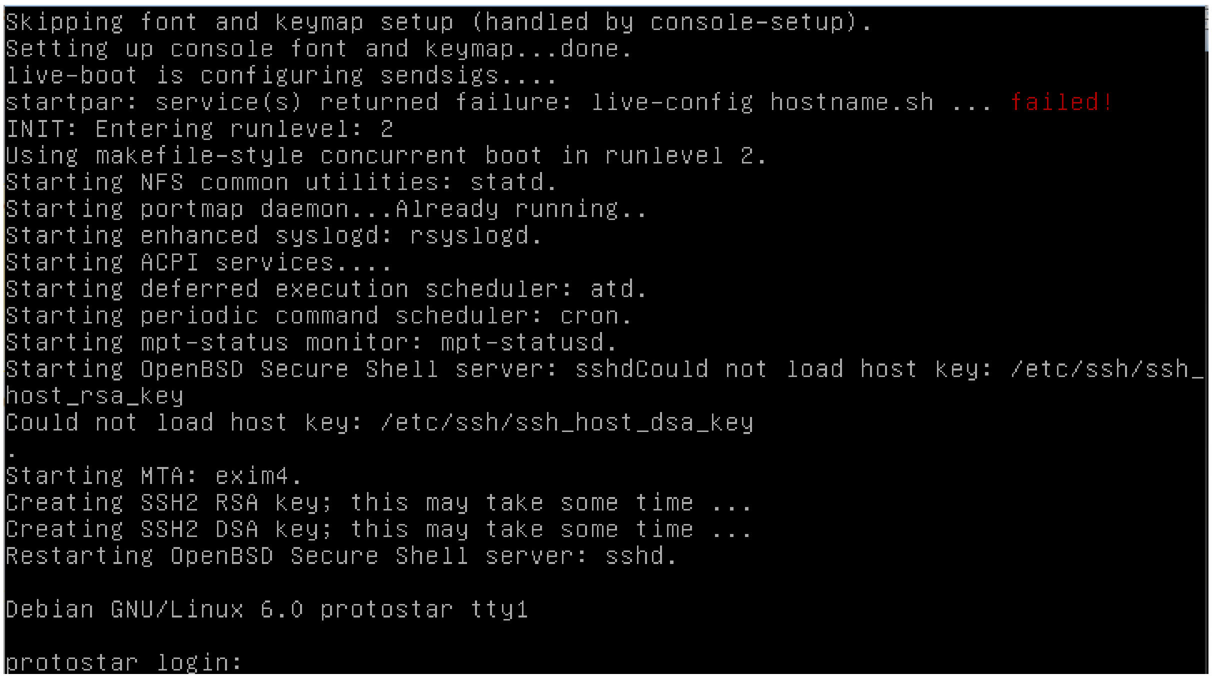
COMP8053 – Embedded Software Security

Lab 4 – Introduction to Buffer Overflows

- Download the “Protostar” ISO from Canvas.
- Open VMWare directly.
- Right-click on Home and select “Create a new VM…”
- Select “Installer disc image file (iso):” and select the protostar .iso you downloaded. Continue through the installer, and when complete, start the Protostar VM image you have created.
- When starting the VM select “Live” to continue booting up.
- No changes you make to the VM will be saved when you close it! If you wish to retain any files you made, you must save them externally to the protostar VM.

The protostar VM contains a very old operating system lacking many of the modern security protections that exist in PCs these days. It can be taken as being representative of an embedded device’s OS due to resource limitations and minimizing costs. All going well, you will be greeted with a screen something like this:



```
Skipping font and keymap setup (handled by console-setup).
Setting up console font and keymap...done.
live-boot is configuring sendsigs....
startpar: service(s) returned failure: live-config hostname.sh ... failed!
INIT: Entering runlevel: 2
Using makefile-style concurrent boot in runlevel 2.
Starting NFS common utilities: statd.
Starting portmap daemon...Already running..
Starting enhanced syslogd: rsyslogd.
Starting ACPI services....
Starting deferred execution scheduler: atd.
Starting periodic command scheduler: cron.
Starting mpt-status monitor: mpt-statusd.
Starting OpenBSD Secure Shell server: sshdCould not load host key: /etc/ssh/ssh_
host_rsa_key
Could not load host key: /etc/ssh/ssh_host_dsa_key
.
Starting MTA: exim4.
Creating SSH2 RSA key; this may take some time ...
Creating SSH2 DSA key; this may take some time ...
Restarting OpenBSD Secure Shell server: sshd.

Debian GNU/Linux 6.0 protostar tty1

protostar login:
```

Login: user
Password: user

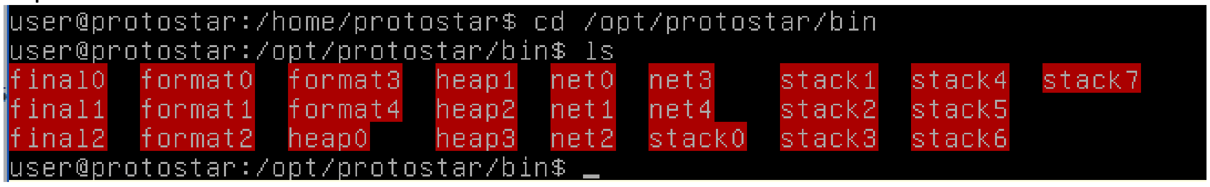
- Recommended: Enter the command ‘bash’ to get a better shell (that allows tab-completion).
- Since it can be useful to have multiple windows open at once for this (and the basic protostar VM does not allow it) we can SSH into the VM to have another window

open. On Windows, use the PuTTY SSH client which is installed in the labs to connect. You can download it here for your own machines: https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html Get the IP address of the VM by typing "ip addr" to get the eth0 ip address (use the default port), then enter those into the SSH client to connect.

(On Linux (and I believe Mac), type: ssh user@<ip address>)

- Change to the directory /opt/protostar/bin/
- (Note that you do not have permissions to create files in this directory, if you want to write to a file, do it in your home directory)

This directory contains the exercises we shall be doing to learn about buffer overflow exploits.



```
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 note that all of the binaries are shown in red, this is because they are all setuid executables created by root.
We will start with the file 'stack0', for which the source code is:

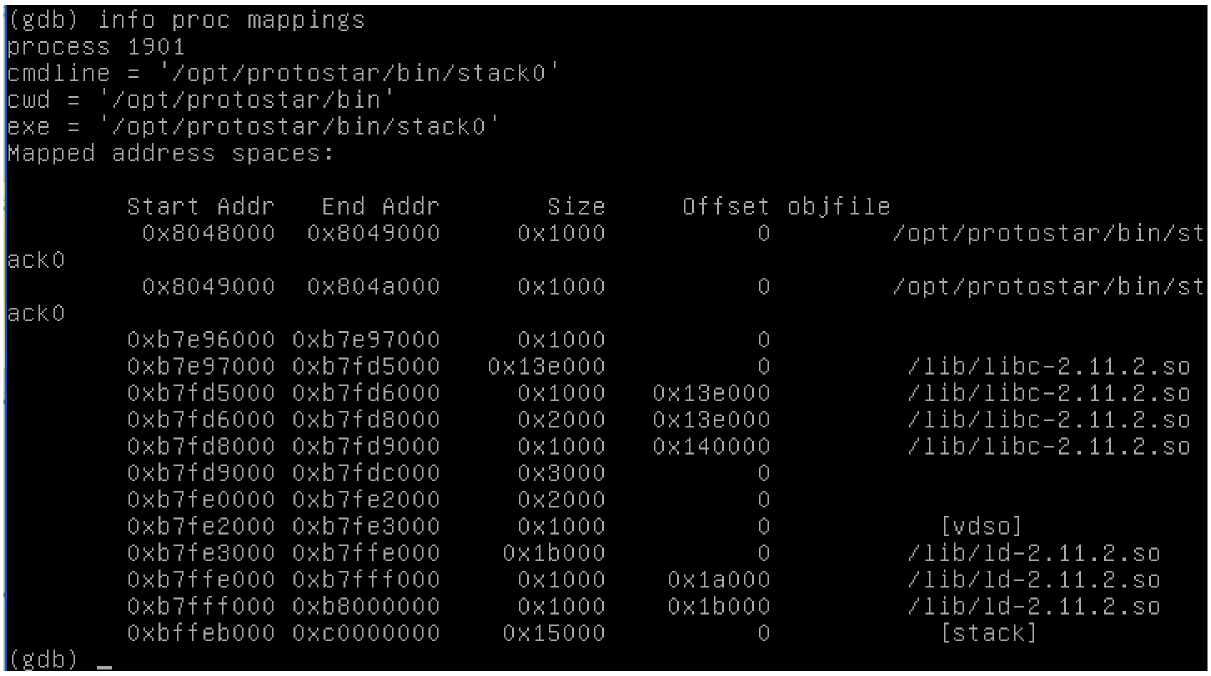
```
1.  #include <stdlib.h>
2.  #include <unistd.h>
3.  #include <stdio.h>
4.
5.  int main(int argc, char **argv)
6.  {
7.    volatile int modified;
8.    char buffer[64];
9.
10.   modified = 0;
11.   gets(buffer);
12.
13.   if(modified != 0) {
14.     printf("you have changed the 'modified' variable\n");
15.   } else {
16.     printf("Try again?\n");
17.   }
18. }
```

In this lab, we will seek to redirect the flow of the program code to run line 14 and output "you have changed the 'modified' variable". Throughout we will keep a close eye on the contents of the stack and registers so we know what is happening at each step. The process we will go to to achieve our goal is:

1. Identify where the value of the variable "modified" is stored on the stack.
2. Set breakpoints and a hookstop to let us easily see the contents of the stack as the program runs.
3. Use a buffer overflow to overwrite values on the stack, until the address storing "modified" has been overwritten.
   - To do this we will use a simple Python script to generate input for us.

After this, you will be challenged to overwrite "modified" with a specific value. This will require you to understand exactly what part of your input string was overwriting which part of the stack.

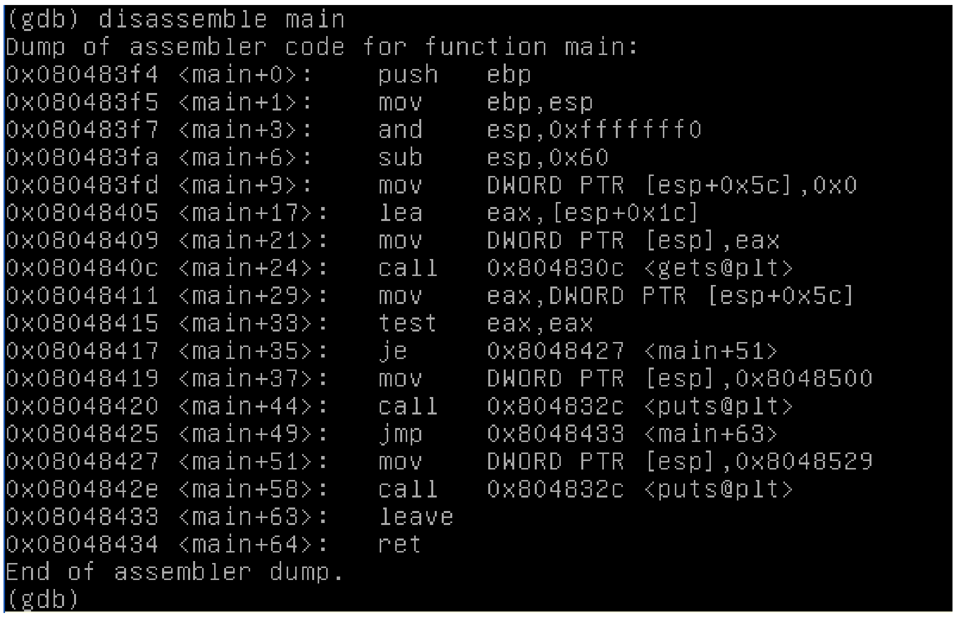
- Open 'stack0' in gdb.
- set disassembly-flavor intel
- enter "break *main" to add a breakpoint when the main function is called
- enter "run" or "r" to run the program until the breakpoint
- enter "info proc mappings" shows us the mapped memory of the system while the program is running (it is still running due to the break point we added which it is waiting at). If you run the program multiple times, you can see that the address of the stack does not change i.e. there is no ASLR in this VM.


```
(gdb) info proc mappings
process 1901
cmdline = '/opt/protostar/bin/stack0'
cwd = '/opt/protostar/bin'
exe = '/opt/protostar/bin/stack0'
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000          0        /opt/protostar/bin/st
ack0
         0x8049000  0x804a000     0x1000          0        /opt/protostar/bin/st
ack0
        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
        0xb7fe0000 0xb7fe2000     0x2000          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]
(gdb) _
```

We can see here that the Stack is stored in addresses 0xbffeb000 to 0xc0000000. The stack actually grows from the bottom, so it starts from 8bytes above 0xc0000000 -> 0xbffffff8.

- Examine the main function by entering: disassemble main

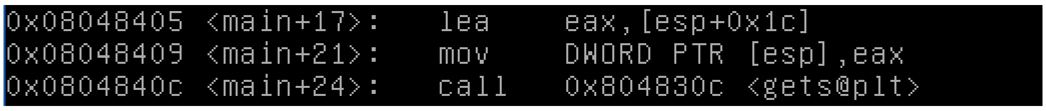


```
(gdb) disassemble main
Dump of assembler code for function main:
0x080483f4 <main+0>:    push   ebp
0x080483f5 <main+1>:    mov    ebp,esp
0x080483f7 <main+3>:    and    esp,0xfffffff0
0x080483fa <main+6>:    sub    esp,0x60
0x080483fd <main+9>:    mov    DWORD PTR [esp+0x5c],0x0
0x08048405 <main+17>:   lea    eax,[esp+0x1c]
0x08048409 <main+21>:   mov    DWORD PTR [esp],eax
0x0804840c <main+24>:   call   0x804830c <gets@plt>
0x08048411 <main+29>:   mov    eax,DWORD PTR [esp+0x5c]
0x08048415 <main+33>:   test   eax,eax
0x08048417 <main+35>:   je     0x8048427 <main+51>
0x08048419 <main+37>:   mov    DWORD PTR [esp],0x8048500
0x08048420 <main+44>:   call   0x804832c <puts@plt>
0x08048425 <main+49>:   jmp    0x8048433 <main+63>
0x08048427 <main+51>:   mov    DWORD PTR [esp],0x8048529
0x0804842e <main+58>:   call   0x804832c <puts@plt>
0x08048433 <main+63>:   leave
0x08048434 <main+64>:   ret
End of assembler dump.
(gdb)
```

We see that as expected the first thing it does is update the values of ebp and esp for the new stack frame.

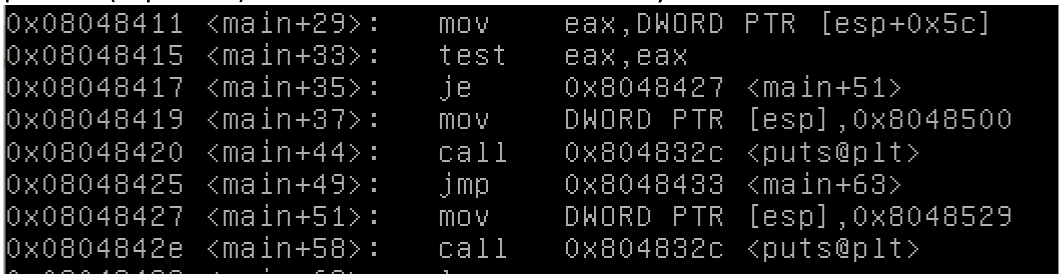
There is one extra line which is: 'and esp, 0xfffffff0' which does a bitwise AND between esp and the value.. effectively setting the last 4 bits to 0 (i.e. masking it). This is done for alignment purposes and not too important for us.

'mov DWORD PTR [esp+0x5c], 0x0' is where the source code line of 'modified = 0' is being executed. The value 0 is stored at a location offset from esp (the top of the stack frame) by 0x5c addresses (which is actually just above where ebp is currently stored).



```
0x08048405 <main+17>:   lea    eax,[esp+0x1c]
0x08048409 <main+21>:   mov    DWORD PTR [esp],eax
0x0804840c <main+24>:   call   0x804830c <gets@plt>
```

'lea' (Load Effective Address) is similar to move, but instead of storing the value of the right argument into the left, it stores the address of the right argument into the left (a register 'eax' in this case). We then move that address, stored in temp register eax, into esp (i.e. the top of the stack). This is because arguments to a function get pushed on top of the stack before adding the stack frame for the called function on top of that. In this case, the address pushed (esp+0x1c) is the address for our char array 'buffer'.



```
0x08048411 <main+29>:   mov    eax,DWORD PTR [esp+0x5c]
0x08048415 <main+33>:   test   eax,eax
0x08048417 <main+35>:   je     0x8048427 <main+51>
0x08048419 <main+37>:   mov    DWORD PTR [esp],0x8048500
0x08048420 <main+44>:   call   0x804832c <puts@plt>
0x08048425 <main+49>:   jmp    0x8048433 <main+63>
0x08048427 <main+51>:   mov    DWORD PTR [esp],0x8048529
0x0804842e <main+58>:   call   0x804832c <puts@plt>
```

Next the value of 'modified' is moved into temp register eax and the 'test' instruction is used to check if the value is 0 (i.e. the 'if' statement in the source). This sets some flags and then either depending on the result of the test, either it will jump to address main+51 or continue to main+37 and call the 'puts' function to print the appropriate output to the screen (the strings we can assume are stored in the addresses visible, and moved to the top of the stack before calling the function, as for 'gets' previously).

Recall some of the gbd commands we learned last week:

- c – continue the currently executing program (after you hit a breakpoint)
- del - deletes all breakpoints
- break *<hex address> - sets a breakpoint at the given memory address (does not need to be a function name)
- define hook-stop - allows you to set a group of commands that will automatically run when a breakpoint is reached. Enter 'end' once you have finished entering commands to run on breakpoints.
- info registers - shows the content of the registers
- x <hex-address or $register-name> – examine the contents of an address
    - It supports extra options.
    - x/<num>wx - shows the next <num> addresses from the location you passed
    - x/<num>i $eip - shows the next <num> instructions from the address of the instruction pointer eip (i.e. the next instructions that will be run next).

Delete all breakpoints, and then add a hook-stop with the commands:

```
define hook-stop
info registers                  #show us what's in the registers
x/24wx $esp                     #show us the contents of the stack (24 addresses)
x/2i $eip                       #next 2 instructions
end
```

1. We will now look at the call to "gets", so set breakpoints at the call to "gets", and on the instruction immediately after the call.
2. Continue execution of the program, which should make you hit the breakpoint on the call to 'gets'.
3. You should be able to see the content of the registers, the stack and the next instruction should be the call to 'gets' if you set the breakpoint correctly.
4. We can check the value of the 'modified' variable by examining its address in memory:
    - x $esp+0x5c   #recall this is where the value 0x0 was moved to earlier
5. We note that this address is 0xbffff7cc which our current view of the stack shows (it is the final/24th block visible). The content is current 0x0000000 and this is what we wish to modify.
6. Type "c" to continue to the 2nd breakpoint after "gets".
7. Since the call to "gets" requires user input, type in a string of 10-20 "A"s (i.e. AAAAAAAAAAAAAAAAAAAA).
8. We then hit the 2nd breakpoint and can view the stack again. You should now see the hex value "41" multiple times in the stack, this is your string of "A"s as stored in hexadecimal.

9. Try re-running the program (“r” again, don’t exit gbd) and entering in a long enough sequence of “A”s to modify memory address ‘0xbffff7cc’.
10. Success!

Next, let’s close gdb (command: q) and try achieving it outside the debug environment. Typing out a ton of “A”s is quite tedious, and we can use python to simplify it! We can generate the “A”s using a python script and feed it as input to the executable as follows:

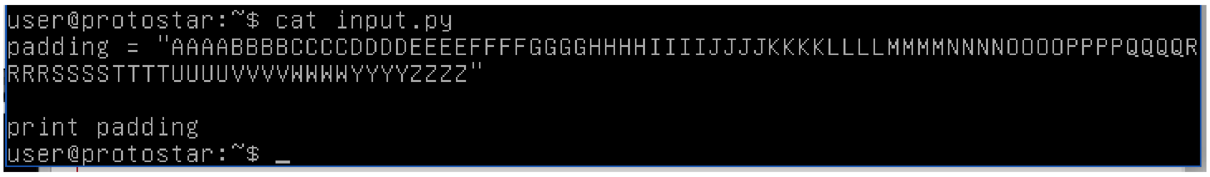
python –c ‘print (“A”*40)+”B” ’ | ./stack0

This would give it an input string of 40 As followed by a single B.

This may get a bit unwieldy, so it may be simpler to take the below approach:

*******Note you cannot write to the /opt/protostar/bin directory, so create your input files in your home directory******

Write a python program in your home directory (e.g. input.py), which prints out your desired input string. The format for a python program which stores a string as a variable “padding” and then prints out that string is shown below:



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

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

Feed that output into a file:
python input.py > infile

Then run the executable with that as input:
/opt/protostar/bin/<executable_name> < infile

Or if inside GDB:
r < infile

- Instead of generating all “A”s, it can be useful to generate a recognisable pattern like “AAAABBBBCCCCDDDD…” etc. so that you can see in memory which block of 4 chars was overwritten where, since each letter has its own hexcode. Listing of the hexcodes for each character: http://www.asciitable.com

Next try the next problem, stack1, on your own with the knowledge you've gained!

Source code of stack1:

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

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

  if(argc == 1) {
    errx(1, "please specify an argument\n");
  }

  modified = 0;
  strcpy(buffer, argv[1]);

  if(modified == 0x61626364) {
    printf("you have correctly got the variable to the right value\n");
  } else {
    printf("Try again, you got 0x%08x\n", modified);
  }
}
```

In this problem, rather than just overwriting a value with anything like in stack0, you must instead replace it with a specific hex value (see the source code..).

Note that this problem requires input as a command line argument, instead of as user input for "gets()". You can enter Python code as input as follows:

./stack1 `python –c "print 'A'*40 " ` #note that the outer quotation marks are the key to the left of the "1" on your keyboard, not a regular single quote.

The above code would generate a string of 40 A's and pass it as the argument to the executable. It is probably more convenient to write a python script that generates an input file again and feed that as input to the program instead.

To pass the contents of a file as a command line argument, you can use it as:

/opt/protostar/bin/stack1 $(cat input) #where 'input' is your file's name

Next try the problem, stack3! (we skip stack2 as it's very similar to the previous ones)

Source code of stack3:

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

void win()
{
  printf("code flow successfully changed\n");
}

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

  fp = 0;

  gets(buffer);

  if(fp) {
    printf("calling function pointer, jumping to 0x%08x\n",
             fp);
    fp();
 }
}
```

In this problem, "fp" is a function pointer (i.e. the address of a function). When it calls "fp()" it will run the function at the address stored in "fp". You want to run the function "win" by changing the value of variable 'fp' to the address of the function "win", using a buffer overflow. To find the location of function "win" in memory, in GDB recall that you can type either:

```
x win
p win
```

To "examine" or "print" 'win' respectively.

Recall that to write bytes in a string, including those which cannot be represented in the ASCII format, we can write them in the format "\xAB" which is interpreted by Python as the byte 0xAB.

Finally try the problem stack4!

Source code of stack4:

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

void win()
{
  printf("code flow successfully changed\n");
}

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

  gets(buffer);
}
```

In this problem, you want to run the function “win” but this time there is no function call for us to overwrite like there was in stack3. Here you must instead overwrite the return address stored for the ‘main’ function on the stack. Recall that due to the push/mov etc.. at the start of a function, the ebp value is just above this return value on the stack.

1. Identify the location of the return address on the stack.
2. Identify the address of the function “win”.
3. Replace the contents of the return address with the address of “win”.