool.firstPrizeId) revert NoPrizesAvailable();
```

This error can be caught in the try-catch block during the prize distribution process, but it leads to a situation where the user is informed that they did not win.

```
            // 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);
            }
```

This handling is illogical, as the user has fulfilled their obligation by paying, and any failure in the prize distribution should not be attributed to them. This could result in user dissatisfaction and a perception that the protocol is unfair, as they are effectively penalized for a failure that is not their fault.

Recommendations:

To improve the error handling, introduce a separate event for cases where there are no prizes available, allowing the contract to **differentiate between a successful spin with no win and a failure due to insufficient prizes.**

[L-26] Inconsistent prize locking/unlocking from dynamic weight changes

In the `SpinLottery` contract, the process of locking and unlocking prizes relies on the `calculatePendingPrizesForRarity` function, which uses the weights defined in `rarityConfigs[rarity]` and the `totalRarityWeight`.

```
        // Calculate based on weight distribution
        uint256 weight = rarityConfigs[rarity].weight;
        uint256 pendingCount = (_prizeCount * weight) / totalRarityWeight;
```

```
        for (uint8 i = 1; i <= maxRarityId; i++) {
            if (rarityConfigs[i].active) {
                pendingToDecrease[i] = calculatePendingPrizesForRarity(req.prizeCount, i);

                // Decrease pending count, ensure we don't underflow
                if (prizePools[i].pendingCount >= pendingToDecrease[i]) {
```

```
                prizePools[i].pendingCount -= uint96(pendingToDecrease[i]);
            } else {
                prizePools[i].pendingCount = 0;
            }
        }
    }
```

However, these weights can be modified through the `configureRarity` function, **leading to potential inconsistencies between the number of prizes locked and the number of prizes unlocked.**

```
    if (oldConfig.active) {
        totalRarityWeight -= oldConfig.weight;
    }

    // Only add new weight if rarity will be active
    if (_active) {
        totalRarityWeight += _weight;
    }

    // Update configuration
    rarityConfigs[_rarityId] = RarityConfig({
        weight: _weight,
        basePrice: _basePrice,
        active: _active
    });
```

It could result in errors during the unlocking phase or cause the entire transaction to revert (especially when a new `rarity` is being added).

Recommendations:

It is recommended to record the numbers being locked for each rarity, or not allow weight changes during when there is a spin going on.

[L-27] Redundant state management and inconsistency risk in `MarketplaceAdmin`

The `MarketplaceAdmin` contract manages the `marketplaceAddress` and maintains a separate set of parameters, including variables like `acceptedCurrency`, etc.

```
contract MarketplaceAdmin is UUPSUpgradeable, RoleBasedAccessControlConsumer,
MarketplaceStorage { // <= MarketplaceStorage here
    /// @notice The address of the Marketplace contract
    address public marketplaceAddress;
    ...
}
```

This duplication of state management is not a best practice, as it increases complexity and the potential for inconsistencies between the two contracts.

Additionally, during the setter functions like `updateAcceptedCurrency` function, the local state is modified before attempting to update the corresponding parameter in the `Marketplace` contract.

```
        address oldCurrency = acceptedCurrency;
        acceptedCurrency = newERC20TokenAddress;
        emit AcceptedCurrencyUpdated(oldCurrency, newERC20TokenAddress);

        // Also update the currency in the Marketplace contract if it exists
        if (marketplaceAddress != address(0)) {
            // This will revert if the Marketplace contract doesn't exist or doesn't have this
function
            try Marketplace(marketplaceAddress).setAcceptedCurrency(newERC20TokenAddress) {
                // Successfully called the function
            } catch {
                // Silently ignore errors
            }
        }
```

The use of a `try-catch` block to handle this update means that **if the call fails silently, the local state and the state in the `Marketplace` contract could become inconsistent**, leading to confusion and potential errors in future transactions.

Recommendations:

Use a Single Source of Truth to eliminate the redundancy.

[L-28] Potential revenue loss due to fee calculation method

The current fee calculation in the `Marketplace` contract uses a straightforward truncation method when determining the `fee` amount from a listing price.

```
            uint256 feeAmount = (listings[i].request.price * fees) / 10000;
            uint256 sellerAmount = listings[i].request.price - feeAmount;
            // Update state before external calls
            totalPendingFees += feeAmount;

            address payee = chargeDelegate ? address(msg.sender) : listings[i].receiver;
```

This can lead to rounding down and result in a loss of revenue for the protocol. For instance, this truncation can lead to a fee loss of up to `0.01 USDC` per `NFT purchase`. Over time, especially with a high volume of transactions, this could accumulate to a significant loss in potential revenue for the project.

Recommendations:

Optimize the fee calculation and ensure it is more favorable to the protocol.

[L-29] Indexing issues in `removeCardBundlesFromPacketPool()`

The `removeCardBundlesFromPacketPool` function in the `CardAllocationPool` contract allows for the removal of card bundles by passing their indices.

```
    function removeCardBundlesFromPacketPool(uint256 packetType, uint256[] memory
cardBundleIndexes)
        external
        onlyRole(redeemManagerRoleId)
    {
        CardBundle[] storage cardBundles = packetTypeToCardBundles[packetType];

        for (uint256 i = 0; i < cardBundleIndexes.length; i++) {
            uint256 index = cardBundleIndexes[i];

            // transfer the cards to the caller
            for (uint256 j = 0; j < cardBundles[index].cardIds.length; j++) {
                IERC721(cardNFTAddress).transferFrom(address(this), msg.sender,
cardBundles[index].cardIds[j]);
            }

            // Swap selected card bundle with last unselected card bundle and pop
            cardBundles[index] = cardBundles[cardBundles.length - 1];
            cardBundles.pop();
        }

        emit CardBundlesRemovedFromPool(packetType, cardBundleIndexes);
    }
```

However, this approach is flawed due to the **dynamic nature of the indices.** When bundles are removed during the `fulfillRandomWords` process, the selected bundle is swapped with the last bundle in the array and then removed.

```
        uint256 index = uint256(keccak256(abi.encode(randomSeed, 1))) %
availableCardBundles.length;
        selectedCardBundle = availableCardBundles[index].cardIds;

        // Swap selected card bundle with last unselected card bundle and pop
        availableCardBundles[index] = availableCardBundles[availableCardBundles.length - 1];
        availableCardBundles.pop();
```

This can lead to several issues: - If the admin initially attempts to remove the last bundle, but the last bundle has been swapped to a different position, potentially causing a denial of service (DoS) due to Out-of-bounds. - Conversely, if a bundle that was intended for removal has already been selected in the `instant Open` process and is removed in a previous operation(`fulfillRandomWords`), the function may end up removing the last bundle instead, leading to incorrect state management and unintended consequences.

Recommendations:

To address these issues, consider implementing a more robust mechanism for managing the removal of card bundles.

[L-30] Use of `transferFrom` over `safeTransferFrom()` in NFT transfers

Reference Finding: https://solodit.cyfrin.io/issues/fuel1-5-use-safetransferfrom-instead-of-transferfrom-for-erc721-transfers-hexens-none-fuel-markdown

The `PacketStore` contract utilizes the transferFrom method to transfer NFTs, which is not the recommended approach for `ERC721` tokens. The `transferFrom` method does not perform any checks to ensure that the recipient is capable of receiving the NFT.

```
        // Transfer packet to buyer
        packetContract.transferFrom(address(this), recipient, packetId);

        emit PacketSold(packetId, recipient, price, packetTypeId);
```

If the recipient is a contract that does not implement the required `ERC721` interface, the NFT could be permanently locked, as it would not be able to be transferred out of that contract.

Note: This is the same in other contracts, including `CardAllocationPool` and `SpinLottery`.

Recommendations:

To enhance the safety of NFT transfers, it is advisable to use the `safeTransferFrom` method instead of `transferFrom`.

[L-31] Use of `_mint` instead of `_safeMint()` in minting

The `_mintNextPacket` function in the `RipFunPacks` contract utilizes the `_mint` method to create new tokens.

```
    function _mintNextPacket(PacketMintParams calldata params, address receiver) internal {
        if (!_packetTypes[params.packetTypeId].isRegistered) {
            revert PacketTypeNotRegistered();
        }

        uint256 tokenId = _nextTokenId++;
        _packetUris[tokenId] = params.packetMetadata;
        _packetTypeIds[tokenId] = params.packetTypeId;
        _mint(receiver, tokenId);

        emit PacketMinted(tokenId, receiver, params.packetTypeId, params.packetMetadata,
params.packetSerialNumber);
    }
```

According to OpenZeppelin's documentation, the use of `_mint` is discouraged in favor of `_safeMint`. The `_safeMint` method includes additional checks to ensure that the `recipient` address is capable of receiving ERC721 tokens, which helps prevent scenarios where tokens could be locked if sent to a contract that does not implement the required interface.

Note: This is the same for `Card` contract, which also uses `_mint` in the function `_mintNextCard`.

Recommendations:

To align with best practices and enhance the safety of token minting, replace the `_mint` call with `_safeMint` in the `_mintNextPacket` function.

[L-32] Potential incomplete role revocation in `revokeRole()` function

The `revokeRole` function is intended to revoke a specific role from a given address. Thus, it is expected to work completely.

```
    /**
     * @notice Revokes a role from an address
     * @dev Only super admin or role manager can revoke roles
     * @param roleId ID of the role to revoke
     * @param user Address to revoke the role from
     *
     * Requirements:
     * - Caller must be super admin or role manager
     * - Role ID cannot be 0 or SUPER_ADMIN_ROLE
     *
     * Emits a {RoleBurned} event
```

However, its implementation does not guarantee that the role is fully revoked.

Since the contract uses `ERC1155` tokens, a user may hold multiple tokens corresponding to the same role(or `tokenId`) (e.g., multiple `ROLE_MANAGER_ROLE` tokens) since there is no such check in `assignRole`.

```
    function assignRole(uint256 roleId, address receiver) external onlySuperAdminOrRoleManager {
        if (roleId == SUPER_ADMIN_ROLE) {
            revert SuperAdminRoleReserved();
        }
        if (roleId == 0) {
            revert InvalidRoleId();
        }

        _mint(receiver, roleId, 1, ""); // <= No check if the user does already have such role
before
        emit RoleMinted(roleId, receiver, msg.sender);
    }
```

The current implementation only burns 1 token when revoking a role, which means that the user could still retain access if they possess an additional `tokenId` representing the same role. This behavior contradicts the intended design of the `revokeRole` function which is intended to revoke a specific role from a given address.

```
function revokeRole(uint256 roleId, address user) external onlySuperAdminOrRoleManager {
    if (roleId == SUPER_ADMIN_ROLE) {
        revert SuperAdminRoleReserved();
    }
    if (roleId == 0) {
        revert InvalidRoleId();
    }

    _burn(user, roleId, 1); // <= only burns 1 token
    emit RoleBurned(roleId, user, msg.sender);
}
```

Note that it's possible for the user to have multiple instances of the same role, see here: (`balanceOf(user, roleId) > 0`)

```
function hasRole(uint256 roleId, address user) public view returns (bool) {
    return balanceOf(user, roleId) > 0;
}
```

Recommendations:

To ensure that the `revokeRole function` effectively revokes all instances of a role from a user, it should first check the total balance of the specified `roleId` tokens held by the user. If the user holds multiple tokens, the function should burn all of them.

[L-33] Trade fee calculation mismatch in documentation and implementation

The `tradeFee` variable is documented in `MarketplaceStorage.sol` as a "percentage in BIPS (1% = 100)" but is implemented in `executeTrade()` as a fixed amount in token units. The contract initializes `tradeFee` to `10e6` with a comment "1 USDC," confirming it's intended as an absolute amount. This inconsistency creates confusion about how fees are calculated and could lead to admin errors when setting fee values.

Recommendation: Update the natspac in `MarketplaceStorage.sol` to accurately reflect that `tradeFee` is a fixed token amount rather than a percentage. Change the comment to:

```
/// @notice Fixed trade fee amount in token units (e.g., 10e6 = 1 USDC)
uint256 public tradeFee;
```

Alternatively, if percentage-based fees are desired, modify the `executeTrade()` implementation to calculate fees as a percentage of trade value.