shieldify

Souls Club

Revolver

SECURITY REVIEW

Date: 17 September 2025

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and has secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Vyper, Rust, Cairo, Move and Go.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Souls Club - Revolver

To create multiplayer and single-player games that are deeply integrated with social network interactions and digital identity, delivering a personalized experience powered by AI for every player while building a recognizable IP.

Revolver

A multi-cell elimination game contract built on zkSync with VRF-based randomness and upgradeable architecture.

Revolver is an on-chain game where players join cells by staking ETH. Cells are eliminated randomly using Verifiable Random Function (VRF) until only one cell remains. Winners share the prize pool proportionally to their stake in the winning cell.

Key Features

- **Multi-cell gameplay**: 6 cells that players can join
- **VRF-based randomness**: Secure, verifiable randomness for fair cell elimination
- **UUPS upgradeable**: Contract can be upgraded by the admin
- **Commission system**: Configurable commission percentage for sustainability
- **Signature-based access**: Backend signature verification for controlled access

Architecture

The project uses: - **Hardhat** with zkSync support - **OpenZeppelin** contracts for upgradeability and security - **TypeScript** for deployment and testing - **Abstract** blockchain (zkSync-based) as the target network

Game Flow

1. Players join cells by calling `joinGame()` with a valid backend signature
2. When 2+ unique players join, the round activates

3. After a timeout period, the backend requests randomness
4. The VRF system provides a random number to eliminate a cell
5. Process repeats until only one cell remains
6. Winners can claim their proportional share of the prize pool

4. Risk Classification

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

4.1 Impact

- **High** - results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** - results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** - losses will be limited but bearable, and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** - almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** - still relatively likely, although only conditionally possible
- **Low** - requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted 6 days, with a total of 96 hours dedicated by 2 researchers from the Shieldify team.

Overall, the code is well-written. The audit report flagged 4 Medium and 6 Low severity issues. They are mainly related to potential chain replay attack, inefficient payout distribution, centralization risks and VRF DoS, among other missed validation checks and potential string collision.

The Souls Club team has done a great job with their test suite and provided exceptional support to the Shieldify researchers.

5.1 Protocol Summary

Project Name	**Souls Club - Revolver**
Repository	revolver-contract
Type of Project	GameFi, ETH Staking, VRF
Audit Timeline	6 days
Review Commit Hash	b9c318f97f10a4f496bda8959aeb90696402e114
Fixes Review Commit Hash	701007ce045efbff5aaf14f215b64f1c900f5601

5.2 Scope

The following smart contracts were in the scope of the security review:

File	nSLOC
contracts/Revolver.sol	411
contracts/MockVRFSystem.sol	17
contracts/UtilityVault.sol	121
contracts/IVRFSystem.sol	3
Total	**552**

6. Findings Summary

The following issues have been identified, sorted by their severity:

- **Medium** issues: **4**
- **Low** issues: **6**

ID	Title	Severity	Status
[M-01]	Join Signature Lacks Domain Separation Leading to Cross-Deployment/Chain Replay	Medium	Fixed
[M-02]	Inefficient Payout Distribution Due to Quadratic Gas Complexity	Medium	Fixed
[M-03]	Critical Variables Can Be Changed Mid-Round, Causing Severe Inconsistencies	Medium	Fixed
[M-04]	VRF Callback Scales with Player Count Can Lead to OOG and Stuck `WaitingVRF`	Medium	Fixed
[L-01]	Payout Rounding Dust Leaks Value Every Round (Player Underpayment)	Low	Acknowledged
[L-02]	Missing Staleness Checks on VRF Randomness	Low	Acknowledged
[L-03]	Packed String Collision in Signed Payload Lets UUID Spoofing in `UtilityVault`	Low	Fixed
[L-04]	Lack of Practical Upper Threshold for Commission Percentage	Low	Fixed
[L-05]	Incorrect round Termination Path Allows Unfair Commission Deduction	Low	Acknowledged
[L-06]	`uniquePlayersCount` Not Updated After Eliminations Leading to Unnecessary Memory Allocation	Low	Acknowledged

7. Findings

[M-01] Join Signature Lacks Domain Separation Leading to Cross-Deployment/Chain Replay

Severity

Medium Risk

Description

In the `joinGame()` function in `Revolver`, the signed payload omits any domain/session fields:

```
// Verify signature - hash uid with msg.sender address (signature format
   unchanged)
bytes32 messageHash = keccak256(
    abi.encodePacked(uid, msg.sender, cellId)
);

bytes32 ethSignedMessageHash = MessageHashUtils.toEthSignedMessageHash(
   messageHash);
address recoveredSigner = ECDSA.recover(ethSignedMessageHash, signature);
if (recoveredSigner != backend) revert InvalidSignature();
```

The signed message does **not** commit to:

- `block.chainid` (chain),
- `address(this)` (contract deployment),
- `currentRoundId` (session),
- or any contract-config parameter like `baseStake`.

If the backend key is reused across networks (e.g., testnet/mainnet) or across multiple deployments/versions, a signature produced for `(uid, user, cellId)` in one environment is **valid verbatim** in another. Likewise, it's reusable across future rounds within the same deployment because `currentRoundId` is not included in the message being signed (note: the contract stores `usedCellHash` keyed with `currentRoundId`, but that only blocks duplicate joins in the same round, not signature reuse across rounds or deployments).

Impact

- Cross-chain / cross-deployment replay: A signature issued on chain A (or deployment X) can authorize a join on chain B (or deployment Y) so long as the backend address is the same.
- Cross-round replay: The same signature can be reused in later rounds of the same deployment so long as (`uid`, `user`, `cellId`) repeats (the contract prevents intra-round replay via `usedCellHash`, but not inter-round reuse of the same signature).
- Outcome: not direct theft (user still pays `baseStake`), but it weakens anti-bot/eligibility controls, undermines "one-shot" off-chain gatekeeping, and enables unintended reuse of an authorization the backend believed was bound to a specific environment/round.

Location of Affected Code:

File: contracts/Revolver.sol#L398

```
function joinGame(
    uint256 cellId,
    string calldata uid,
    bytes calldata signature
) external payable nonReentrant {
    if (cellId < 1 || cellId > NUM_CELLS) revert InvalidCell();
    Round storage round = rounds[currentRoundId];

    if (round.status != RoundStatus.Gathering) revert RoundNotGathering()
       ;
    if (round.locked) revert RoundLocked();
    if (msg.value != baseStake) revert IncorrectStakeAmount();

    bytes32 uidHash = keccak256(abi.encodePacked(uid));

    address existingUidOwner = usedUidsToPlayer[uidHash];
    if (existingUidOwner == address(0)) {
        usedUidsToPlayer[uidHash] = msg.sender;
    } else if (existingUidOwner != msg.sender) {
        revert UidAlreadyUsed();
    }

    bytes32 existingPlayerUid = playerToUid[msg.sender];
    if (existingPlayerUid == bytes32(0)) {
        playerToUid[msg.sender] = uidHash;
    } else if (existingPlayerUid != uidHash) {
        revert UidAlreadyUsed();
    }

    // Verify signature - hash uid with msg.sender address (signature
       format unchanged)
    bytes32 messageHash = keccak256(
        abi.encodePacked(uid, msg.sender, cellId)
    );

    // Prevent replay within the same round by including the current
       round id in the used key
    bytes32 usedKey = keccak256(
        abi.encodePacked(currentRoundId, uid, msg.sender, cellId)
    );
    if (usedCellHash[usedKey]) revert UidAlreadyUsed();
    bytes32 ethSignedMessageHash = MessageHashUtils.
       toEthSignedMessageHash(
        messageHash
    );
    address recoveredSigner = ECDSA.recover(
        ethSignedMessageHash,
        signature
    );

    if (recoveredSigner != backend) revert InvalidSignature();

    usedCellHash[usedKey] = true;

    bytes32 roundUidKey = keccak256(abi.encodePacked(currentRoundId,
       uidHash));
    if (uidToEntry[roundUidKey] >= 3) revert UidAlreadyUsed();
    uidToEntry[roundUidKey]++;

    Cell storage cell = round.cells[cellId];
    if (!cell.isActive) {
        cell.isActive = true;
        round.activeCellsCount++;
    }

    if (cell.stakeByPlayer[msg.sender] > 0) revert AlreadyInCell();


    if (round.playerInfo[msg.sender].stake == 0) {
        round.uniquePlayersCount++;
    }

    cell.players.push(msg.sender);
    cell.totalStake += msg.value;
    cell.stakeByPlayer[msg.sender] += msg.value;
    round.playerInfo[msg.sender].stake += msg.value;
    round.pool += msg.value;

    emit GameJoined(currentRoundId, msg.sender, cellId, uid, msg.value);
    // Activate only when there are 2+ active cells AND at least two
       different users across those cells
    if (round.activeCellsCount >= 2 && round.startTimestamp == 0) {
        // Use a more efficient approach with early exit and player
           tracking
        bool hasTwoUniqueUsersInTwoCells = _checkTwoUniqueUsersInTwoCells
           (round);

        if (hasTwoUniqueUsersInTwoCells) {
            round.startTimestamp = block.timestamp;
            emit RoundActivated(currentRoundId, round.startTimestamp);
        }
    }
}
```

The problem is the `messageHash` preimage: it only uses (`uid` , `msg.sender` , `cellId`) and omits `block.chainid` , `address(this)` , `currentRoundId` , and (optionally) `baseStake` .

Recommendation

Bind domain & session (minimal, safe change)

Include fixed domain identifiers and the session in the signed preimage:

```
bytes32 messageHash = keccak256(
    abi.encodePacked(
        block.chainid,
        address(this),
        currentRoundId,
        uid,
        msg.sender,
        cellId,
        baseStake // optional but useful to prevent price-mismatch
           replays
    )
);
bytes32 ethSigned = MessageHashUtils.toEthSignedMessageHash(messageHash);
address recovered = ECDSA.recover(ethSigned, signature);
```

- Commit the domain and session:

```
bytes32 messageHash = keccak256(
    abi.encodePacked(block.chainid, address(this), currentRoundId, uid,
       msg.sender, cellId, baseStake)
);
```

Also, you can consider EIP-712 (structured data) with an explicit `JOIN_TYPEHASH` .

Team Response

Fixed.

[M-02] Inefficient Payout Distribution Due to Quadratic Gas Complexity

Severity

Medium Risk

Description

The `_distributeProportionalPayouts()` function uses nested for-loops with a linear search to find whether a player already exists in the `uniquePlayers` array.
This creates **quadratic gas complexity ($O(p^2)$)** with respect to the number of participants across all cells.

As the participant count grows, gas usage increases disproportionately. In large rounds, the protocol may end up spending **more on gas fees than it earns in commission**, leading to economic inefficiency and making the protocol vulnerable to denial-of-service scenarios if a larger number of participants inflate participation.

Location of Affected Code

File: contracts/Revolver.sol#L826

```
function _distributeWinnerPayouts( Round storage round, uint256
   payoutPool, uint256 roundId ) private returns (address[] memory) {
  // code
  unchecked {
    // Single pass: collect active stakes and unique players
    for (uint256 i = 1; i <= NUM_CELLS; ++i) {
        Cell storage cell = round.cells[i];
        if (!cell.isActive) continue;

        uint256 playersLength = cell.players.length;
        for (uint256 j = 0; j < playersLength; ++j) {
            address player = cell.players[j];
            uint256 stakeInCell = cell.stakeByPlayer[player];
            if (stakeInCell == 0) continue;

            // Find or add player to unique list
            uint256 playerIndex = type(uint256).max;
            for (uint256 k = 0; k < uniquePlayerCount; ++k) {
                if (uniquePlayers[k] == player) {
                    playerIndex = k;
                    break;
                }
            }
            if (playerIndex == type(uint256).max) {
                // New player
                playerIndex = uniquePlayerCount;
                uniquePlayers[uniquePlayerCount] = player;
                playerTotalStakes[uniquePlayerCount] = stakeInCell;
                uniquePlayerCount++;
            } else {
                // Existing player - add to their stake
                playerTotalStakes[playerIndex] += stakeInCell;
            }

            totalActiveStake += stakeInCell;
        }
    }
    return winners;
}
```

Recommendation

- Use libraries like iterable mapping to store all the Unique players
- Instead of linear search, implement better and efficient algorithms like binary search or Graph search
- Limit the number of users per cell

Team Response

Fixed.

[M-03] Critical Variables Can Be Changed Mid-Round, Causing Severe Inconsistencies

Severity

Medium Risk

Description

The protocol allows the admin to change `commissionPercent` and `baseStake` in the middle of an active round.
These two variables are fundamental to round calculations, and modifying them mid-game can lead to severe inconsistencies between users:

- **Changing Commission Percent Mid-Round**
  If the commission is updated while a round is ongoing, users who joined earlier are subject to a different fee structure compared to those who joined later.
  This creates unfairness, breaks predictability, and can lead to disputes over payouts.
- **Changing Base Stake Mid-Round**
  If the base stake is changed during an active round, players who already joined did so under different requirements than those joining afterwards.
  This inconsistency may allow some players to enter with a lower stake or force others to meet a higher threshold, breaking the fairness and balance of the game.

Both cases result in **protocol-wide inconsistencies**, undermine trust in the system, and can make the round logic disputed.

Location of Affected Code

File: contracts/Revolver.sol

```
 function setCommissionPercent(uint256 percent) external onlyAdmin {
    if (percent > 10000) revert TooHighCommission();
    commissionPercent = percent;
}
```

```
function setBaseStake(uint256 _newStake) external onlyAdmin {
    if (_newStake == 0) revert StakeTooLow();
    baseStake = _newStake;
}
```

Recommendation

- Restrict updates to `commissionPercent` and `baseStake` so they can only be applied **between rounds**, never during an active round.
- Introduce versioning or effective-after mechanisms (e.g., changes only apply starting from the next round).

Team Response

Fixed.

[M-04] VRF Callback Scales with Player Count Can Lead to OOG and Stuck `WaitingVRF`

Severity

Medium Risk

Description

In `Revolver` the `randomNumberCallback(uint256 requestId, uint256 randomNumber)` calls `_checkSolePlayer(round)` every callback.

However, the `_checkSolePlayer()` function iterates all active cells and then all players in those cells:

```
for (uint256 i = 1; i <= NUM_CELLS; ++i) {
    Cell storage cell = round.cells[i];
    if (!cell.isActive || cell.players.length == 0) continue;
    uint256 playersLength = cell.players.length;
    for (uint256 j = 0; j < playersLength; ++j) {
        address p = cell.players[j];
        if (cell.stakeByPlayer[p] == 0) continue;
        // code
    }
}
```

With enough players, this scan exceeds the tx gas budget and reverts. The revert occurs before settlement logic; `status` remains `WaitingVRF`.

This results in a state where rounds with thousands of active players can't progress past the VRF callback. The contract stays in `WaitingVRF`; no further eliminations; no settlement; funds are effectively stuck until an upgrade/migration.

Let's do some quick math on how many players this bug requires:

- `_checkSolePlayer()` does ~2 SLOADs per player (`players[j] + stakeByPlayer[p]`) + loop overhead **~5k gas per player** (rough order).
- With a ~30M gas envelope, OOG around **~6,000 active players** across the still-active cells: `30,000,000 / 5,000   6k`.

Location of Affected Code:

File: contracts/Revolver.sol#L531

```
function randomNumberCallback( uint256 requestId, uint256 randomNumber )
   external onlyVRFSystem {
    uint256 roundId = vrfRequestToRoundId[requestId];
    Round storage round = rounds[roundId];
    if (round.status != RoundStatus.WaitingVRF) revert InvalidState();
    // ... eliminate a cell, decrement activeCellsCount, emit ...

    //  Unbounded scan over all players in all active cells
    if (_checkSolePlayer(round)) {
        round.status = RoundStatus.Finished;
        finishGameProportional(roundId);
        return;
    }

    if (round.activeCellsCount == 1) {
        round.status = RoundStatus.Finished;
        distributePayout(roundId);
    } else {
        round.status = RoundStatus.Gathering;
        round.startTimestamp = block.timestamp;
        round.locked = true;
    }
}
```

[language=solidity]

File: contracts/Revolver.sol#L774

```
function _checkSolePlayer(Round storage round) private view returns (bool
   ) {
    address solePlayer;
    bool foundPlayer = false;

    for (uint256 i = 1; i <= NUM_CELLS; ++i) {
        Cell storage cell = round.cells[i];
        if (!cell.isActive || cell.players.length == 0) continue;

        uint256 playersLength = cell.players.length;
        for (uint256 j = 0; j < playersLength; ++j) {
            address p = cell.players[j];
            if (cell.stakeByPlayer[p] == 0) continue;

            if (!foundPlayer) {
                solePlayer = p;
                foundPlayer = true;
            } else if (p != solePlayer) {
                return false; // Multiple players found
            }
        }
    }
    return foundPlayer;
}
```

Recommendation

1. Remove `_checkSolePlayer()` from `randomNumberCallback()`. In the callback, only:
   - mark eliminated cell inactive,
   - decrement `activeCellsCount`,
   - set `status/locked`,
   - emit event.
2. Decide whether to finish outside the callback (e.g., next backend tick) or maintain an `O(1)` counter:
   - Track `uniqueActivePlayers` (increment on first active stake, decrement when player loses last active stake), or
   - Track per-player `activeCellCount/activeStakeTotal` and adjust only for players in the eliminated cell (bounded by cell size).
3. Consider a participant cap per round/cell if you retain any per-player scans.

Team Response

Fixed.

[L-01] Payout Rounding Dust Leaks Value Every Round (Player Underpayment)

Severity

Low Risk

Description

Let's take a look at the following:

- **Single-cell finish** (`distributePayout() → _distributeWinnerPayouts()`)

```
uint256 playerPayout = (stakeInCell * payoutPool) / winning.totalStake;
prizeBalance[p] += playerPayout; // remainder ignored
```

- **Proportional finish** (`finishGameProportional() → _distributeProportionalPayouts()`)

```
uint256 playerPayout = (playerStake * payoutPool) / totalActiveStake;
prizeBalance[player] += playerPayout; // remainder ignored
```

In both paths, per-winner payouts use integer division. The sum of floors is strictly `payoutPool` and is < `payoutPool` whenever there's a remainder. The leftover remainder is not assigned to any winner and is not added to `accumulatedCommission`.

- **Commission is fixed up-front**:

```
uint256 commission = (round.pool * commissionPercent) / 10000;
accumulatedCommission += commission;
uint256 payoutPool = round.pool - commission;
```

The remainder from the per-winner division is not included in `commission`. * **No path to recover the remainder**:

- Only winners' `prizeBalance` can be claimed (`claimAllPrizes()`), and it is credited solely by the floored shares above.
- `withdrawCommission()` can only withdraw `accumulatedCommission`.
- There is **no function** that sweeps/distributes the arithmetic remainder.
- Result: **systematic underpayment** of winners; wei accumulates in the contract at each settlement.

Concrete example

Pool = 15 wei, commission = 0% (for simplicity), three winners with equal stake.

- `payoutPool = 15`.
- Each `playerPayout = 15 / 3 = 5` sum = 15, **no remainder** (OK).

Now change to `payoutPool = 14` (e.g., due to 5% commission on 15).

- Each `playerPayout = 14 / 3 = 4` (floor).
- Sum credited = `4 + 4 + 4 = 12` **< 14 2 wei stranded** in the contract balance.
- This wei is **not** part of `accumulatedCommission` and **not** claimable by winners.

This occurs for most non-even splits and **compounds every round**.

Location of Affected Code

File: contracts/Revolver.sol#L586

```
function distributePayout(uint256 roundId) internal {
    Round storage round = rounds[roundId];
    uint256 commission = (round.pool * commissionPercent) / 10000;
    accumulatedCommission += commission;

    uint256 payoutPool = round.pool - commission;              // ←
       payoutPool fixed here

    address[] memory winners = _distributeWinnerPayouts(round, payoutPool
       , roundId);
    emit GameFinished(roundId, winners, payoutPool, commission);
    advanceRound();
}
```

File: contracts/Revolver.sol#L807

```
function _distributeWinnerPayouts(
    Round storage round,
    uint256 payoutPool,
    uint256 roundId
) private returns (address[] memory) {
    // code
    unchecked {
        for (uint256 j = 0; j < winnersCount; ++j) {
            address p = winning.players[j];
            // code
            uint256 playerPayout = (stakeInCell * payoutPool) / winning.
               totalStake;  // ← floor division
            round.playerInfo[p].payout = playerPayout;
            prizeBalance[p] += playerPayout;
                                                  // ← remainder
               never assigned
        }
    }
    // code
}
```

File: contracts/Revolver.sol#L605

```
function finishGameProportional(uint256 roundId) internal {
    Round storage round = rounds[roundId];
    uint256 commission = (round.pool * commissionPercent) / 10000;
    accumulatedCommission += commission;

    uint256 payoutPool = round.pool - commission;              // ←
       payoutPool fixed here

    address[] memory winners = _distributeProportionalPayouts(round,
       payoutPool, roundId);
    emit GameFinished(roundId, winners, payoutPool, commission);
    advanceRound();
}
```

File: contracts/Revolver.sol#848

```
function _distributeProportionalPayouts( Round storage round, uint256
   payoutPool, uint256 roundId ) private returns (address[] memory) {
    // code
    unchecked {
        // code
        for (uint256 i = 0; i < uniquePlayerCount; ++i) {
            address player = uniquePlayers[i];
            uint256 playerStake = playerTotalStakes[i];
            uint256 playerPayout = (playerStake * payoutPool) /
               totalActiveStake;     // ← floor division
            round.playerInfo[player].payout = playerPayout;
            prizeBalance[player] += playerPayout;
                                                     // ← remainder never
               assigned
            // code
        }
    }
    // code
}
```

Recommendation

Track `remaining = payoutPool`, subtract each assigned amount and give the last recipient the remainder (or add remainder to `accumulatedCommission`).

```
uint256 remaining = payoutPool;
for (uint256 i=0; i<winnersCount-1; ++i) {
    uint256 amt = stakeInCell[i] * payoutPool / winning.totalStake;
    remaining -= amt;
    prizeBalance[winners[i]] += amt;
}
prizeBalance[winners[winnersCount-1]] += remaining;
```

Team Response

Acknowledged.

[L-02] Missing Staleness Checks on VRF Randomness

Severity

Low Risk

Description

The randomness returned by the VRF system is accepted without any staleness validation. This means a delayed or manipulated VRF callback can still be applied to a round even after its state has changed. Since the random number is directly used to determine which cell gets eliminated, a stale or predictable value allows an attacker (or compromised VRF system) to bias the outcome of the round. This undermines fairness and can result in the round being manipulated to favour specific participants.

Location of Affected Code

File: contracts/Revolver.sol#531

```
function randomNumberCallback( uint256 requestId, uint256 randomNumber )
     external onlyVRFSystem {
    uint256 roundId = vrfRequestToRoundId[requestId];
    Round storage round = rounds[roundId];
    if (round.status != RoundStatus.WaitingVRF) revert InvalidState();

    // collect only active and occupied cells
    uint256[] memory occupiedCells = new uint256[](round.activeCellsCount
       );
    uint256 occupiedCount;

    for (uint256 i = 1; i <= NUM_CELLS; i++) {
        if (round.cells[i].isActive && round.cells[i].players.length > 0)
            {
            occupiedCells[occupiedCount] = i;
            occupiedCount++;
        }
    }
    // code
}
```

Recommendation

Consider applying the following changes: - Add strict staleness checks so that VRF responses are only valid within a short and predefined time window after the request.
- Reject any VRF responses that arrive after the round state has advanced.

Team Response

Acknowledged.

[L-03] Packed String Collision in Signed Payload Lets UUID Spoofing in `UtilityVault`

Severity

Low Risk

Description

In `UtilityVault.sol`:

```
// taskConfirmationFee
bytes32 messageHash = keccak256(abi.encodePacked(
    block.chainid,
    address(this),
    "task",
    taskUuid,         // string (dynamic)
    authUserUuid,     // string (dynamic)
    msg.sender,
    msg.value
));

// paymentConfirmation (same pattern; "payment", itemUuid, authUserUuid)
bytes32 messageHash = keccak256(abi.encodePacked(
    block.chainid,
    address(this),
    "payment",
    itemUuid,         // string (dynamic)
    authUserUuid,     // string (dynamic)
    msg.sender,
    msg.value
));
```

`abi.encodePacked` concatenates dynamic arguments **without length delimiters**. With **adjacent dynamic strings** (`taskUuid` and `authUserUuid` / `itemUuid` and `authUserUuid`), **different pairs** can yield the **same concatenated bytes** the **same** `messageHash`. Example: `("ab","c")` and `("a","bc")` both produce the bytes `"abc"` for that section.

- The backend signs a hash for `(taskUuid_A, authUserUuid_A)`.
- A user can submit `(taskUuid_B, authUserUuid_B)` whose concatenation matches, so the contract recomputes the **same** hash; the signature **verifies**; events log the **attacker-chosen strings**.
- This **spoofs/misattributes** which UUIDs were "paid" and breaks off-chain accounting/audit trails.

Secondary effect `usedSignatures[messageHash]` is keyed by the **ambiguous** hash. Two logically different requests that collide will cause `SignatureAlreadyUsed()` after the first is consumed **grief** legitimate follow-ups.

This will lead to on-chain evidence that can claim payment for **different UUIDs** than the backend intended. Also, a crafted collision can preempt a later legitimate payment by tripping `SignatureAlreadyUsed`.

Location of Affected Code:

File: contracts/UtilityVault.sol#L94

```
function taskConfirmationFee(
    string calldata taskUuid,
    string calldata authUserUuid,
    bytes calldata signature
) external payable nonReentrant {
    if (msg.value == 0) revert InvalidAmount();

    bytes32 messageHash = keccak256(
        abi.encodePacked( // <-- uses packed encoding
                block.chainid,
                address(this),
                "task",
                taskUuid, // <-- dynamic string
                authUserUuid, // <-- dynamic string, adjacent to taskUuid
                msg.sender,
                msg.value
            )
    ); // <-- vulnerable to ("ab","c") vs ("a","bc") collision

    _consumeSignature(messageHash, signature);

    totalDeposits += msg.value;
    emit TaskConfirmationFee(msg.sender, taskUuid, authUserUuid, msg.
       value);
}
```

File: contracts/UtilityVault.sol#L123

```
function paymentConfirmation(
    string calldata itemUuid,
    string calldata authUserUuid,
    bytes calldata signature
) external payable nonReentrant {
    if (msg.value == 0) revert InvalidAmount();

    bytes32 messageHash = keccak256(
        abi.encodePacked( // <-- uses packed encoding
                block.chainid,
                address(this),
                "payment",
                itemUuid, // <-- dynamic string
                authUserUuid, // <-- dynamic string, adjacent to itemUuid
                msg.sender,
                msg.value
            )
    ); // <-- same packed-collision issue

    _consumeSignature(messageHash, signature);

    totalDeposits += msg.value;
    emit PaymentConfirmation(msg.sender, itemUuid, authUserUuid, msg.
       value);
}
```

Recommendation

Use `abi.encode` (not packed) so lengths are encoded:

```
bytes32 messageHash = keccak256(abi.encode(
    block.chainid,
    address(this),
    keccak256("task"),          // or a typehash/enum
    taskUuid,
    authUserUuid,
    msg.sender,
    msg.value
));
```

- Or hash each string separately when packing:

```
bytes32 messageHash = keccak256(abi.encodePacked(
    block.chainid,
    address(this),
    "task",
    keccak256(bytes(taskUuid)),
    keccak256(bytes(authUserUuid)),
    msg.sender,
    msg.value
));
```

Also, you can move to **EIP-712** typed data with explicit fields (best solution).

Team Response

Fixed.

[L-04] Lack of Practical Upper Threshold for Commission Percentage

Severity

Low Risk

Description

The function `setCommissionPercent()` currently allows the admin to set `commissionPercent` to any value up to `10000` (representing 100%).
While technically valid, setting a very high commission (e.g., 80–100%) would result in **significant user losses** and make participation economically unviable.
This is not a direct security vulnerability but poses a **severe economic risk**, as users would lose trust and abandon the protocol if commissions become unreasonable.

Location of Affected Code

File: contracts/Revolver.sol#680

```
function setCommissionPercent(uint256 percent) external onlyAdmin {
    if (percent > 10000) revert TooHighCommission();
    commissionPercent = percent;
}
```

Recommendation

Introduce a **practical upper bound** for `commissionPercent` (e.g., 5–15%) to ensure commissions remain fair and sustainable for users, while still providing revenue to the protocol.

Team Response

Fixed.

[L-05] Incorrect round Termination Path Allows Unfair Commission Deduction

Severity

Low Risk

Description

The `finishGame()` function can be called directly instead of proceeding through the intended round-elimination flow.
When this happens, the protocol prematurely finishes the round using `finishGameProportional()`, which applies commission deductions.

This creates a scenario where: - Users lose part of their funds to commission **without the round being fairly resolved through eliminations**.
- The purpose of joining the round is undermined, as players are penalized by commission despite the game not being played out.
- The outcome appears arbitrary and unfair, damaging trust in the protocol.

Effectively, this function enables an unintended shortcut that benefits the protocol while penalizing users for no reason, making participation economically irrational.

Location of Affected Code

File: contracts/Revolver.sol#L515

```
function finishGame(uint256 traceId) external onlyBackend {
    Round storage round = rounds[currentRoundId];
    if (round.status != RoundStatus.Gathering) revert InvalidState();
    if (round.uniquePlayersCount == 0) revert InvalidState();
    round.status = RoundStatus.Finished;
    finishGameProportional(currentRoundId);
}
```

Recommendation

- Enforce that game termination only occurs via the designed elimination-based flow unless an exceptional safeguard condition (e.g., emergency halt) is triggered.
- If `finishGame()` is required for backend operations, ensure it bypasses commission deductions.

Team Response

Acknowledged.

[L-06] `uniquePlayersCount` Not Updated After Eliminations Leading to Unnecessary Memory Allocation

Severity

Low Risk

Description

The variable `uniquePlayersCount` is never decreased after the elimination of cells in a round.
As a result, when arrays like `tempWinners`, `uniquePlayers`, and `playerTotalStakes` are pre-allocated using `round.uniquePlayersCount`, they may reserve **more memory than required**.

This does not break core functionality but leads to **wasted memory allocation and higher gas usage**, especially in rounds with multiple eliminations where the actual number of unique players is significantly smaller than the pre-allocated size.

Location of Affected Code

File: contracts/Revolver.sol#L848

```
function _distributeProportionalPayouts( Round storage round, uint256
   payoutPool, uint256 roundId ) private returns (address[] memory) {
  // code
  // Pre-allocate arrays based on maximum possible size
  address[] memory tempWinners = new address[](round.uniquePlayersCount);
  address[] memory uniquePlayers = new address[](round.uniquePlayersCount
     );
  uint256[] memory playerTotalStakes = new uint256[](round.
     uniquePlayersCount);
  uint256 winnerCount = 0;
  uint256 uniquePlayerCount = 0;
  uint256 totalActiveStake = 0;

  unchecked {
      // Single pass: collect active stakes and unique players
      for (uint256 i = 1; i <= NUM_CELLS; ++i) {
          Cell storage cell = round.cells[i];
          if (!cell.isActive) continue;

          uint256 playersLength = cell.players.length;
          for (uint256 j = 0; j < playersLength; ++j) {
              address player = cell.players[j];
              uint256 stakeInCell = cell.stakeByPlayer[player];
              if (stakeInCell == 0) continue;

              // Find or add player to unique list
              uint256 playerIndex = type(uint256).max;
              for (uint256 k = 0; k < uniquePlayerCount; ++k) {
                  if (uniquePlayers[k] == player) {
                      playerIndex = k;
                      break;
                  }
              }
```

```
                if (playerIndex == type(uint256).max) {
                    // New player
                    playerIndex = uniquePlayerCount;
                    uniquePlayers[uniquePlayerCount] = player;
                    playerTotalStakes[uniquePlayerCount] = stakeInCell;
                    uniquePlayerCount++;
                } else {
                    // Existing player - add to their stake
                    playerTotalStakes[playerIndex] += stakeInCell;
                }

                totalActiveStake += stakeInCell;
            }
        }

        // Distribute payouts based on collected stakes
        for (uint256 i = 0; i < uniquePlayerCount; ++i) {
            address player = uniquePlayers[i];
            uint256 playerStake = playerTotalStakes[i];
            uint256 playerPayout = (playerStake * payoutPool) /
               totalActiveStake;
            round.playerInfo[player].payout = playerPayout;
            prizeBalance[player] += playerPayout;
            tempWinners[winnerCount] = player;
            winnerCount++;
        }
    }
```

```
    // Create final winners array with exact size
    address[] memory winners = new address[](winnerCount);
    for (uint256 i = 0; i < winnerCount; ++i) {
        winners[i] = tempWinners[i];
    }

    return winners;
}
```

Recommendation

- Recalculate or adjust `uniquePlayersCount` whenever players are eliminated.
- Alternatively, allocate arrays dynamically based on the **current number of active players** instead of relying solely on the initial `uniquePlayersCount`.

Team Response

Acknowledged.

our shielding · Your smart contracts, our shielding · Your smart

shieldify

Thank you!