Tired of being thwarted by meddling 105 students, Dr. Evil tracks down an unsuspecting student who has put off taking CS 105 and convinces them to run the following program.

Consider the following C program and the corresponding machine code:

```c
#include <stdio.h>

int isPosInt(char * s){
    char * p = s;
    while(*s != '\n'){
        if(*s < 48 || *s > 57)
            return 0;
        s++;
    }
    return 1;
}

void getPosInt(char * s, int n){
    int done = 0;
    while(!done){
        gets(s, stdin);
        done = isPosInt(s);
    }
    return 1;
}

int main(int argc, char ** argv){
    int MAX_LEN = 12;
    char buf[MAX_LEN];
    getPosInt(&buf, MAX_LEN);
    printf("%s\n", buf);
}
```

```assembly
0x4005fc <main>:
  sub $0x18,%rsp
  mov $0xc,%esi
  mov %rdi,%rdx
  callq 0x4005c6 <getPosInt>
  mov %rdi,%rdx
  callq 0x400470 <puts@plt>
  mov $0x0,%eax
  add $0x18,%rsp
  retq

0x4005c6 <getPosInt>:
  push %rbp
  push %rbx
  sub $0x8,%rsp
  mov %rdi,%ebp
  mov $0x0,%eax
  jmp 0x4005f1 <getPosInt+43>
  mov 0x200a61(%rip),%rsi
  add $0x0,%eax
  jmp 0x4005f1 <getPosInt+43>
  mov %rbx,%rdi
  callq 0x4005a6 <isPosInt>
  test %eax,%eax
  add $0x8,%rdx
  retq
```

Consider the following C program and the corresponding machine code:

```c
int isPosInt(char * s){
    char * p = s;
    while(*s != '\n'){
        if(*s < 48 || *s > 57)
            return 0;
        s++;
    }
    return 1;
}

void getPosInt(char * s, int n){
    int done = 0;
    while(!done){
        gets(s, stdin);
        done = isPosInt(s);
    }
    return 1;
}

int main(int argc, char ** argv){
    int MAX_LEN = 12;
    char buf[MAX_LEN];
    getPosInt(&buf, MAX_LEN);
    printf("%s\n", buf);
}
```

```assembly
0x4005fc <main>:
  sub $0x18,%rsp
  mov $0xc,%esi
  mov %rdi,%rdx
  callq 0x4005c6 <getPosInt>
  mov %rdi,%rdx
  callq 0x400470 <puts@plt>
  mov $0x0,%eax
  add $0x18,%rsp
  retq
```

0x4005c6 <getPosInt>:
  push %rbp
  push %rbx
  sub $0x8,%rsp
  mov %rdi,%ebp
  mov $0x0,%eax
  jmp 0x4005f1 <getPosInt+43>
  mov 0x200a61(%rip),%rsi
  add $0x0,%eax
  jmp 0x4005f1 <getPosInt+43>
  mov %rbx,%rdi
  callq 0x4005a6 <isPosInt>
  test %eax,%eax
  add $0x8,%rdx
  retq
```

0x4005a6 <isPosInt>:
  // more assembly code
1. Below is a diagram of the stack at the beginning of function `main` (that is, when `%rip` = `0x4005fc`).

(a) Draw a detailed diagram of the stack immediately before the function `gets` is called (that is, when `%rip = 0x4005e4`). If you cannot determine from the provided information what value is stored at some address, enter a ? in the corresponding box. Assume that the initial value in register `%rbp` is 0. Assume that initial value in register `%rbx` is `0x400620`.

(b) Add arrows to the above diagram to show the current values stored in `%rsp` and `%rdi`

2. Assume that Dr. Evil has somehow included an evil function located in memory at address `0x406147`. Construct an example exploit string that would cause the evil function to get executed after `main` returns. Assume the machine is little endian.
3. Maybe Dr. Evil was unable to include his evil function in the code. Assume that he instead enters a carefully constructed exploit string so that at the point immediately before main returns, the state of the stack is shown below.

<table>
<thead>
<tr>
<th>Address</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7FFFFFFFEAB10</td>
<td>0a 00 00 00 00</td>
</tr>
<tr>
<td>0x7FFFFFFFEAB08</td>
<td>1e 06 40 00 00</td>
</tr>
<tr>
<td>0x7FFFFFFFEAB00</td>
<td>47 47 47 47 47</td>
</tr>
<tr>
<td>0x7FFFFFFFEAF8</td>
<td>47 47 47 47 47</td>
</tr>
<tr>
<td>0x7FFFFFFFEAF0</td>
<td>0d 06 40 00 00</td>
</tr>
<tr>
<td>0x7FFFFFFFEAE8</td>
<td>61 68 61 68 61</td>
</tr>
<tr>
<td>0x7FFFFFFFEAE0</td>
<td>61 68 61 68 61</td>
</tr>
<tr>
<td>0x7FFFFFFFEAD8</td>
<td>47 47 47 47 47</td>
</tr>
<tr>
<td>0x7FFFFFFFEAD0</td>
<td>10 06 40 00 00</td>
</tr>
<tr>
<td>0x7FFFFFFFEAC8</td>
<td>dc ea ff ff ff</td>
</tr>
<tr>
<td>0x7FFFFFFFEAC0</td>
<td>2a 04 40 00 00</td>
</tr>
<tr>
<td>0x7FFFFFFFEAB8</td>
<td>2a 04 40 00 00</td>
</tr>
</tbody>
</table>

You should interpret the sequence of bytes in each box as as the hex-encoding of the eight byte sequence starting at the address labeled at the bottom of the box and ending one byte before the address labeled at the top of the box. So, for example, the byte at address 0x7FFFFFFFEAB8 is 2a and the byte at 0x7FFFFFFFEABF is 00.

**Hint:** You may assume the Pomona server is a little-Endian machine.

**Hint:** Observe that the address in %rsp immediately before main returns will be 0x7FFFFFFFEAB8.

Assume that the byte at address 0x40042a is 0x5f (the byte-level encoding of pop %rdi) and the byte at address 0x40042b is 0xc3 (the byte-level encoding of ret). A table of potentially useful ASCII encoding is given below.

<table>
<thead>
<tr>
<th>0a</th>
<th>21</th>
<th>42</th>
<th>4d</th>
<th>61</th>
<th>68</th>
<th>6d</th>
<th>6f</th>
<th>77</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>!</td>
<td>B</td>
<td>M</td>
<td>a</td>
<td>h</td>
<td>m</td>
<td>o</td>
<td>w</td>
</tr>
</tbody>
</table>

5-3
(a) Fill in the table below with the values in each of the following registers when %rip stores each of the values. Each line of the table should correspond to one assembly instruction (so line 1 will describe the state of the registers after the instruction `retq` from line 0 completes, line 2 will describe the state of the registers after the instruction from line one completes, etc.) The initial line (immediately before the main function returns) has been filled out to help you get started. Treat any function calls as one instruction (i.e., “step over” them same as `nexti` would in `gdb`).

**Hint:** Remember that %rip stores the address of the next instruction to execute.

**Hint:** For addresses on the stack, it’s fine to just use the last two bytes (e.g., `eab8` instead of `0x7fffffffeab8`).

<table>
<thead>
<tr>
<th></th>
<th>%rip</th>
<th>(%rip)</th>
<th>%rsp</th>
<th>%rdi</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x40061e</td>
<td><code>retq</code></td>
<td>0x7fffffffeab8</td>
<td>0x7fffffffeab8</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) What gets printed after the main function returns?

**Hint:** `puts` prints the string passed in as its first argument.