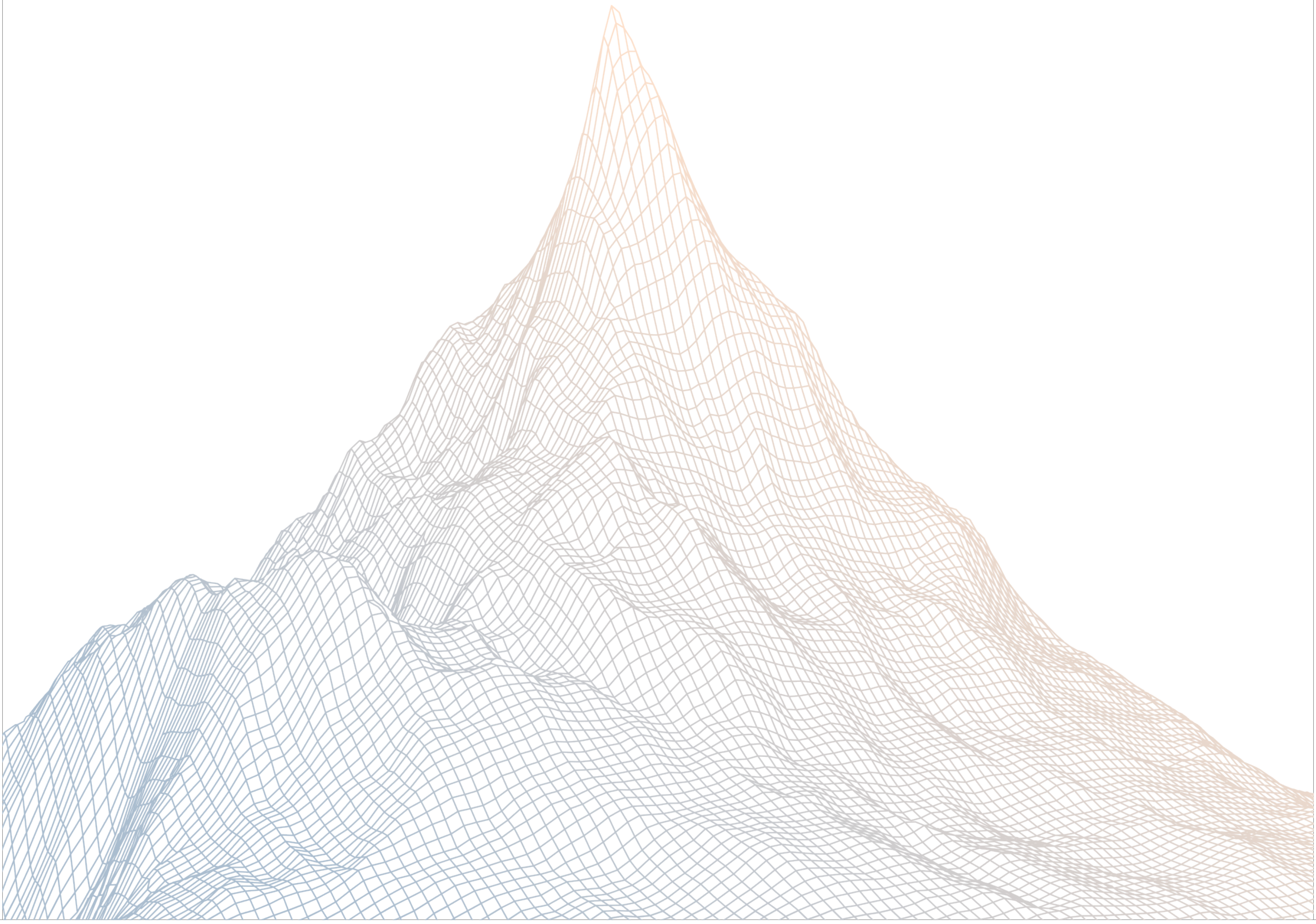
Zenith

RareBetSports

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: September 4th to September 8th, 2025

AUDITED BY: adriro
said

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2 Executive Summary

2.1 About RareBetSports

RareBetSports is a web3 ecosystem designed to capture the thrill of sports and crypto. Built for the next generation of fans and communities of the future

2.2 Scope

The engagement involved a review of the following targets:

Target	rbs-contracts
Repository	https://github.com/RareBetSports/rbs-contracts
Commit Hash	`e6516111db8411f3044fc13bff9b92659b0e548e`
Files	`contracts/**.sol`

Target	RareBetSports Mitigation Review
Repository	https://github.com/RareBetSports/rbs-contracts
Commit Hash	`3cdfd5835e4d415780c0c396f184a77a8c64c6c1`
Files	`Diff up to 3cdfd5835e4d415780c0c396f184a77a8c64c6c1`

2.3 Audit Timeline

September 4th, 2025	Audit start
September 8th, 2025	Audit end
September 24th, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	3
Low Risk	9
Informational	6
Total Issues	**18**

3 Findings Summary

ID	Description	Status
M-1	Slips can be refunded more than once	Resolved
M-2	All unlisted bets marked Lost instead of Cancelled	Resolved
M-3	gradeBet before openUntil exposes outcomes on-chain	Resolved
L-1	Improve argument validations on SlipRepository	Resolved
L-2	Check if the bet has been previously invalidated	Resolved
L-3	Improve validations in grade bet	Resolved
L-4	Player ID duplicate check fails for zero values	Resolved
L-5	Unsafe ERC20 transfers used in BatchSender and SlipRepository	Resolved
L-6	Use of transfer in native payouts/refunds enables DoS	Resolved
L-7	Retroactive changes to multipliers, minNumBetsPerSlip and maxNumBetsPerSlip affect slip rewards	Resolved
L-8	invalidateBet can overwrite graded bets	Resolved
L-9	gradeSlips may cause user reward loss	Acknowledged
I-1	Multiplier field in Bet structure is not used	Acknowledged
I-2	Owner is not needed in BatchSender	Acknowledged
I-3	Consider using a role-based authentication	Acknowledged
I-4	Improve multipliers validation	Resolved
I-5	Zero hashId creates inconsistent bet state	Resolved
I-6	Excess native in batchSendEther not refunded	Acknowledged

4 Findings

4.1 Medium Risk

A total of 3 medium risk findings were identified.

[M-1] Slips can be refunded more than once

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- SlipRepository.sol

Description:

Although the action is controlled by the admin, the implementation of `refundSlips()` allows cancelled slips to be refunded multiple times, leading to potential accidental loss of assets. The function simply toggles `isCancelRefund` to true without validating whether the slip was previously refunded.

```
        for (uint256 i = 0; i < slipIds.length; i++) {
            uint256 currentSlipId = slipIds[i];
            Slip storage slip = slips[currentSlipId];

            if (slip.status == SlipStatus.Cancelled &&
    !slip.isCancelRefund) {
                slip.isPaid = true;
                slip.isCancelRefund = true;
            }
        }

```

Slips that have been refunded in previous calls or duplicates in the array would be paid again.

Recommendations:

The implementation can validate that all slips are indeed cancelled and not refunded:

```
for (uint256 i = 0; i < slipIds.length; i++) {
    uint256 currentSlipId = slipIds[i];
    Slip storage slip = slips[currentSlipId];

    require(slip.status == SlipStatus.Cancelled && !slip.isCancelRefund);

    slip.isPaid = true;
    slip.isCancelRefund = true;
}
```

The second loop can then avoid re-checking the status or the flag, as this validation has already been enforced in the first loop.

Additionally, note that the grading process should have already set the `isPaid` flag to true when the slip is marked as cancelled.

RareBetSports: Resolved with PR-14.

Zenith: Verified.

[M-2] All unlisted bets marked `Lost` instead of `Cancelled`

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- SlipRepository.sol#L602-L612

Description:

If admin unlists all bets in a `slip`, `gradeIndividualSlip` sets `SlipStatus.Lost` when `totalBetsWon == 0`, even though `totalBetsUnlisted` equal to the `bets.length`, which is unintuitive and unfair for the users.

```
    function gradeIndividualSlip(
        Slip storage slip,
        uint256 slipId
    ) internal returns (bool) {
        // ...

        slip.isGraded = true;
        if (totalBetsWon < minNumBetsPerSlip) {
            if (totalBetsWon > 0 && totalBetsUnlisted > 0) {
                slip.status = SlipStatus.Cancelled;
            } else {
>>>             slip.status = SlipStatus.Lost;
            }
            slip.isPaid = true;
            emit SlipGraded(slip.player, slipId, slip.status);
            return false;
        }

        slip.status = SlipStatus.Won;

        // NOTE: The multipliers array is indexed from 0 to maxNumBetsPerSlip
        // It should be noted that the first element of the array is always 0,
        // (this is for the losing none of the the bets)
        // So, the index of the multiplier to be used is totalBetsWon
        // If winning is between 3 and 6, this means indexes 0, 1, 2 are
    equal to 0
```

```
        uint256 multiplier = multipliers[totalBetsWon];
        require(multiplier > 0, "Multiplier must be greater than zero");
        slip.winningAmount =
            (slip.parlayAmount * multiplier) /
            MULTIPLIER_PRECISION;

        emit SlipGraded(slip.player, slipId, slip.status);
        return true;
    }
```

Recommendations:

If `totalBetsUnlisted > 0` and there are no Lost bets (`totalBetsUnlisted == slip.betIds.length`) , set `Cancelled` (refund) instead of `Lost`.

RareBetSports: Resolved with PR-20.

Zenith: Verified.

[M-3] `gradeBet` before `openUntil` exposes outcomes on-chain

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- SportOracle.sol#L180-L192

Description:

`gradeBet` can occur before `openUntil`. This leaks outcome on-chain during the betting window.

```
function gradeBet(
    bytes32 _hashId,
    Result _result
) external onlyGradingAdmin {
    require(bets[_hashId].hashId == _hashId, "Not a valid bet"); // Ensure
    the bet exists
    require(
        _result != Result.Incomplete,
        "Cannot set Incomplete result again"
    );
    bets[_hashId].result = _result; // Mark result as published
    emit BetGraded(_hashId, _result);
}
```

When a `slip` is created, it only validates that `bet.openUntil` hasn't passed, but doesn't check whether the `bet.result` is still `Incomplete`.

```
function _validateBets(
    bytes32[] calldata _betIds,
    bytes32[] memory playerIds
) internal view returns (bool hasFreebie, bytes32 freebieTeamId) {
    uint256 freebieCount = 0;

    for (uint256 i = 0; i < _betIds.length; i++) {
        bytes32 betId = _betIds[i];
        SportOracle.Bet memory bet = oracle.getBet(betId);
```

```
            require(
                block.timestamp < bet.openUntil,
                "Bet is not open for betting"
            );
            require(!bet.unlisted, "Bet is unlisted");

            require(
                !_isPlayerIdDuplicate(playerIds, bet.playerId),
                "Multiple bets with same playerId not allowed"
            );
            playerIds[i] = bet.playerId;

            if (bet.betType == SportOracle.BetType.Freebie) {
                freebieCount++;
                require(
                    freebieCount <= 1,
                    "Only one freebie bet allowed per slip"
                );
                hasFreebie = true;
                freebieTeamId = bet.teamId;
            }
        }
    }
```

Users can abuse this by create `slips` including a known `Won` bet.

Recommendations:

Add time check in `gradeBet`, require `block.timestamp > bet.openUntil`. Or In `SlipRepository._validateBets`, reject pre-graded bets, require `bet.result == Incomplete`.

RareBetSports: Resolved with PR-23.

Zenith: Verified.

4.2 Low Risk

A total of 9 low risk findings were identified.

[L-1] Improve argument validations on SlipRepository

`SEVERITY:` Low	`IMPACT:` Low
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- SlipRepository.sol

Description:

- In `_validateBets()`, explicitly check that bet IDs are not repeated within the `betIds` array and the bets are indeed registered by checking that their `hashId` is not zero.
- In `gradeSlips()`, `refundSlips()` and `claimSlips()` check that the given slip ID is valid within the enumeration (i.e., between 1 and `nextSlipId`).

Recommendations:

Even though these cases should fail due to other checks, consider adding the explicit validations to provide better defensive guarantees.

RareBetSports: Resolved with PR-15.

Zenith: Verified.

[L-2] Check if the bet has been previously invalidated

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- SportOracle.sol

Description:

The implementation of `invalidateBet()` fails silently if the bet was previously invalidated.

```
    function invalidateBet(bytes32 _hashId) public onlyDataAdmin {
        require(bets[_hashId].hashId != 0, "Bet with hashId does not exist");

        Bet storage bet = bets[_hashId];

        if (!bet.unlisted) {
            bet.unlisted = true;
            bet.result = Result.NotResulted;
            emit BetIsInactive(_hashId);
        }
    }
```

Recommendations:

Consider adding a check to ensure the bet has not been invalidated.

```
function invalidateBet(bytes32 _hashId) public onlyDataAdmin {
    require(bets[_hashId].hashId != 0, "Bet with hashId does not exist");
    require(!bets[_hashId].unlisted, "Bet already invalidated");

    Bet storage bet = bets[_hashId];

    bet.unlisted = true;
    bet.result = Result.NotResulted;
    emit BetIsInactive(_hashId);
}
```

RareBetSports: Resolved with PR-16.

Zenith: Verified.

[L-3] Improve validations in grade bet

`SEVERITY:` Low	`IMPACT:` Low
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- SportOracle.sol

Description:

There are multiple potential enhancements in the `gradeBet()` function:

- Check that the current result is `Incomplete` to avoid re-grading a bet.
- If the new result is `NotResulted`, then the bet would be partially invalidated, cause the `unlisted` flag would still be false. For better consistency, prevent setting the `NotResulted` result and always use `invalidateBet()`.

Recommendations:

Consider strengthening the validations in the `gradeBet()` function.

RareBetSports: Resolved with PR-17.

Zenith: Verified.

[L-4] Player ID duplicate check fails for zero values

`SEVERITY:` Low	`IMPACT:` Low
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- SlipRepository.sol

Description:

The implementation of `_validateBets()` checks for duplicate player IDs using the `_isPlayerIdDuplicate()` function, which traverses the array to check if the value is already present.

As the array is initially populated with zero values, a player ID with an actual zero value would conflict with the check and cause a permanent revert.

Recommendations:

Enforce non-zero player ID values when registering bets in the SportOracle contract.

RareBetSports: Resolved with PR-18.

Zenith: Verified.

[L-5] Unsafe ERC20 transfers used in `BatchSender` and `SlipRepository`

`SEVERITY:` Low	`IMPACT:` Low
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- BatchSender.sol#L38
- SlipRepository.sol#L428-L435
- SlipRepository.sol#L705

Description:

`BatchSender.batchSendTokens` and `SlipRepository.createSlipERC20` uses raw `IERC20.transferFrom` and `SlipRepository.withdrawToken` uses raw `transfer` relies on a boolean return. Many ERC20 tokens are non-standard (return no bool or revert differently), causing unexpected reverts.

Recommendations:

Use the `safeTransfer` and `safeTransferFrom` from the openzeppelin `SafeERC20` library.

RareBetSports: Resolved with PR-19. We will not need BatchTransfer so it is only updated for SlipRepository.

Zenith: Verified.

[L-6] Use of `transfer` in native payouts/refunds enables DoS

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- SlipRepository.sol#L546
- SlipRepository.sol#L668

Description:

Native transfers inside `claimSlips` and `refundSlips` use transfer (2300 gas). This can DoS payouts/refunds for recipients.

```
    function claimSlips(
        uint256[] calldata slipIds
    ) external nonReentrant {
        // ...
        // Second pass: perform all external calls
        for (uint256 i = 0; i < slipIds.length; i++) {
            uint256 currentSlipId = slipIds[i];
            Slip storage slip = slips[currentSlipId];

            if (slip.status == SlipStatus.Won && slip.isPaid) {
                if (slip.currency == Currency.Native) {
>>>                  payable(slip.player).transfer
            (slip.paidAmount); // Transfer native token
                } else if (slip.currency == Currency.ERC20) {
                    IERC20 tokenContract =
                    IERC20(slip.tokenAddress); // Get the token contract
                    tokenContract.safeTransfer(slip.player,
    slip.paidAmount); // Transfer ERC20 tokens
                }

                emit SlipClaimed(
                    slip.player,
                    currentSlipId,
                    slip.paidAmount,
                    slip.status
                );
            }
        }
    }
```

Recommendations:

Replace transfer with low-level call and check success result.

RareBetSports: Resolved with PR-21.

Zenith: Verified.

[L-7] Retroactive changes to `multipliers`, `minNumBetsPerSlip` and `maxNumBetsPerSlip` affect slip rewards

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- SlipRepository.sol#L602-L630

Description:

Owner can change `minNumBetsPerSlip`, `maxNumBetsPerSlip` and the `multiplier` after `slip` creation but before claim.

```
function setBetLimitsAndMultipliers(
    uint256[] calldata _multipliers,
    uint8 _minNumBetsPerSlip,
    uint8 _maxNumBetsPerSlip
) external onlyOwner {
    require
    (_multipliers.length > 0, "Multipliers array cannot be empty");

    uint8 minBets = 0;
    uint8 maxBets = 0;

    for (uint8 i = 0; i < _multipliers.length; i++) {
        if (_multipliers[i] > 0) {
            if (minBets == 0) {
                minBets = i; // First non-zero index
            }
            maxBets = i; // Last non-zero index
        }
    }

    require(
        minBets > 0 && maxBets > 0,
        "Invalid multipliers: no non-zero values"
    );
    require(
        minBets == _minNumBetsPerSlip && maxBets == _maxNumBetsPerSlip,
        "Invalid multipliers: min and max bets do not match"
    );

    minNumBetsPerSlip = minBets;
    maxNumBetsPerSlip = maxBets;

    uint256[] memory tempMultipliers =
    new uint256[](maxBets + 1);
    // zero index is used for winning zero bets

    for (uint8 i = 0; i <= maxBets; i++) {
        tempMultipliers[i] = _multipliers[i];
    }

    multipliers = tempMultipliers;

    emit BetsPerSlipUpdated(msg.sender, maxBets, minBets);
    emit MultipliersSet(msg.sender, tempMultipliers);
}
```

This retroactively changes user payout expectations, potentially creating inconsistent payouts for different users with the same slips. At worst, a user could create a slip with a relatively easy to predict but low `multiplier`. Instead of claiming the reward immediately, they could wait until the `multiplier` is updated to a higher value and then claim more than they should receive.

In the `maxNumBetsPerSlip` case, if the owner reduces `maxNumBetsPerSlip` after a slip is created, `totalBetsWon` at claim time can exceed `multipliers.length - 1`. This can cause an out-of-bounds read during grading/claim, reverting and bricking payouts.

Recommendations:

Snapshot the `minNumBetsPerSlip`, `maxNumBetsPerSlip` and the `multiplier` into each slip at creation, and use the snapshots during grading/payout.

RareBetSports: Resolved with PR-24.

Zenith: Verified.

[L-8] `invalidateBet` can overwrite graded bets

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- SportOracle.sol#L195-L205

Description:

`invalidateBet` can overwrite a previously graded bet to `NotResulted` , changing expected outcomes for the users.

```
function invalidateBet(bytes32 _hashId) public onlyDataAdmin {
    require(bets[_hashId].hashId ≠ 0, "Bet with hashId does not exist");

    Bet storage bet = bets[_hashId];
    if (!bet.unlisted) {
        bet.unlisted = true;
        bet.result = Result.NotResulted;
        emit BetIsInactive(_hashId);
    }
}
```

This can result in inconsistent reward payouts for users and create a race condition to claim rewards before they are invalidated.

Recommendations:

Disallow invalidation if `bet.result ≠ Incomplete`.

RareBetSports: Resolved with PR-22.

Zenith: Verified.

[L-9] `gradeSlips` may cause user reward loss

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- SlipRepository.sol#L514-L525
- SlipRepository.sol#L653

Description:

`gradeSlips` can be called by the control admin to grade a list of `slipIds`. It sets `isGraded = true` but doesn't send the `winningAmount` to the user if the slip status is `Won`.

```
function gradeSlips(
    uint256[] calldata slipIds
) external onlyControlAdmin returns (bool[] memory) {
    bool[] memory winningResults = new bool[](slipIds.length); // Array to
    store winning results

    for (uint256 i = 0; i < slipIds.length; i++) {
        Slip storage slip = slips[slipIds[i]];
        winningResults[i] = gradeIndividualSlip(slip, slipIds[i]);
    }

    return winningResults; // Return the array of winning results
}
```

Later, `claimSlips` reverts on already graded slips (requires `!isGraded`), leaving winners cannot claim their rewards.

```
    function claimSlips(
        uint256[] calldata slipIds
    ) external nonReentrant {
        require(!payoutsPaused, "Payouts are paused");
        require(
            !flaggedUsers[msg.sender],
            "User is pending review"
        );
        for (uint256 i = 0; i < slipIds.length; i++) {
            uint256 currentSlipId = slipIds[i];
            Slip storage slip = slips[currentSlipId]; // Get the slip

            require(!slip.isPaid, "Slip already paid");
            require(
                slip.player == msg.sender,
                "Allowed to grade own slip only"
            );
            require(!slip.isPaused, "Slip is paused");

>>>         gradeIndividualSlip(slip, currentSlipId); // Grade the slip

            if (slip.status == SlipStatus.Won) {
                slip.isPaid = true; // Mark payout as done
                slip.paidAmount = slip.winningAmount; // Update paid amount
            }
        }

        // Second pass: perform all external calls
        for (uint256 i = 0; i < slipIds.length; i++) {
            // ...
        }
    }
```

```
    function gradeIndividualSlip(
        Slip storage slip,
        uint256 slipId
    ) internal returns (bool) {
>>>     require(!slip.isGraded, "Only Ungraded Slips Allowed");
        require(!slip.isPaid, "Slip already paid out");
        require(!slip.isPaused, "Slip is paused");

        // ...

>>>     slip.isGraded = true;
        if (totalBetsWon < minNumBetsPerSlip) {
            if (totalBetsWon > 0 && totalBetsUnlisted > 0) {
                slip.status = SlipStatus.Cancelled;
            } else {
                slip.status = SlipStatus.Lost;
            }
            slip.isPaid = true;
            emit SlipGraded(slip.player, slipId, slip.status);
            return false;
        }

        slip.status = SlipStatus.Won;
}
```

Recommendations:

Allow `claimSlips` to pay already graded `slips` (skip re-grade if `isGraded == true` and use the status/amount). Or, make `gradeSlips` also perform the payout.

RareBetSports: Acknowledged

4.3 Informational

A total of 6 informational findings were identified.

[I-1] Multiplier field in Bet structure is not used

`SEVERITY:` Informational	`IMPACT:` Informational
`STATUS:` Acknowledged	`LIKELIHOOD:` Low

Target

- SportOracle.sol

Description:

The `multiplier` field in the Bet structure of the SportOracle contract is not used in any part of the protocol.

Recommendations:

Consider removing the field if this is not used to avoid potential confusion with the multipliers in the SlipRepository contract.

RareBetSports: Acknowledged.

[I-2] Owner is not needed in BatchSender

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- BatchSender.sol

Description:

The BatchSender contract defines an `owner` variable which is not used.

Recommendations:

Remove the `owner` variable.

RareBetSports: Acknowledged.

[I-3] Consider using a role-based authentication

`SEVERITY:` Informational	`IMPACT:` Informational
`STATUS:` Acknowledged	`LIKELIHOOD:` Low

Target

- SlipRepository.sol
- SportOracle.sol

Description:

The SportOracle and SlipRepository contracts can leverage an RBAC implementation such as OpenZeppelin's AccessControl library.

Currently, the contracts define different types of admin addresses, which can be changed to roles, with the added benefit of eventually having multiple accounts with the same authorization if needed.

Recommendations:

Change the Ownable mixin for the AccessControl library. The current owner should be the `DEFAULT_ADMIN`, and `controlAdmin`, `dataAdmin`, and `gradingAdmin` can be roles in the system.

RareBetSports: Acknowledged.

[I-4] Improve multipliers validation

`SEVERITY`: Informational	`IMPACT`: Informational
`STATUS`: Resolved	`LIKELIHOOD`: Low

Target

- SlipRepository.sol

Description:

The checks over the `_multipliers` array in `setBetLimitsAndMultipliers()` can be strengthened to provide better guarantees.

Recommendations:

Given `_minNumBetsPerSlip` and `_maxNumBetsPerSlip`, check that:

1. `_minNumBetsPerSlip > 0` and `_minNumBetsPerSlip ≤ _maxNumBetsPerSlip`
2. Each element from 0 up to `_minNumBetsPerSlip` (exclusive) is zero
3. Each element from `_minNumBetsPerSlip` up to `_maxNumBetsPerSlip` (inclusive) is non-zero (and optionally check these are sorted)
4. Each element from `_maxNumBetsPerSlip` up to the length of the array is zero

RareBetSports: Resolved with PR-25.

Zenith: Verified.

[I-5] Zero `hashId` creates inconsistent bet state

`SEVERITY`: Informational	`IMPACT`: Informational
`STATUS`: Resolved	`LIKELIHOOD`: Low

Target

- SportOracle.sol

Description:

Zero `hashId` is allowed when `createBet` is called. Existence checks rely on `bets[hashId].hashId == 0`, so a bet with `hashId == 0x0` can be created multiple times.

```
    function createBet(
        bytes32 hashId,
        uint32 multiplier, // in decimal basis points ie 1.00 = 100 basis
    points
        uint256 openUntil,
        bytes32 playerId,
        bytes32 teamId,
        BetType betType
    ) external onlyDataAdmin {
        // Ensure multiplier is positive
        require(multiplier > 0, "Multiplier must be greater than 0");
        // Ensure bet is active in the future
        require(openUntil > block.timestamp, "Bet end time must be in
    future");
        // Ensure bet with same hashId doesn't exist
>>>     require(
        bets[hashId].hashId == 0, "Bet with hashId already exists");

        Bet memory newBet = Bet({
            hashId: hashId,
            multiplier: multiplier,
            openUntil: openUntil,
            result: Result.Incomplete,
            unlisted: false,
            playerId: playerId,
            teamId: teamId,
            betType: betType
        });
        bets[hashId] = newBet;
        emit BetCreated
        (hashId, multiplier, openUntil, playerId, teamId, betType);
    }
```

`gradeBet` validates with `bets[_hashId].hashId == _hashId`, for `_hashId == 0x0`, this passes even if no proper bet was created, enabling grading/overwriting the zero slot. This breaks consistency and can lead to unexpected state mutations for the zero `hashId`.

Recommendations:

Prevent zero `hashId` across all entry points.

RareBetSports: Resolved with PR-26.

Zenith: Verified.

[I-6] Excess native in `batchSendEther` not refunded

`SEVERITY`: Informational	`IMPACT`: Informational
`STATUS`: Acknowledged	`LIKELIHOOD`: Low

Target

- BatchSender.sol#L48-L68

Description:

When `batchSendEther` is called with `msg.value > totalAmount`, the excess native is not refunded.

```
function batchSendEther(
    address[] calldata recipients,
    uint256[] calldata amounts
) external payable {
    require(recipients.length == amounts.length, "Recipients and amounts
    length mismatch");
    require(recipients.length > 0, "No recipients provided");

    uint256 totalAmount = 0;
    for (uint256 i = 0; i < amounts.length; i++) {
        totalAmount += amounts[i];
    }
    require(totalAmount <= msg.value, "Insufficient Ether sent");

    for (uint256 i = 0; i < recipients.length; i++) {
        require(recipients[i] != address(0), "Invalid recipient address");
        require(amounts[i] > 0, "Invalid transfer amount");

        (bool success, ) = recipients[i].call{value: amounts[i]}("");
        require(success, "Ether transfer failed");
    }
}
```

Recommendations:

Enforce equality, `require(totalAmount == msg.value)` or refund if `(msg.value > totalAmount)`.

RareBetSports: Acknowledged