Dragon — A Spoiler-Free Reverse-Engineering Case Study

pwnable.kr · Rookiss · Binary exploitation · Ghidra / Linux ELF

Introduction

Dragon is a binary-exploitation challenge from pwnable.kr's Rookiss category. This article documents my reverse-engineering workflow and the general lessons I took away, without revealing the challenge's vulnerability, solution, payload, or flag. I deliberately omit details that would allow a reader to reconstruct the solve.

Initial reconnaissance

I began by inspecting the supplied executable with standard Linux tooling. The file command identified a 32-bit, little-endian Intel x86 ELF for Linux. The executable is dynamically linked and retains symbols, making function-level exploration easier in a disassembler. I also used checksec to survey mitigations: my notes recorded NX enabled, a stack canary, partial RELRO, and no PIE.

From strings to hypotheses

I used strings to identify possible user-facing prompts, imported library functions, and interesting execution paths. Strings are clues rather than proof: the presence of a function name or command-related string does not, by itself, establish exploitability. My next step was to check how those references were used in the actual control flow.

Mapping the program in Ghidra

I loaded the executable into Ghidra and started at main, then followed the calls into the game's menu and combat routines. I mapped which functions accept user input, which ones modify program state, and where the program makes decisions about continuing or ending an interaction. Because the binary retains function symbols, I could organize the investigation around recognizable routines instead of starting entirely from unnamed addresses.

How I evaluated suspicious behavior

I focused on the relationship between state transitions, memory ownership, and later uses of data. For each potentially interesting operation, I asked: What is the data type? Who allocates or owns the object? Which branches change its state? Does a later operation assume an invariant that earlier code may have invalidated? I compared decompiled pseudocode against assembly whenever signedness, pointer arithmetic, or call behavior was ambiguous.

Why low-level details matter

This exercise reinforced that a C-like decompiler view can hide important distinctions. A narrow integer can behave differently from a wider one; signed and unsigned comparisons have different meanings; and an apparently simple structure access may depend on exact layout and lifetime. Rather than trusting a single decompiler expression, I learned to confirm the underlying instructions and reason about what the CPU actually reads or writes.

General memory-safety lessons

A useful habit in binary analysis is to trace allocation, mutation, release, and subsequent access as one connected lifecycle. Similarly, when reviewing arithmetic, track both the intended value range and the representation actually used in memory. These are general techniques for identifying classes of implementation mistakes; this article intentionally does not identify which observations, if any, form the solution to Dragon.

My workflow going forward

I would repeat this process in a structured order: characterize the binary; inspect its external interfaces; build a high-level call graph; document state and ownership; verify suspicious decompiler output in assembly; test hypotheses in a debugger; and keep separate notes for observations versus assumptions. This makes the investigation more reproducible without turning the public article into a walkthrough.

Closing

The most valuable part of Dragon for me was practicing disciplined reverse engineering: following control flow, validating low-level semantics, and distinguishing a plausible hypothesis from something established by code. I am intentionally withholding challenge-specific findings, exploitation steps, addresses, and artifacts so others can investigate independently.

Publication note

This is a spoiler-free learning reflection, not a solution or walkthrough. pwnable.kr asks participants to refrain from publishing solutions to challenges outside the Toddler's Bottle category, including Rookiss. Readers should consult the site's current rules before sharing challenge-specific material.