Last week we covered Protostar's Stack4:

Stack4, much like the previous stack0-3, required a buffer overflow to exploit and execute the `win()` function. However employing the same approach as used in stack0-3 caused some trouble at the end of stack4’s execution.

Nonetheless, an understanding of the stack—in particular the stack frame of the main function—is required, so we'll start there:

Run gdb on stack4 and disassemble the main function

*To display Intel syntax (the correct syntax) in gdb "set disassembly-flavor intel"

First, we need to calculate the offset after the base pointer (ebp) is pushed on the stack and moved into the stack pointer (esp):

```plaintext
push ebp:
         mov ebp, esp:
```

This increases the offset. In this case it increases it by 4 bytes. These two instructions are referred to as the function prologue.

```plaintext
and esp, 0xffffffff:
```

This is the stack alignment, which throws away the last digit on the stack pointer address. This accounts for the offset as well (8 bytes), and usually rounds the address to a base 2 number.

```plaintext
sub esp, 0x50:
```

Creates space on the stack frame to store data, increases the offset by 0x50 (80 bytes). Since the stack grows down, subtracting 0x50 from the stack pointer increases the stack’s size.

Remember though, data grows up in memory such that the stack and data grow in opposing directions.
lea eax, [esp+0x10] <--start of buffer:

    Grab address 16bytes above the stack pointer and store it in eax. These first 16bytes of space at the start of the buffer created in the instruction above will be used as parameter storage for an upcoming function call.

mov DWORD ptr [esp], eax:

    Store double word pointer to 16byte address space from eax in esp. This is a pointer to the parameters of the upcoming function call.

Calculating the offset is a crucial aspect of understanding how the stack works and being able to reverse engineer programs.

So in short the offset can be calculated as:

0x50-0x10 = 0x40 = 64bytes bitmask
0xffffffff0 and esp = 8bytes
push ebp = 4 bytes

total offset = 64+8+4 = 76bytes

payload = 76bytes junk + win() addr

Problem is we segfault. This isn’t good:

    Segfaults are usually universally logged on systems and act as a huge red flag for an impending threat.

NOTE*(RoP can be used in bypassing ASLR)

Question, how do we avoid segfault?:

If we imagine our stack, after we overflow 76bytes of junk, and after we write address of win() to main’s ebp we force the program to return to win() when ebp is popped off of main's stack frame.

It's important to know the difference in returning to a function and calling a function. In a function call the eip saves the return address on the stack, but in a
return, eip is not pushed on the stack. Typically there would be no need for this if it were a conventional return.

So here we can see where the segfault comes from. When the win() function tries to return using the eip that would have typically been pushed on its stack in a call, all it gets is 0x00000000.. This is not a valid address to return to.

Lets try adding a call to exit after our call to win:

This will circumvent the problem of not being able to return to main. Adding a libc function call in our payload will allow eip to be saved on the stack.

payload = 76*'A' + addr to win:

addr to exit():

When making the payload, it is important in this case to have the address of win() come before the address of exit(). In gdb you can type "print exit" to get the address of a function that is loaded into libc.

Stack5:

The Goal is to execute some ret2libc attack on stack5. This can be done similarly to what we did in stack4; calling exit() or getting root and popping a shell.

Answer:

```
python -c "print 'A'*76 + '\xc0\x60\xec\xb7'" 1 ./stack5
```

76 byte junk + exit() address

A ret2libc (return to libc, or return to the C library) attack is one in which the attacker does not require any shellcode to take control of the target, vulnerable process:

It's been nice so far using protostar, which has no protection and is great theory practice but it is not the whole picture by any means.
ret2libc is used to bypass the DEP (non-executable stack) protection:

If you put your executable code in the stack it will not be executed.

The basic idea is that most C programs will be compiled with libc and thus you'll have access to these functions (printf, gets, system, exit, etc):

Instead of relying on our shellcode we can use the functions already loaded by libc.

Even though stack/heap is nx (non-executable stack), libc programs loaded in memory can be used, thus our stack4 example was a version of ret2libc:

One interesting thing that ret2libc can do is actually execute a shell. The payload would look as such:

junk to overwrite ebp+address to system() + 4bytes junk or address to exit() + address to "/bin/sh". Of course "/bin/sh" has to be loaded in memory somewhere:

System() is a libc function, which can be used to pop a shell. The addr of /bin/sh will be loaded in memory somewhere by libc. You might be able to find the addr with gdb.

Execute a shell (note: there are multiple ways to do this)

For purposes of popping a shell, you can use an environment variable address to load "/bin/sh" and search for it.

This is apposed to /bin/bash which does not actually give root access. bash also logs information where as sh does not.

A helpful command to search for env variable addresses in gdb is x/32s*environ

Below is a good video tutorial on better understanding x86 architecture and how stack frames and registers work:

https://www.youtube.com/watch?v=H4Z0S9ZbC0g&list=PL038BE01D3BAEFDB0&index=1