CSC648 Project 2

Shuowen Wei

1.

1.1 Firstly, set up a break point in the called function, and compute the difference of eip value (the return address) and buffer address value, then get the return address value ret by adding this difference to buffer’s address

Secondly, compute the difference of the addresses between the instruction we want to skip (here, it’s “x=1;”) and its next instruction (here, “printf(…)”).

ret = (long \*)(buffer + 21); // the difference between buffer and eip is 21

\*ret += 8;// 8 is the difference from instruction “x=1” to “printf…”

1.2 The stack of this function is:

Buffer(0xbffff787), Ebp (0xbffff798), eip (0xbffff79c)

1.3 Use list in dbg to find the lines where the next two instructions are, here, “x=1;” is on line 26 and “printf(…);” is on line 27, then use info line XX to find their rip values.

(gdb) info line 26

Line 26 of "skipLine.c" starts at address **0x80484e5** <main+45>

and ends at 0x80484ed <main+53>.

(gdb) info line 27

Line 27 of "skipLine.c" starts at address **0x80484ed** <main+53>

and ends at 0x8048501 <main+73>.

1.4 No, it doesn’t prevent my program from working. Because we just modify the return address of the called function to skip the next instruction, we don’t need the execute any codes in the stack of the called function, so with or without the executable stack flag on, it donesn’t matter.

2.

2.1 First, use the same way to get the value for ret, the return address, then simply set the ret to the address of the shell code, included in the header file.

ret = (long \*)(buffer+21);// same with last question 1.1, the address of return (main function)

(\*ret) =(long)amdshellcode; // set return to address of shell code

2.2 the stack of the called function is:

Buffer(0xbffff787), Ebp(0xbffff798), eip (0xbffff79c)

2.3 Just add a line:

printf("0x%x\n",amdshellcode);

Then the rip value is printed out as: 0x80497c0

2.4 No, we didn’t put the shell code in the stack of the called function, so with stack executable or not, it doesn’t matter.

3.

3.1 since strcpy doesn’t check the boundaries of strings, thus by copying a larger string into a smaller buffer, and the larger string is filled with the address of the buffer, until overwrite the return value ret with the address of buffer (then the ret points to the beginning of the buffer), then put the shell code at the beginning of the buffer, then the ret use the address of buffer to return to the beginning of the buffer, then to execute the shell code located there.

#define BUFFER\_SIZE 128 // should at least be greater than 116

…

for(int i=0; i< (BUFFER\_SIZE)/4 ; i++)

\*(longPtr + i) = (int) buffer; // fill the ret value with the address of buffer

for(int i=0; i < strlen(amdshellcode); i++)

largeString[i] = amdshellcode[i]; // put the shell code at the beginning of the string

3.2 the stack of the called function;

Buffer (0xbffff750), ebp (0xbffff7b8), eip (0xbffff7bc)

3.3 The size of the largeString array should be larger than the sum of the size of ret (which is 8bytes) and the difference of addresses of the buffer and the ret (which is 7bc-750=108). (But we don’t know why 116 and 120 doesn’t work)

3.4 No, my program shows that turn on and off the executable stack flag doesn’t affect. I want to say “yes” since it’s supposed to prevent my program from working, but I don’t know why, because with the executable stack flag off, my program can still execute the shell code in the stack.

4.

4.1 First, we compute the difference of the addresses of the buffer and the ret in vulnerable.c, then add 8 and set it to the size of the buffer in exploitBuffer.c (also is the size of the generated badfile), initialize the buffer in the exploitBuffer.c with NO-OP, then put the shell code at the beginning of the buffer, in the end put the address of the buffer (in vulnerable.c file) at the very end of the buffer (of the exploitBuffer.c), whose location is corresponding to the location of ret in vulnerable.c. In the end, write this buffer into a badfile, to be supplied for vulnerable.c.

#define DEFAULT\_BUFFER\_SIZE 532

………

long \*ptr;

ptr = (long \*)((long)buff+524);//point the very end of the buffer, which is corresponding to where ret locates in the buffer of vulnerable.c

\*ptr = **0xbffff1a0**; // this is the address of the buffer in vulnerable.c, put it at the end of the badfile

for(int i=0; i < strlen(amdshellcode); i++)

buff[i]=amdshellcode[i]; //copy and put the shell code at the beginning of the buffer

…….

4.2 the stack of the called function is:

Buffer (**0xbffff1a0**), ebp (0xbffff398), eip (0xbffff39c)

4.3 compute the difference of addresses between buffer and ret in vulnerable.c, then add 8, which is the size of the ret, that’s the size of the exploit array.

4.4 No, same things happened with 3.4, that my program shows that turn on and off the executable stack flag doesn’t affect.

5.

You told me don’t do question 5.