Windows1 CTF Writeup

Challenge Overview

The goal of this challenge was to gain access to a flag by bypassing a two-step validation process in a Windows binary named `numbers.exe`. The binary first prompts the user for an authorization code, and then for a password. Providing the correct inputs leads to the flag being revealed.

Analysis

Upon running the binary, two sequential prompts appear:

1. An **authorization number** is requested.
2. If successful, a **password** must then be entered.

A quick inspection of the binary using tools like Ghidra or x64dbg reveals a couple of interesting conditions:

- The first check compares the input against a UNIX timestamp or related value.
- The second check performs a multiplication between the first input and the second input, expecting a very specific result.

This suggests that the challenge revolves around logic flaws or integer manipulation, rather than memory corruption.

Key Code Analysis

```
// Authorization check
if (input == time(NULL)) {
    // valid
} else {
    // rejected
}

// Password check
if (input1 * input2 == 0x13333337) {
    // vault();
}
```

- The first condition checks whether the input equals the current UNIX timestamp. Since this value constantly changes, it's almost impossible to hit the exact value manually.
- However, due to how negative integers are represented in memory (two's complement), a bypass is possible.
- The second condition expects the product of the first and second inputs to equal a specific constant (`0x13333337` or `322122551` in decimal).

Exploitation and Payload Strategy

Step 1: Bypassing the Authorization

- By entering `-1` as the first input, we exploit how integers are stored in memory:
  - `-1` is stored as `0xFFFFFFFF`, which is a very large unsigned integer.
  - This can bypass the comparison logic due to signed/unsigned mismatches or logic flaws in how the input is validated.

Step 2: Solving the Password Equation

- The program checks:

```
if (-1 * password == 0x13333337)
```

- To satisfy this, the second input must be:

```
password = -0x13333337  # → -322122551
```

Final Payload:

```
-1
-322122551
```

Python Automation Script:

```
import subprocess

p = subprocess.Popen(["numbers.exe"], stdin=subprocess.PIPE,
stdout=subprocess.PIPE, text=True)

p.stdin.write("-1\n")
p.stdin.write("-322122551\n")
p.stdin.flush()

output, _ = p.communicate()
print(output)  # Should print the flag
```

Result and Solution

By supplying the values `-1` and `-322122551`, the binary bypasses both checks and reveals the flag:

```
🏁 Flag: <flag_value_here>
```

This worked both via manual input and through automation using a Python subprocess.

Conclusion and Lessons

- Integer comparisons are prone to logic flaws when signed and unsigned types are mixed.
- A solid understanding of two's complement representation can help bypass such logic checks.
- Reverse engineering tools like Ghidra or x64dbg are essential in dissecting Windows binaries to understand logic-based challenges.
- This challenge reinforced the importance of proper input validation and awareness of integer overflows.

References