Antares (Launched 2001)

- *Username:* `antares`
- ▶ *Click to reveal password:*
- *Points:* 10 for code, 5 for writeup

Relevant lectures: 4 - Memory Safety Vulnerabilities II

> **STORY**
> The exchange from Deneb was shocking. You realize the Jupiter orbiter may not be what you once thought it was. You are left with no choice but to dig deeper. Antares is a Gobian targeting satellite that is used to provide midcourse calibrations to the royal guard's anti-spacecraft missiles. Your job is to hack into Antares, obtain the targeting data, and with it, what Gobians knew about the orbiter.

In this question, we're going to walk you through using a format string vulnerability to redirect execution to malicious shellcode.

Step 0: High-Level Overview

Our high-level goal is to redirect execution to our malicious shellcode. We have an arg file, which is loaded into the `argv` parameter of `main`, and an env file, which is piped into standard input.

For this question, we place the shellcode in arg. Your first step is to find the address of this shellcode: do so, and then write that address down - we'll need it later.

Remember, the shellcode itself should start with `0xcd58326a`.

Step 1: Analyze the Code

At what line is the vulnerable `printf` call? Set a breakpoint at the vulnerable function call, and draw a stack diagram up to that point. Below is a template you may use to write out a text-based stack diagram - if you request help during office hours, this is the first thing that we'll want to see!

```
...
0x00000000 [  ][  ][  ][  ] __________
```

```
0x00000000 [  ][  ][  ][  ] RIP of Main
0x00000000 [  ][  ][  ][  ] ___________
...
```

Step 2: Quick Format String Review

A quick reminder about how format string vulnerabilities work: when you have a line of code that looks like `print(buf)`, where we control `buf`, you can pass format string specifiers into the user-provided input. When the CPU sees a format string identifier being used, it expects arguments located in incrementally increasing positions above the zeroth argument to `printf` (`&buf`, denoted `arg0` here), seen here on the stack as `arg1`, `arg2`, etc.

```
...
[  ][  ][  ][  ] <-- arg2
[  ][  ][  ][  ] <-- arg1
[  ][  ][  ][  ] <-- arg0 (&buf)
[  ][  ][  ][  ] <-- RIP of printf
[  ][  ][  ][  ] <-- SFP of printf
...
```

Imagine that `printf` has a pointer that initially points at `arg1`. Every time it sees a format string identifier, it moves that pointer up by four, thus "consuming" the argument located at the original location of the pointer. For example, if we set `buf` to `'%d%d'`, then `printf` would look at `arg1` for the first `'%d'`, and `arg2` for the second `'%d'`. Here are a few important format string specifiers you should be aware of:

Specifier	Description
`%c`	Treats the corresponding `arg` as a VALUE. Print it as a character.
`%<k>u`	Treats the corresponding `arg` as a VALUE. Prints the corresponding `arg` as an unsigned integer and adds whitespace in front to display a total of `k` characters. For example, `printf("%7u\n", 123)` prints " 123", i.e. 4 spaces before 123 and 7 total characters.
`%s`	Treats the corresponding `arg` as a POINTER. Dereference the pointer and print the resulting value as a string.
`%n`	Treats the corresponding `arg` as a POINTER. Write the number of bytes that have been currently printed (as a four-byte number) to the memory address in the corresponding `arg`.

Specifier	Description
`%hn`	Treats the corresponding `arg` as a POINTER. Write the number of bytes that have been currently printed (as a two-byte number) to the memory address in the corresponding `arg`.

We often use specifiers that read values (e.g. `%c`, which reads a char) to "skip" arguments on the stack. Why? Sometimes, we want to work our way up the stack until we reach a place that we have write-access to (e.g. a buffer), so that we can use user-crafted inputs in our format string exploits. As such, we may find ourselves using something like `'%c' * ____`, which will walk up the stack and skip past `arg1`, `arg2`, etc.

Step 3: Analyzing our Write Vector

Ok, so what do we know at this point?

1. We know that (a) we want to redirect execution to shellcode by setting the RIP of `calibrate` to a shellcode address. This is our end goal.
2. We can use our write vector (the `%hn` in `printf`) to write numbers to certain locations at the stack.

That's great...but how do we use such a limited write vector (`'%n'` or `'%hn'`) to write an entire memory address? We could try to convert the memory address to an integer (e.g. `0xDEADBEEF` => `3735928559`) and print that many bytes, and then use `%n` to write that number to the stack. But printing that many bytes would crash the program! Instead, we can break up our write into two halves, and use the `'%hn'` specifier instead to write one half at a time.

For example, if we're trying to write `0xFFFF1234` to `0xFFFF5550`, we can:

1. Write `0x1234` to memory address `0xFFFF5550`, and then...
2. Write `0xFFFF` to memory address `0xFFFF5552`

After these writes, the stack will look like the following:

```
0xFFFF5550 [??][??][??][??] (original)
0xFFFF5550 [34][12][??][??] (after first '%hn' write)
0xFFFF5550 [34][12][FF][FF] (after second '%hn' write)
```

Step 4: Attack

See the comments in the blocks to walk through the attack. Good luck!

Deliverables

- Two scripts, `egg` and `arg`
- A writeup.
