Crash Course in C, x86 Assembler, and GCC Compilation

This revised handout fixes some bugs in the earlier handout and adds some new material.

Read The Following Papers

Aleph One, "Smashing the Stack for Fun and Profit" (can be found at `http://cs.wellesley.edu/~cs342/stack-smashing.txt`).

scut/team teso, "Exploiting Format String Vulnerabilities" (can be found at `http://cs.wellesley.edu/~security/papers/formatstring/formatstring-1.2.pdf`).

Overview

Our next topic is software vulnerabilities like buffer overflow attacks and format string exploits. We need to know a lot of low-level details in order to understand/launch such exploits (e.g., reading the two papers listed above):

- Ability to read high-level programs that might containing buffer overflow vulnerabilities (typically C/C++ programs).
- Understanding conventions used by compiler to translate high-level programs to low-level assembly code (in our case, using Gnu C Compiler (gcc) to compile C programs).
- Ability to read low-level assembly code (in our case, Intel x86).
- Understanding how assembly code instructions are represented as machine code.

We will learn these details in the context of some examples spread over two handouts. So you'll be getting a crash course in C programming, Intex x86 assembly code, and compilation.

A Sum-of-Squares (SOS) Program in C

```
/* Contents of the file sos.c */

/* Calculates the square of integer X */
int sq (int x) {
  return x*x;
}

/* Calculates the sum of squares of a integers Y and Z */
int sos (int y, int z) {
  return sq(y) + sq(z);
}

/* Reads two integer inputs from command line
   and displays result of SOS program */
int main (int argn, char** argv) {
  int a = atoi(argv[1]);
  int b = atoi(argv[2]);
  printf("sos(%i,%i)=%i\n", a, b, sos(a,b));
}
```

Notes:

- The program (assumed to be in `sos.c`) is compiled and executed as follows:

```
[cs342@puma] gcc -o sos sos.c

[cs342@puma] sos 3 4
sos(3,4)=25

[cs342@puma] sos -9 -10
sos(-9,-10)=181

[cs342@puma] sos foo bar
sos(0,0)=0

[cs342@puma] sos 3.1 4.9
sos(3,4)=25

[cs342@puma] sos 3foo -10bar
sos(3,-10)=109
```

- The `sq` and `sos` functions almost have exactly the same syntax as Java class methods (except for omission of the `static` keyword, which means something different in C).

- The entry point to the program is the `main` function.

  - `argn` is the number of command-line arguments, which are indexed from 0 to `argn` − 1. Argument 0 is the command name. E.g., for `sos 3 4`, `argn` is 3.
  - `argv` is an array of strings holding the command-line arguments. E.g., in `sos 3.1 4.9`:

    ```
    argv[0] = "sos"
    argv[1] = "3.1"
    argv[2] = "4.9"
    ```

  - In C, arrays of elements of type T are represented as pointers to the 0th element of the array. E.g., `char*` is a pointer to a character, which is the representation of an array of characters (i.e, a string). `char**` is a pointer to a pointer to a character, which can be an array of strings.
  - Note that `main` has type `int`. C programs return an integer *exit status*. A program that executes without error returns 0. A program that encounters an error returns an integer error code > 0.

- The `atoi` function converts a string representation of an integer to the integer. If the string does not denote an integer, but has a prefix that does, `atoi` returns the the integer of the prefix. It returns 0 if the string can't be interpreted as an integer at all.

```
atoi("42")       42
atoi("-273")     -273
atoi("123go")    123
atoi("12.345")   12
atoi("12.345")   12
atoi("foo")      0
```

- The `printf` function is the typical means of displaying output on the console. Consider:

```
printf("sos(%i,%i)=%i\n", a, b, sos(a,b));
```

  The first argument, `"sos(%i,%i)=%i\n"`, is the *format string*, which contains three "holes" indicated by the output specifiers `%i`, which means "an integer goes here". (We will see other output specifiers later. *Note:* `%d` is a synonym for `%i`.)

  The remaining arguments, in this case `a`, `b`, `sos(a,b)`, are expressions denoting the values to fill the holes of the output specifiers.

C Types and Their Representations

We can learn about C value representations and `printf` via the following example:

```
int main (int argn, char** argv) {
  int i; /* uninitialized integer variable */
  int j = 42;
  int k = -1;
  int a[3] = {17,342,-273};
  float f = 1234.5678;
  int* p = &j; /* &a denotes the address of a in memory. */
  char* s = "abcdefg";

  /*********************************************************************/
  /* Typical things we expect to do: */
  printf("----------------------------------------------------------\n");
  printf("i = %i (signed int); %u (unsigned int); %x (hex);\n\n", i, i, i);

  printf("j = %i (signed int); %u (unsigned int); %x (hex);\n\n", j, j, j);

  printf("k = %i (signed int); %u (unsigned int); %x (hex);\n\n", k, k, k);

  for (i=0; i<3; i++) {
    printf("a[%i] = %i (signed int); %u (unsigned int); %x (hex);\n", i, a[i], a[i], a[i]);
  }
  printf("i = %i (signed int); %u (unsigned int); %x (hex);\n\n", i, i, i);

  printf("f = %f (floating point); %e (scientific notation);\n\n", f, f);

  /* p denotes the address of an integer variable; *p denotes its contents */
  printf("p = %u (unsigned int); %x (hex);\n", p, p);
  printf("*p = %i (signed int); %u (unsigned int); %x (hex);\n\n", *p, *p, *p);

  /* s denotes the address of an char/string variable; *s denotes its contents */
  printf("s = %u (unsigned int); %x (hex); %s (string);\n", s, s, s);
  printf("*s = %c (char);\n\n", *s);

  /* More printf statements will be added here later */
}
```

Let's compile this and study the result of executing it:

```
[cs342@puma] gcc -o reps reps.c

[cs342@puma] reps
------------------------------------------------------------
i = 3996344 (signed int); 3996344 (unsigned int); 3cfab8 (hex);

j = 42 (signed int); 42 (unsigned int); 2a (hex);

k = -1 (signed int); 4294967295 (unsigned int); ffffffff (hex);

a[0] = 17 (signed int); 17 (unsigned int); 11 (hex);
a[1] = 342 (signed int); 342 (unsigned int); 156 (hex);
a[2] = -273 (signed int); 4294967023 (unsigned int); fffffeef (hex);
i = 3 (signed int); 3 (unsigned int); 3 (hex);

f = 1234.567749 (floating point); 1.234568e+03 (scientific notation);

p = 3221208936 (unsigned int); bfffbf68 (hex);
*p = 42 (signed int); 42 (unsigned int); 2a (hex);

s = 134514788 (unsigned int); 8048864 (hex); abcdefg (string);
*s = a (char);
```

Something Bad is Happening in Oz

We can do some very unexpected things with `printf` in the above example. Suppose we add the following:

```
/**********************************************************************/
/* Some unexpected things we can always do: */
printf("------------------------------------------------------------\n");

printf("i = %c (char); %f (floating point); %e (scientific notation);\n\n", i, i, i);
/* similar for j, k */

printf("f = %c (char); %i (signed int); %u (unsigned int); %x (hex);\n\n", f, f, f, f);

printf("p = %c (char); %i (signed int); %x (hex); %s (string);\n", p, p, p);

printf("*p = %c (char); %f (floating point); %e (scientific notation);\n\n", *p, *p, *p);

printf("s = %c (char); %i (signed int); %x (hex);\n", s, s, s);
printf("*s =  %i (signed int); %u (unsigned int);\n\n", *s, *s);

/* (int*) s casts s to an integer pointer */
printf("*((int*) s) = %i (signed int); %u (unsigned int); %x (hex);\n", *((int*)s), *((int*)s), *((int*)s));
printf("(97*256*256*256)+(98*256*256)+(99*256)+100=%u;\n",
       (97*256*256*256)+(98*256*256)+(99*256)+100);
printf("(100*256*256*256)+(99*256*256)+(98*256)+97=%u;\n\n",
       (100*256*256*256)+(99*256*256)+(98*256)+97);

/* a+i uses "pointer arithmetic" to give address of a[i] */
printf("*a = %c (char); %i (signed int); %u (unsigned int); %x (hex);\n", *a, *a, *a, *a);
printf("*a = %f (floating point); %e (scientific notation);\n", *a, *a);
printf("a+1 = %u (unsigned int); %x (hex); %s (string);\n", a+1, a+1, a+1);
printf("a+2 = %u (unsigned int); %x (hex); %s (string);\n\n", a+2, a+2, a+2);
```

```
/* a[i] is equivalent to *(a+i) */
printf("*a = %c (char); %i (signed int); %u (unsigned int); %x (hex);\n", *a, *a, *a, *a);
printf("*a = %f (floating point); %e (scientific notation);\n", *a, *a);
printf("*(a+1) = %c (char); %i (signed int); %u (unsigned int); %x (hex);\n", (a+1), *(a+1), *(a+1), *(a+1));
printf("*(a+1) = %f (floating point); %e (scientific notation);\n", *(a+1), *(a+1));
printf("*(a+2) = %c (char); %i (signed int); %u (unsigned int); %x (hex);\n", *(a+2), *(a+2), *(a+2), *(a+2));
printf("*(a+2) = %f (floating point); %e (scientific notation);\n", *(a+2), *(a+2));
printf("2[a] = %c (char); %i (signed int); %u (unsigned int); %x (hex);\n", 2[a], 2[a], 2[a], 2[a]);
printf("2[a] = %f (floating point); %e (scientific notation);\n", 2[a], 2[a]);
```

Then we get the following results (where code font shows the actual results and italics are some notes on how to interpret the results):

```
-----------------------------------------------------------
i = ^C (char); 0.000000 (floating point); 7.319816e-308 (scientific notation);
```
`^C` is the printed representation of ASCII 3.

```
f = ^@ (char); 1083394629 (signed int); 1610612736 (unsigned int); 40934a45 (hex);
```
*449a522b, *not* 40934a45, is the hex representation of the bits.*
The reason for the difference is that single-precision floats are converted.
to double-precision floats when they are passed to `printf` on the stack.

```
p = h (char); -1073758360 (signed int); bfffbf68 (hex); (null) (string);
```
`h` is ASCII 104 = x68; but why is string null?
`*p = * (char); 0.000000 (floating point); 7.319816e-308 (scientific notation);` *`*` is ASCII 42.*

`s = d (char); 134514788 (signed int); 8048864 (hex);` *`d` is ASCII 100 = x64.*
`*s =  97 (signed int); 97 (unsigned int);` *`a` is ASCII 97.*

```
*((int*) s) = 1684234849 (signed int); 1684234849 (unsigned int); 64636261 (hex);
```
`(97*256*256*256)+(98*256*256)+(99*256)+100=1633837924;` *big endean interpretation of "abcd".*
`(100*256*256*256)+(99*256*256)+(98*256)+97=1684234849;` *little endean interpretation of "abcd".*

```
a = P (char); -1073758384 (signed int); 3221208912 (unsigned int); bfffbf50 (hex);
```
`P` is ASCII 80 = x50.
```
a = -1.984208 (floating point); 1.591489e-314 (scientific notation); I (string);
a+1 = 3221208916 (unsigned int); bfffbf54 (hex); V^A (string);
a+2 = 3221208920 (unsigned int); bfffbf58 (hex); \357\376\377\377\252\207\363\I (string);
```

```
*a = ^Q (char); 17 (signed int); 17 (unsigned int); 11 (hex);
```
`^Q` is the printed representation of ASCII 17.
```
*a = 0.000000 (floating point); 1.591489e-314 (scientific notation);
*(a+1) = V (char); 342 (signed int); 342 (unsigned int); 156 (hex);
```
`V` is ASCII 86 = x56.
```
*(a+1) = 0.000000 (floating point); 1.591489e-314 (scientific notation);
*(a+2) = \357 (char); -273 (signed int); 4294967023 (unsigned int); fffffeef (hex);
```
-273 = -256 - 17; 256-17 = 239 = octal \357.
```
*(a+2) = nan (floating point); 1.591489e-314 (scientific notation);
2[a] = \357 (char); -273 (signed int); 4294967023 (unsigned int); fffffeef (hex);
2[a] = nan (floating point); 1.591489e-314 (scientific notation);
```

Below are some things that will only work sometimes (because they may cause segmentation errors by referring to addresses inaccessible to the current process:

```
printf("-------------------------------------------------------------\n");
printf("*i = %u (unsigned int); %s (string);\n", *((int*) i), *((int*) i)); /* similar for j, k */
/* Can't say ((int*) f) directly, but *can* say ((int*) ((int) f))! */
printf("*f = %u (unsigned int); %s (string);\n", *((int*) ((int) f)), *((int*) ((int) f)));
printf("*s = %s (string);\n", *s);
```

Walking the Stack

Using any of the pointers in our example (`a`, `p`, and `s`), we can walk through stack memory to learn more about its layout:

```
printf("------------------------------------------------------------\n");
for (i=-5; i<10; i++) {
  printf("%x: %i (signed int); %u (unsigned int); %x (hex);\n", a+i, *(a+i), *(a+i), *(a+i));
}
```

Here's the resulting printout. Can you find where the variables are stored?

```
bfffbf3c: 0 (signed int); 0 (unsigned int); 0 (hex);
bfffbf40: 14454680 (signed int); 14454680 (unsigned int); dc8f98 (hex);
bfffbf44: 134514788 (signed int); 134514788 (unsigned int); 8048864 (hex);
bfffbf48: -1073758360 (signed int); 3221208936 (unsigned int); bfffbf68 (hex);
bfffbf4c: 1150964267 (signed int); 1150964267 (unsigned int); 449a522b (hex);
bfffbf50: 17 (signed int); 17 (unsigned int); 11 (hex);
bfffbf54: 342 (signed int); 342 (unsigned int); 156 (hex);
bfffbf58: -273 (signed int); 4294967023 (unsigned int); fffffeef (hex);
bfffbf5c: 134514678 (signed int); 134514678 (unsigned int); 80487f6 (hex);
bfffbf60: 13359603 (signed int); 13359603 (unsigned int); cbd9f3 (hex);
bfffbf64: -1 (signed int); 4294967295 (unsigned int); ffffffff (hex);
bfffbf68: 42 (signed int); 42 (unsigned int); 2a (hex);
bfffbf6c: 7 (signed int); 7 (unsigned int); 7 (hex);
bfffbf70: 8753184 (signed int); 8753184 (unsigned int); 859020 (hex);
bfffbf74: 134514652 (signed int); 134514652 (unsigned int); 80487dc (hex);
```

Intel x86 Assembly Language

Since Intel x86 processors are ubiquitous, it is helpful to know how to read assembly code for these processors.

We will use the following terms: *byte* refers to 8-bit quantities; *short word* refers to 16-bit quantities; *word* refers to 32-bit quantities; and *long word* refers to 64-bit quantities.

There are many registers, but we mostly care about the following:

- EAX, EBX, ECX, EDX are 32-bit registers used for general storage.
- ESI and EDI are 32-bit indexing registers that are sometimes used for general storage.
- ESP is the 32-bit register for the *stack pointer*, which holds the address of the element currently at the top of the stack. The stack grows "up" from high addresses to low addresses. So pushing an element on the stack decrements the stack pointer, and popping an element increments the stack pointer.
- EBP is the 32-bit register for the *base pointer*, which is the address of the current activation frame on the stack (more on this below).
- EIP is the 32-bit register for the *instruction pointer*, which holds the address of the next instruction to execute.

At the end of this handout is a two-page "Code Table" summarizing Intel x86 instructions. The Code Table uses the standard Intel conventions for writing instructions. Unfortunately (and confusingly) the GNU assembler in class uses the so-called AT&T conventions, which are different. Some examples:

AT&T Format	**Intel Format**	**Meaning**
`movl $4, %eax`	`movl eax, 4`	Load 4 into EAX.
`addl %ebx, %eax`	`addl eax, ebx`	Put sum of EAX and EBX into EAX.
`pushl $X`	`pushl [X]`	Push the contents of memory location named X onto the stack.
`popl %ebp`	`popl ebp`	Pop the top element off the stack and put it in EBP.
`movl %ecx, -4(%esp)`	`movl [esp - 4] ecx`	Store contents of ECX into memory at an address that is 4 less than the contents of ESP.
`leal 12(%ebp), %eax`	`leal eax [ebp + 12]`	Load into EAX the address that is 12 more than the contents of EBP.
`movl (%ebx,%esi,4), %eax`	`movl eax [ebx + 4*esi]`	Load into EAX the contents of the memory location whose address
`cmpl $0, 8(%ebp)`	`cmpl [ebp + 8] 0`	Compare the contents of memory at an address 8 more than the contents of EBP with 0.
`jg L1`	`jg L1`	Jump to label `L1` if last comparison indicated "greater than".
`jmp L2`	`jmp L2`	Unconditional jump to label `L2`.
`call printf`	`call printf`	Call the `printf` subroutine.

We will focus on instructions that operate on 32-bit words, but there are ways to manipulate quantities of other sizes.

Typical Calling Conventions for Compiled C Code

The stack is typically organized into a list of activation frames. Each frame has a base pointer that points to highest address in the frame; since stacks grow from high to low, this is at the bottom of the frame. (Draw picture here:)

To maintain this layout, the calling convention is as follows:

1. The caller pushes the subroutine arguments on the stack from last to first.
2. The caller uses the `call` instruction to call the subroutine. This pushes the return address (address of the instruction after the `call` instruction) on the stack and jumps to the entry point of the called subroutine.
3. In order to create a new frame, the callee pushes the old base pointer and remembers the current stack address as the new base pointer via the following instructions:

```
pushl %ebp              # \ Standard callee entrance
movl %esp, %ebp         # /
```

4. The callee then allocates local variables and performs its computation.

When the callee is done, it does the following to return:

1. It stores the return value in the EAX register.
2. It pops the current activation frame off the stack via:

```
movl %ebp, %esp
pop %ebp
```

   This pair of instructions is often written as the `leave` pseudo-instruction.

3. It returns control to the caller via the `ret` instruction, which pops the return address off the stack and jumps there.
4. The caller is responsible for removing arguments to the call from the stack.

Understanding the `main` function of `reps.c`

We now know enough to understand the assembly code for the `main` function of `reps.c`. We can compile `reps.c` to assembly code as follows:

```
[cs342@puma] gcc -S reps.c
```

The resulting assembly code is put in the file `reps.s`. Below is some of the assembly code for the `main` function in `reps.s`.

```
        .section        .rodata  This declaration begins the area where read-only data like string are stored.
.LC0:
        .string        "abcdefg"
        .align 4
.LC1:
        .string        "-----------------------------------------------------------\n"
        .align 4
.LC2:
        .string        "i = %i (signed int); %u (unsigned int); %x (hex);\n\n"
        .align 4
         I elided a whole lot of strings here.
        .text  This declaration begins the area where code is stored.
.globl main    This says that the name main may be referenced and linked externally.
        .type        main,@function
main:
        pushl  %ebp First two instruction are standard callee prolog
        movl   %esp, %ebp
        subl   $72, %esp   Set aside space (18 words, more than we need) for local variables.
        andl   $-16, %esp -16 is xfffffff0, so this sets stack pointer to 16-byte boundary.
        movl   $0, %eax    I have no clue why the compiler generates the next two instructions.
        subl   %eax, %esp
        movl   $42, -16(%ebp)     Store 42 in 4th local word (j)
        movl   $-1, -20(%ebp)     Store -1 in 5th local word (k)
        movl   $17, -40(%ebp)     Store 17 in 10th local word (a[0])
        movl   $342, -36(%ebp)   Store 342 in 9th local word (a[1])
        movl   $-273, -32(%ebp) Store -273 in 8th local word (a[2])
        movl   $0x449a522b, -44(%ebp) Store bits for single-precision value 1234.5678 in 11th local word (f)
        leal   -16(%ebp), %eax Store address of 4th local word (j)
        movl   %eax, -48(%ebp)     into 12th local word (p)
        movl   $.LC0, -52(%ebp) Store address of "abcdefg" in 13th local word
        subl   $12, %esp      Push 3 dummy arguments on stack. Why?
        pushl  $.LC1          Push address of string for dotted line
        call   printf         Print the dotted line
        addl   $16, %esp      Pop arguments off stack.
        pushl  -12(%ebp)      Push contents of 3rd local word (i)
        pushl  -12(%ebp)      Push contents of 3rd local word (i)
        pushl  -12(%ebp)      Push contents of 3rd local word (i)
        pushl  $.LC2          Push string that displays i in various formats.
        call   printf         Print information about variable i
        addl   $16, %esp      Pop arguments off stack.
        The rest of the code (not shown) prints out the values in various other ways.
```

Using GDB to Disassemble Code

What if we don't have the source code to generate assembly code, but only the binary code? Then we can use the GNU Debugger (`gdb`) to disasseble the binary, as shown below:

```
[cs342@puma] gdb reps
GNU gdb Red Hat Linux (6.3.0.0-1.132.EL3rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...(no debugging symbols found)
Using host libthread_db library "/lib/tls/libthread_db.so.1".

(gdb) disassemble main
Dump of assembler code for function main:
0x08048344 <main+0>: push   %ebp
0x08048345 <main+1>: mov    %esp,%ebp
0x08048347 <main+3>: sub    $0x48,%esp
0x0804834a <main+6>: and    $0xfffffff0,%esp
0x0804834d <main+9>: mov    $0x0,%eax
0x08048352 <main+14>: sub    %eax,%esp
0x08048354 <main+16>: movl   $0x2a,0xfffffff0(%ebp)
0x0804835b <main+23>: movl   $0xffffffff,0xffffffec(%ebp)
0x08048362 <main+30>: movl   $0x11,0xffffffd8(%ebp)
0x08048369 <main+37>: movl   $0x156,0xffffffdc(%ebp)
0x08048370 <main+44>: movl   $0xfffffeef,0xffffffe0(%ebp)
0x08048377 <main+51>: movl   $0x449a522b,0xffffffd4(%ebp)
0x0804837e <main+58>: lea    0xfffffff0(%ebp),%eax
0x08048381 <main+61>: mov    %eax,0xffffffd0(%ebp)
0x08048384 <main+64>: movl   $0x80488b0,0xffffffcc(%ebp)
0x0804838b <main+71>: sub    $0xc,%esp
0x0804838e <main+74>: push   $0x80488b8
0x08048393 <main+79>: call   0x8048288
0x08048398 <main+84>: add    $0x10,%esp
0x0804839b <main+87>: pushl  0xfffffff4(%ebp)
0x0804839e <main+90>: pushl  0xfffffff4(%ebp)
0x080483a1 <main+93>: pushl  0xfffffff4(%ebp)
0x080483a4 <main+96>: push   $0x80488f8
0x080483a9 <main+101>: call   0x8048288
0x080483ae <main+106>: add    $0x10,%esp
```

Writing Assembly Code by Hand for the SOS Program

```
# HANDWRITTEN ASSEMBLY CODE FOR THE SOS PROGRAM (in the file sos.s)

        .section .rodata        # Begin read-only data segment
        .align 32               # Address of following label will be a multiple of 32
.fmt:                           # Label of SOS format string
        .string "sos(%i,%i)=%i\n" # SOS format string
.text                           # Begin text segment (where code is stored)
        .align 4                # Address of following label will be a multiple of 4
sq:                             # Label for sq() function
        pushl   %ebp            # \ Standard callee entrance
        movl    %esp, %ebp      # /
        movl    8(%ebp), %eax   # result <- x
        imull   8(%ebp), %eax   # result <- x*result
        leave                   # \ Standard callee exit
        ret                     # /
                .align 4        # Address of following label will be a multiple of 4
sos:                            # Label for sos() function
        pushl   %ebp            # \ Standard callee entrance
        movl    %esp, %ebp      # /
        pushl   8(%ebp)         # push y as arg to sq()
        call    sq              # %eax <- sq(y)
        movl    %eax, %ebx      # save sq(y) in %ebx
        addl    $4, %esp        # pop y off stack (not really necessary)
        pushl   12(%ebp)        # push z as arg to sq()
        call    sq              # %eax <- sq(z)
        addl    $4, %esp        # pop z off stack (not really necessary)
        addl    %ebx, %eax      # %eax <- %eax + %ebx
        leave                   # \ Standard callee exit
        ret                     # /
        .align 4                # Address of following label will be a multiple of 4
.globl main                     # Main entry point is visible to outside world
main:                           # Label for main() function
        pushl   %ebp            # \ Standard callee entrance
        movl    %esp, %ebp      # /

        # int a = atoi(argv[1])
        subl    $8, %esp        # Allocate space for local variables a and b
        movl    12(%ebp), %eax  # %eax <- argv pointer
        addl    $4, %eax        # %eax <- pointer to argv[1]
        pushl   (%eax)          # push string pointer in argv[1] as arg to atoi()
        call    atoi            # %eax <- atoi(argv[1])
        movl    %eax, -4(%ebp)  # a <- %eax
        addl    $4, %esp        # pop arg to atoi off stack

        # int b = atoi(argv[2])
        movl    12(%ebp), %eax  # %eax <- argv pointer
        addl    $8, %eax        # %eax <- pointer to argv[2]
        pushl   (%eax)          # push string pointer in argv[2] as arg to atoi()
        call    atoi            # %eax <- atoi(argv[2])
        movl    %eax, -8(%ebp)  # b <- %eax
```

```
        addl    $4, %esp        # pop arg to atoi off stack

        # printf("sos(%i,%i)=%d\n", a, b, sos(a,b))#
        # First calculate sos(a,b) and push it on stack
        pushl   -8(%ebp)        # push b
        pushl   -4(%ebp)        # push a
        call    sos             # %eax <- sos(a,b)
        addl    $8, %esp        # pop args to sos off stack
        pushl   %eax            # push sos(a,b)
        # Push remaining args to printf
        pushl   -8(%ebp)        # push b
        pushl   -4(%ebp)        # push a
        pushl   $.fmt           # push format string for printf
        # Now call printf
        call    printf
        addl    $16, %esp       # pop args to printf off stack (not really necessary)
        leave                   # \ Standard callee exit
        ret                     # /
# END OF ASSEMBLY CODE FILE
```

Here's how to compile and run our hand-written code:

```
[cs342@puma] gcc -o sos-by-hand sos-by-hand.s
[cs342@puma] sos-by-hand 3 4
sos(3,4)=25
[cs342@puma] sos-by-hand 10 5
sos(10,5)=125
```

(Part of) what the Compiler Produces:

```
# gcc -S sos.c
sq:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
        imull   8(%ebp), %eax
        leave
        ret

sos:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $4, %esp
        subl    $12, %esp
        pushl   8(%ebp)
        call    sq
        addl    $16, %esp
        movl    %eax, %ebx
        subl    $12, %esp
        pushl   12(%ebp)
        call    sq
        addl    $16, %esp
        addl    %eax, %ebx
        movl    %ebx, %eax
        movl    -4(%ebp), %ebx
        leave
       ret

# gcc -S -O3 sos.c
sq:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
        imull   %eax, %eax
        leave
        ret

sos:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
        movl    12(%ebp), %ecx
        imull   %eax, %eax
        imull   %ecx, %ecx
        addl    %ecx, %eax
        leave
        ret
```

A Recursive Factorial Program

Below is a C program for recursively calculating factorials.

```
/* This is the contents of the file fact.c */
int fact (int n) {
  if (n <= 0) {
    return 1;
  } else {
    return n*fact(n-1);
  }
}

int main (int argn, char** argv) {
  int x = atoi(argv[1]);
  printf("fact(%i)=%i\n", x, fact(x));
}
```

Let's compile it and take it for a spin!

```
[cs342@puma] gcc -o fact fact.c
[cs342@puma] fact 3
fact(3)=6
[cs342@puma] fact 4
fact(4)=24
```

Hand-written x86 Assembly for Recursive Factorial Program

Below is the result of hand-compiling the factorial program using the calling conventions studied earlier:

```
# This is the contents of the file fact-by-hand.s

        .section        .rodata # Begin read-only data segment
        .align 32               # Address of following label will be a multiple of 32
.fmt:                           # Label of fact program format string
        .string "fact(%i)=%i\n" # fact program format string
.text                           # Begin text segment (where code is stored)
        .align 4                # Address of following label will be a multiple of 4
fact:                           # Label for factorial function
        pushl %ebp              # \ Standard callee entrance
        movl %esp, %ebp         # /
        cmpl $0, 8(%ebp)        # Compare n and 0
        jg factGenCase          # Jump if greater to general case
        call print_stack        # Base case: show the stack state using Lyn's stack walker
        movl $1, %eax           # result <- 1
        jmp factRet             # Jump to shared return code
        .align 4                # Address of following label will be a multiple of 4
factGenCase:                    # Label for general case
        movl 8(%ebp), %eax      # %eax <- n
        subl $1, %eax           # %eax <- (n-1)
        pushl %eax              # push (n-1) for recursive call to factorial
        call fact               # call fact(n-1)
        imull 8(%ebp), %eax     # result <- n*result
        .align 4                # Address of following label will be a multiple of 4
factRet:                        # Shared return code for factorial
        leave                   # \ Standard callee exit
        ret                     # /
        .align 4                # Address of following label will be a multiple of 4
.globl main                      # Main entry point is visible to outside world
main:                           # Label for main() function
        pushl %ebp              # \ Standard callee entrance
        movl %esp, %ebp         # /
        subl $4, %esp           # Allocate space for local variable x
        movl 12(%ebp), %eax     # %eax <- argv pointer
        addl $4, %eax           # %eax <- pointer to argv[1]
        pushl (%eax)            # push string pointer in argv[1] as arg to atoi()
        call atoi               # %eax <- atoi(argv[1])
        movl %eax, -4(%ebp)     # Save x for later printf
        pushl %eax              # Push x for fact call
        call fact               # Call fact(x)
        pushl %eax              # Push result of fact(x) for printf
        pushl -4(%ebp)          # push x for printf
        pushl $.fmt             # push format string for printf
        call printf             # Call printf("fact(%i)=%i\n", n, fact(n))
        leave                   # \ Standard callee exit
        ret                     # /
```

Using GDB again

If only the binary for a program is available, can use the GNU Debugger (gdb) to disassemble it:

```
[cs342@puma overflow] gdb fact-by-hand
GNU gdb Red Hat Linux (6.3.0.0-1.132.EL3rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...(no debugging symbols found)
Using host libthread_db library "/lib/tls/libthread_db.so.1".

(gdb) disassemble main
Dump of assembler code for function main:
0x08048830 <main+0>: push   %ebp
0x08048831 <main+1>: mov    %esp,%ebp
0x08048833 <main+3>: sub    $0x4,%esp
0x08048836 <main+6>: mov    0xc(%ebp),%eax
0x08048839 <main+9>: add    $0x4,%eax
0x0804883c <main+12>: pushl  (%eax)
0x0804883e <main+14>: call   0x80482bc
0x08048843 <main+19>: mov    %eax,0xfffffffc(%ebp)
0x08048846 <main+22>: push   %eax
0x08048847 <main+23>: call   0x8048804 <fact>
0x0804884c <main+28>: push   %eax
0x0804884d <main+29>: pushl  0xfffffffc(%ebp)
0x08048850 <main+32>: push   $0x8048aa0
0x08048855 <main+37>: call   0x80482ac
0x0804885a <main+42>: leave
0x0804885b <main+43>: ret
End of assembler dump.
(gdb) disassemble fact
Dump of assembler code for function fact:
0x08048804 <fact+0>: push   %ebp
0x08048805 <fact+1>: mov    %esp,%ebp
0x08048807 <fact+3>: cmpl   $0x0,0x8(%ebp)
0x0804880b <fact+7>: jg     0x804881c <factGenCase>
0x0804880d <fact+9>: call   0x80485e9 <print_stack>
0x08048812 <fact+14>: mov    $0x1,%eax
0x08048817 <fact+19>: jmp    0x804882c <factRet>
0x08048819 <fact+21>: lea    0x0(%esi),%esi
End of assembler dump.
(gdb) disassemble 0x0804880b
Dump of assembler code for function fact:
0x08048804 <fact+0>: push   %ebp
0x08048805 <fact+1>: mov    %esp,%ebp
0x08048807 <fact+3>: cmpl   $0x0,0x8(%ebp)
0x0804880b <fact+7>: jg     0x804881c <factGenCase>
0x0804880d <fact+9>: call   0x80485e9 <print_stack>
0x08048812 <fact+14>: mov    $0x1,%eax
0x08048817 <fact+19>: jmp    0x804882c <factRet>
0x08048819 <fact+21>: lea    0x0(%esi),%esi
End of assembler dump.
```

Displaying the Stack

The hand-compiled factorial program uses a stack display program named `print_stack` that displays the state of the stack when it's called. Let's see what it does in the case of invoking the factorial program on 3:

```
[cs342@puma] gcc -o fact-by-hand print_stack.o fact-by-hand.s
[cs342@puma] fact-by-hand 3

[cs342@puma overflow] fact-by-hand 3
---------------------------------TOP-OF-STACK---------------------------------
bfffb358: bfffb360
bfffb35c: 08048812
bfffb360: bfffb36c
------------------
bfffb364: 08048828
bfffb368: 00000000
bfffb36c: bfffb378
------------------
bfffb370: 08048828
bfffb374: 00000001
bfffb378: bfffb384
------------------
bfffb37c: 08048828
bfffb380: 00000002
bfffb384: bfffb398
------------------
bfffb388: 0804884c
bfffb38c: 00000003
bfffb390: bfffd8a8 ->3
bfffb394: 00000003
bfffb398: bfffb3f8 ->
------------------
bfffb39c: 0061079a
bfffb3a0: 00000002
bfffb3a4: bfffb424
bfffb3a8: bfffb430
bfffb3ac: 00000000
bfffb3b0: 00730ab8
bfffb3b4: 00855020
bfffb3b8: 0804885c
bfffb3bc: bfffb3f8 ->
bfffb3c0: bfffb3a0
bfffb3c4: 0061075c
bfffb3c8: 00000000
bfffb3d4: 00855518
bfffb3d8: 00000002
bfffb3dc: 080482cc
bfffb3e0: 00000000
bfffb3e4: 0084c330
bfffb3e8: 006106cd
bfffb3ec: 00855518
bfffb3f0: 00000002
bfffb3f4: 080482cc
bfffb3f8: 00000000
------------------
bfffb3fc: 080482ed
```

```
bfffb400: 08048830
bfffb404: 00000002
bfffb408: bfffb424
bfffb40c: 0804885c
bfffb410: 080488a4
bfffb414: 0084ccc0
bfffb418: bfffb41c
bfffb41c: 00853133
bfffb420: 00000002
bfffb424: bfffd89b ->fact-by-hand
bfffb428: bfffd8a8 ->3
bfffb42c: 00000000
bfffb430: bfffd8aa ->BIBINPUTS=:/home/fturbak/church/lib/bibtex
bfffb434: bfffd8d5 ->DVIPSHEADERS=.:/usr/share/texmf/dvips//:/home/fturbak/lib/tex/psfonts/cmpsfont/pfb:/home/fturbak/
bfffb438: bfffd96a ->TWHOMEDIR=/home/cs307/public_html/tw
bfffb43c: bfffd98f ->HOSTNAME=puma.wellesley.edu
bfffb440: bfffd9ab ->BSTINPUTS=:/home/fturbak/church/lib/bibtex:/home/fturbak/lib/tex/jfp
bfffb444: bfffd9f0 ->SHELL=/bin/bash
bfffb448: bfffda00 ->TERM=dumb
bfffb44c: bfffda0a ->HISTSIZE=1000
bfffb450: bfffda18 ->SSH_CLIENT=149.130.162.226 50063 22
bfffb454: bfffda3c ->SSH_TTY=/dev/pts/5
bfffb458: bfffda4f ->USER=cs342
bfffb45c: bfffda5a ->EMACS=t
bfffb460: bfffda62 ->LS_COLORS=
bfffb464: bfffda6d ->TERMCAP=
bfffb468: bfffda76 ->COLUMNS=107
bfffb46c: bfffda82 ->MAIL=/var/spool/mail/cs342
bfffb470: bfffda9d ->PATH=/usr/java/sdk/bin:/usr/network/bin:/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6
bfffb474: bfffdb15 ->INPUTRC=/etc/inputrc
bfffb478: bfffdb2a ->PWD=/home/cs342/development/overflow
bfffb47c: bfffdb4f ->JAVA_HOME=/usr/java/sdk
bfffb480: bfffdb67 ->LANG=en_US.UTF-8
bfffb484: bfffdb78 ->SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
bfffb488: bfffdbab ->TEXINPUTS=:/home/cs230/lib/tex:/home/cs342/lib/tex:/home/fturbak/lib/tex:/home/cs230/lib/tex:/hom
bfffb48c: bfffde86 ->SHLVL=3
bfffb490: bfffde8e ->HOME=/home/cs342
bfffb494: bfffde9f ->LOGNAME=cs342
bfffb498: bfffdead ->PRINTER=minir
bfffb49c: bfffdebb ->CLASSPATH=:/home/cs230/download/HiLo:/home/cs230/download/TextFun:/home/cs230/download/TextStats
bfffb4a0: bfffdf1e ->SSH_CONNECTION=149.130.162.226 50063 149.130.136.19 22
bfffb4a4: bfffdf55 ->NPX_PLUGIN_PATH=/usr/java/j2sdk1.4.0/jre/plugin/i386/ns4
bfffb4a8: bfffdf8e ->LESSOPEN=|/usr/bin/lesspipe.sh %s
bfffb4ac: bfffdfb0 ->DISPLAY=localhost:10.0
bfffb4b0: bfffdfc7 ->G_BROKEN_FILENAMES=1
bfffb4b4: bfffdfdc ->_=./fact-by-hand
bfffb4b8: 00000000
bfffb4bc: 00000010
bfffb4c0: 0383fbff
bfffb4c4: 00000006
bfffb4c8: 00001000
bfffb4cc: 00000011
bfffb4d0: 00000064 [^@^@^@d]
bfffb4d4: 00000003
bfffb4d8: 08048034
bfffb4dc: 00000004
bfffb4e0: 00000020 [^@^@^@ ]
bfffb4e4: 00000005
```

```
bfffb4e8: 00000007
bfffb4ec: 00000007
bfffb4f0: 00840000
bfffb4f4: 00000008
bfffb4f8: 00000000
bfffb4fc: 00000009 [^@^@^@ ]
bfffb500: 080482cc
bfffb504: 0000000b
bfffb508: 00000e03
bfffb50c: 0000000c
bfffb510: 00000e03
bfffb514: 0000000d
bfffb518: 000001f5
bfffb51c: 0000000e
bfffb520: 000001f5
bfffb524: 0000000f
bfffb528: bfffd896 ->i686
bfffb52c: 00000000
~~~~~~~~: 00000000
bfffd894: 00000000
bfffd894: 36690000
bfffd898: 66003638 [f^@68]
bfffd89c: 2d746361 [-tca]
bfffd8a0: 682d7962 [h-yb]
bfffd8a4: 00646e61 [^@dna]
bfffd8a8: 49420033 [IB^@3]
bfffd8ac: 504e4942 [PNIB]
bfffd8b0: 3d535455 [=STU]
bfffd8b4: 6f682f3a [oh/:]
bfffd8b8: 662f656d [f/em]
bfffd8bc: 62727574 [brut]
bfffd8c0: 632f6b61 [c/ka]
bfffd8c4: 63727568 [cruh]
bfffd8c8: 696c2f68 [il/h]
bfffd8cc: 69622f62 [ib/b]
bfffd8d0: 78657462 [xetb]
bfffd8d4: 49564400 [IVD^@]
bfffd8d8: 45485350 [EHSP]
bfffd8dc: 52454441 [REDA]
bfffd8e0: 3a2e3d53 [:.=S]
bfffd8e4: 7273752f [rsu/]
bfffd8e8: 6168732f [ahs/]
bfffd8ec: 742f6572 [t/er]
bfffd8f0: 666d7865 [fmxe]
bfffd8f4: 6976642f [ivd/]
bfffd8f8: 2f2f7370 [//sp]
bfffd8fc: 6f682f3a [oh/:]
bfffd900: 662f656d [f/em]
bfffd904: 62727574 [brut]
bfffd908: 6c2f6b61 [l/ka]
bfffd90c: 742f6269 [t/bi]
bfffd910: 702f7865 [p/xe]
bfffd914: 6e6f6673 [nofs]
bfffd918: 632f7374 [c/st]
bfffd91c: 6673706d [fspm]
bfffd920: 2f746e6f [/tno]
bfffd924: 3a626670 [:bfp]
bfffd928: 6d6f682f [moh/]
```

```
bfffd92c: 74662f65 [tf/e]
bfffd930: 61627275 [abru]
bfffd934: 696c2f6b [il/k]
bfffd938: 65742f62 [et/b]
bfffd93c: 6d612f78 [ma/x]
bfffd940: 66737073 [fsps]
bfffd944: 702f746e [p/tn]
bfffd948: 2f3a6266 [/:bf]
bfffd94c: 656d6f68 [emoh]
bfffd950: 7574662f [utf/]
bfffd954: 6b616272 [kabr]
bfffd958: 7568632f [uhc/]
bfffd95c: 2f686372 [/hcr]
bfffd960: 2f62696c [/bil]
bfffd964: 2f786574 [/xet]
bfffd968: 5754002f [WT^@/]
bfffd96c: 454d4f48 [EMOH]
bfffd970: 3d524944 [=RID]
bfffd974: 6d6f682f [moh/]
bfffd978: 73632f65 [sc/e]
bfffd97c: 2f373033 [/703]
bfffd980: 6c627570 [lbup]
bfffd984: 685f6369 [h_ci]
bfffd988: 2f6c6d74 [/lmt]
bfffd98c: 48007774 [H^@wt]
bfffd990: 4e54534f [NTSO]
bfffd994: 3d454d41 [=EMA]
bfffd998: 616d7570 [amup]
bfffd99c: 6c65772e [lew.]
bfffd9a0: 6c73656c [lsel]
bfffd9a4: 652e7965 [e.ye]
bfffd9a8: 42007564 [B^@ud]
bfffd9ac: 4e495453 [NITS]
bfffd9b0: 53545550 [STUP]
bfffd9b4: 682f3a3d [h/:=]
bfffd9b8: 2f656d6f [/emo]
bfffd9bc: 72757466 [rutf]
bfffd9c0: 2f6b6162 [/kab]
bfffd9c4: 72756863 [ruhc]
bfffd9c8: 6c2f6863 [l/hc]
bfffd9cc: 622f6269 [b/bi]
bfffd9d0: 65746269 [etbi]
bfffd9d4: 682f3a78 [h/:x]
bfffd9d8: 2f656d6f [/emo]
bfffd9dc: 72757466 [rutf]
bfffd9e0: 2f6b6162 [/kab]
bfffd9e4: 2f62696c [/bil]
bfffd9e8: 2f786574 [/xet]
bfffd9ec: 0070666a [^@pfj]
bfffd9f0: 4c454853 [LEHS]
bfffd9f4: 622f3d4c [b/=L]
bfffd9f8: 622f6e69 [b/ni]
bfffd9fc: 00687361 [^@hsa]
bfffda00: 4d524554 [MRET]
bfffda04: 6d75643d [mud=]
bfffda08: 49480062 [IH^@b]
bfffda0c: 49535453 [ISTS]
bfffda10: 313d455a [1=EZ]
```

```
bfffda14: 00303030 [^@000]
bfffda18: 5f485353 [_HSS]
bfffda1c: 45494c43 [EILC]
bfffda20: 313d544e [1=TN]
bfffda24: 312e3934 [1.94]
bfffda28: 312e3033 [1.03]
bfffda2c: 322e3236 [2.26]
bfffda30: 35203632 [5 62]
bfffda34: 33363030 [3600]
bfffda38: 00323220 [^@22 ]
bfffda3c: 5f485353 [_HSS]
bfffda40: 3d595454 [=YTT]
bfffda44: 7665642f [ved/]
bfffda48: 7374702f [stp/]
bfffda4c: 5500352f [U^@5/]
bfffda50: 3d524553 [=RES]
bfffda54: 34337363 [43sc]
bfffda58: 4d450032 [ME^@2]
bfffda5c: 3d534341 [=SCA]
bfffda60: 534c0074 [SL^@t]
bfffda64: 4c4f435f [LOC_]
bfffda68: 3d53524f [=SRO]
bfffda6c: 52455400 [RET^@]
bfffda70: 5041434d [PACM]
bfffda74: 4f43003d [OC^@=]
bfffda78: 4e4d554c [NMUL]
bfffda7c: 30313d53 [01=S]
bfffda80: 414d0037 [AM^@7]
bfffda84: 2f3d4c49 [/=LI]
bfffda88: 2f726176 [/rav]
bfffda8c: 6f6f7073 [oops]
bfffda90: 616d2f6c [am/l]
bfffda94: 632f6c69 [c/li]
bfffda98: 32343373 [243s]
bfffda9c: 54415000 [TAP^@]
bfffdaa0: 752f3d48 [u/=H]
bfffdaa4: 6a2f7273 [j/rs]
bfffdaa8: 2f617661 [/ava]
bfffdaac: 2f6b6473 [/kds]
bfffdab0: 3a6e6962 [:nib]
bfffdab4: 7273752f [rsu/]
bfffdab8: 74656e2f [ten/]
bfffdabc: 6b726f77 [krow]
bfffdac0: 6e69622f [nib/]
bfffdac4: 73752f3a [su/:]
bfffdac8: 656b2f72 [ek/r]
bfffdacc: 72656272 [rebr]
bfffdad0: 622f736f [b/so]
bfffdad4: 2f3a6e69 [/:ni]
bfffdad8: 2f727375 [/rsu]
bfffdadc: 61636f6c [acol]
bfffdae0: 69622f6c [ib/l]
bfffdae4: 622f3a6e [b/:n]
bfffdae8: 2f3a6e69 [/:ni]
bfffdaec: 2f727375 [/rsu]
bfffdaf0: 3a6e6962 [:nib]
bfffdaf4: 7273752f [rsu/]
bfffdaf8: 3131582f [11X/]
```

```
bfffdafc: 622f3652 [b/6R]
bfffdb00: 2e3a6e69 [.:ni]
bfffdb04: 6f682f3a [oh/:]
bfffdb08: 632f656d [c/em]
bfffdb0c: 32343373 [243s]
bfffdb10: 6e69622f [nib/]
bfffdb14: 504e4900 [PNI^@]
bfffdb18: 43525455 [CRTU]
bfffdb1c: 74652f3d [te/=]
bfffdb20: 6e692f63 [ni/c]
bfffdb24: 72747570 [rtup]
bfffdb28: 57500063 [WP^@c]
bfffdb2c: 682f3d44 [h/=D]
bfffdb30: 2f656d6f [/emo]
bfffdb34: 34337363 [43sc]
bfffdb38: 65642f32 [ed/2]
bfffdb3c: 6f6c6576 [olev]
bfffdb40: 6e656d70 [nemp]
bfffdb44: 766f2f74 [vo/t]
bfffdb48: 6c667265 [lfre]
bfffdb4c: 4a00776f [J^@wo]
bfffdb50: 5f415641 [_AVA]
bfffdb54: 454d4f48 [EMOH]
bfffdb58: 73752f3d [su/=]
bfffdb5c: 616a2f72 [aj/r]
bfffdb60: 732f6176 [s/av]
bfffdb64: 4c006b64 [L^@kd]
bfffdb68: 3d474e41 [=GNA]
bfffdb6c: 555f6e65 [U_ne]
bfffdb70: 54552e53 [TU.S]
bfffdb74: 00382d46 [^@8-F]
bfffdb78: 5f485353 [_HSS]
bfffdb7c: 504b5341 [PKSA]
bfffdb80: 3d535341 [=SSA]
bfffdb84: 7273752f [rsu/]
bfffdb88: 62696c2f [bil/]
bfffdb8c: 63657865 [cexe]
bfffdb90: 65706f2f [epo/]
bfffdb94: 6873736e [hssn]
bfffdb98: 6f6e672f [ong/]
bfffdb9c: 732d656d [s-em]
bfffdba0: 612d6873 [a-hs]
bfffdba4: 61706b73 [apks]
bfffdba8: 54007373 [T^@ss]
bfffdbac: 4e495845 [NIXE]
bfffdbb0: 53545550 [STUP]
bfffdbb4: 682f3a3d [h/:=]
bfffdbb8: 2f656d6f [/emo]
bfffdbbc: 33327363 [32sc]
bfffdbc0: 696c2f30 [il/0]
bfffdbc4: 65742f62 [et/b]
bfffdbc8: 682f3a78 [h/:x]
bfffdbcc: 2f656d6f [/emo]
bfffdbd0: 34337363 [43sc]
bfffdbd4: 696c2f32 [il/2]
bfffdbd8: 65742f62 [et/b]
bfffdbdc: 682f3a78 [h/:x]
bfffdbe0: 2f656d6f [/emo]
```

```
bfffdbe4: 72757466 [rutf]
bfffdbe8: 2f6b6162 [/kab]
bfffdbec: 2f62696c [/bil]
bfffdbf0: 3a786574 [:xet]
bfffdbf4: 6d6f682f [moh/]
bfffdbf8: 73632f65 [sc/e]
bfffdbfc: 2f303332 [/032]
bfffdc00: 2f62696c [/bil]
bfffdc04: 3a786574 [:xet]
bfffdc08: 6d6f682f [moh/]
bfffdc0c: 65732f65 [es/e]
bfffdc10: 69727563 [iruc]
bfffdc14: 6c2f7974 [l/yt]
bfffdc18: 742f6269 [t/bi]
bfffdc1c: 2f3a7865 [/:xe]
bfffdc20: 656d6f68 [emoh]
bfffdc24: 7574662f [utf/]
bfffdc28: 6b616272 [kabr]
bfffdc2c: 7568632f [uhc/]
bfffdc30: 2f686372 [/hcr]
bfffdc34: 2f62696c [/bil]
bfffdc38: 3a786574 [:xet]
bfffdc3c: 6d6f682f [moh/]
bfffdc40: 74662f65 [tf/e]
bfffdc44: 61627275 [abru]
bfffdc48: 68632f6b [hc/k]
bfffdc4c: 68637275 [hcru]
bfffdc50: 62696c2f [bil/]
bfffdc54: 7865742f [xet/]
bfffdc58: 74616c2f [tal/]
bfffdc5c: 2f3a7865 [/:xe]
bfffdc60: 656d6f68 [emoh]
bfffdc64: 7574662f [utf/]
bfffdc68: 6b616272 [kabr]
bfffdc6c: 7568632f [uhc/]
bfffdc70: 2f686372 [/hcr]
bfffdc74: 2f62696c [/bil]
bfffdc78: 2f786574 [/xet]
bfffdc7c: 6574616c [etal]
bfffdc80: 6d612f78 [ma/x]
bfffdc84: 6e6f6673 [nofs]
bfffdc88: 2f3a7374 [/:st]
bfffdc8c: 656d6f68 [emoh]
bfffdc90: 7574662f [utf/]
bfffdc94: 6b616272 [kabr]
bfffdc98: 7568632f [uhc/]
bfffdc9c: 2f686372 [/hcr]
bfffdca0: 2f62696c [/bil]
bfffdca4: 2f786574 [/xet]
bfffdca8: 6574616c [etal]
bfffdcac: 6f662f78 [of/x]
bfffdcb0: 65746c69 [etli]
bfffdcb4: 682f3a78 [h/:x]
bfffdcb8: 2f656d6f [/emo]
bfffdcbc: 72757466 [rutf]
bfffdcc0: 2f6b6162 [/kab]
bfffdcc4: 72756863 [ruhc]
bfffdcc8: 6c2f6863 [l/hc]
```

```
bfffdccc: 742f6269 [t/bi]
bfffdcd0: 6c2f7865 [l/xe]
bfffdcd4: 78657461 [xeta]
bfffdcd8: 73696d2f [sim/]
bfffdcdc: 682f3a63 [h/:c]
bfffdce0: 2f656d6f [/emo]
bfffdce4: 72757466 [rutf]
bfffdce8: 2f6b6162 [/kab]
bfffdcec: 72756863 [ruhc]
bfffdcf0: 6c2f6863 [l/hc]
bfffdcf4: 742f6269 [t/bi]
bfffdcf8: 732f7865 [s/xe]
bfffdcfc: 6e696d65 [nime]
bfffdd00: 692f7261 [i/ra]
bfffdd04: 7475706e [tupn]
bfffdd08: 682f3a73 [h/:s]
bfffdd0c: 2f656d6f [/emo]
bfffdd10: 72757466 [rutf]
bfffdd14: 2f6b6162 [/kab]
bfffdd18: 72756863 [ruhc]
bfffdd1c: 6c2f6863 [l/hc]
bfffdd20: 742f6269 [t/bi]
bfffdd24: 6a2f7865 [j/xe]
bfffdd28: 2f3a7066 [/:pf]
bfffdd2c: 656d6f68 [emoh]
bfffdd30: 7574662f [utf/]
bfffdd34: 6b616272 [kabr]
bfffdd38: 7568632f [uhc/]
bfffdd3c: 2f686372 [/hcr]
bfffdd40: 2f62696c [/bil]
bfffdd44: 2f786574 [/xet]
bfffdd48: 69727073 [irps]
bfffdd4c: 7265676e [regn]
bfffdd50: 6e6c6c2d [nll-]
bfffdd54: 3a2f7363 [:/sc]
bfffdd58: 6d6f682f [moh/]
bfffdd5c: 74662f65 [tf/e]
bfffdd60: 61627275 [abru]
bfffdd64: 68632f6b [hc/k]
bfffdd68: 68637275 [hcru]
bfffdd6c: 62696c2f [bil/]
bfffdd70: 7865742f [xet/]
bfffdd74: 7473702f [tsp/]
bfffdd78: 6b636972 [kcir]
bfffdd7c: 682f3a73 [h/:s]
bfffdd80: 2f656d6f [/emo]
bfffdd84: 72757466 [rutf]
bfffdd88: 2f6b6162 [/kab]
bfffdd8c: 72756863 [ruhc]
bfffdd90: 6c2f6863 [l/hc]
bfffdd94: 742f6269 [t/bi]
bfffdd98: 702f7865 [p/xe]
bfffdd9c: 70736f72 [psor]
bfffdda0: 2f3a7265 [/:re]
bfffdda4: 656d6f68 [emoh]
bfffdda8: 7574662f [utf/]
bfffddac: 6b616272 [kabr]
bfffddb0: 7568632f [uhc/]
```

```
bfffddb4: 2f686372 [/hcr]
bfffddb8: 2f62696c [/bil]
bfffddbc: 2f786574 [/xet]
bfffddc0: 736f7270 [sorp]
bfffddc4: 2f726570 [/rep]
bfffddc8: 3a676d69 [:gmi]
bfffddcc: 6d6f682f [moh/]
bfffddd0: 74662f65 [tf/e]
bfffddd4: 61627275 [abru]
bfffddd8: 68632f6b [hc/k]
bfffdddc: 68637275 [hcru]
bfffdde0: 62696c2f [bil/]
bfffdde4: 7865742f [xet/]
bfffdde8: 6f63782f [ocx/]
bfffddec: 3a726f6c [:rol]
bfffddf0: 6d6f682f [moh/]
bfffddf4: 74662f65 [tf/e]
bfffddf8: 61627275 [abru]
bfffddfc: 68632f6b [hc/k]
bfffde00: 68637275 [hcru]
bfffde04: 62696c2f [bil/]
bfffde08: 7865742f [xet/]
bfffde0c: 6667702f [fgp/]
bfffde10: 682f3a2f [h/:/]
bfffde14: 2f656d6f [/emo]
bfffde18: 72757466 [rutf]
bfffde1c: 2f6b6162 [/kab]
bfffde20: 72756863 [ruhc]
bfffde24: 6c2f6863 [l/hc]
bfffde28: 742f6269 [t/bi]
bfffde2c: 622f7865 [b/xe]
bfffde30: 656d6165 [emae]
bfffde34: 61622f72 [ab/r]
bfffde38: 3a2f6573 [:/es]
bfffde3c: 6d6f682f [moh/]
bfffde40: 74662f65 [tf/e]
bfffde44: 61627275 [abru]
bfffde48: 68632f6b [hc/k]
bfffde4c: 68637275 [hcru]
bfffde50: 62696c2f [bil/]
bfffde54: 7865742f [xet/]
bfffde58: 6165622f [aeb/]
bfffde5c: 2f72656d [/rem]
bfffde60: 6d656874 [meht]
bfffde64: 3a2f7365 [:/se]
bfffde68: 7273752f [rsu/]
bfffde6c: 74656e2f [ten/]
bfffde70: 6b726f77 [krow]
bfffde74: 7865742f [xet/]
bfffde78: 742f666d [t/fm]
bfffde7c: 6e697865 [nixe]
bfffde80: 73747570 [stup]
bfffde84: 4853003a [HS^@:]
bfffde88: 3d4c564c [=LVL]
bfffde8c: 4f480033 [OH^@3]
bfffde90: 2f3d454d [/=EM]
bfffde94: 656d6f68 [emoh]
bfffde98: 3373632f [3sc/]
```

```
bfffde9c: 4c003234 [L^@24]
bfffdea0: 414e474f [ANGO]
bfffdea4: 633d454d [c=EM]
bfffdea8: 32343373 [243s]
bfffdeac: 49525000 [IRP^@]
bfffdeb0: 5245544e [RETN]
bfffdeb4: 6e696d3d [nim=]
bfffdeb8: 43007269 [C^@ri]
bfffdebc: 5353414c [SSAL]
bfffdec0: 48544150 [HTAP]
bfffdec4: 682f3a3d [h/:=]
bfffdec8: 2f656d6f [/emo]
bfffdecc: 33327363 [32sc]
bfffded0: 6f642f30 [od/0]
bfffded4: 6f6c6e77 [olnw]
bfffded8: 482f6461 [H/da]
bfffdedc: 3a6f4c69 [:oLi]
bfffdee0: 6d6f682f [moh/]
bfffdee4: 73632f65 [sc/e]
bfffdee8: 2f303332 [/032]
bfffdeec: 6e776f64 [nwod]
bfffdef0: 64616f6c [daol]
bfffdef4: 7865542f [xeT/]
bfffdef8: 6e754674 [nuFt]
bfffdefc: 6f682f3a [oh/:]
bfffdf00: 632f656d [c/em]
bfffdf04: 30333273 [032s]
bfffdf08: 776f642f [wod/]
bfffdf0c: 616f6c6e [aoln]
bfffdf10: 65542f64 [eT/d]
bfffdf14: 74537478 [tStx]
bfffdf18: 3a737461 [:sta]
bfffdf1c: 5353002e [SS^@.]
bfffdf20: 4f435f48 [OC_H]
bfffdf24: 43454e4e [CENN]
bfffdf28: 4e4f4954 [NOIT]
bfffdf2c: 3934313d [941=]
bfffdf30: 3033312e [031.]
bfffdf34: 3236312e [261.]
bfffdf38: 3632322e [622.]
bfffdf3c: 30303520 [005 ]
bfffdf40: 31203336 [1 36]
bfffdf44: 312e3934 [1.94]
bfffdf48: 312e3033 [1.03]
bfffdf4c: 312e3633 [1.63]
bfffdf50: 32322039 [22 9]
bfffdf54: 58504e00 [XPN^@]
bfffdf58: 554c505f [ULP_]
bfffdf5c: 5f4e4947 [_NIG]
bfffdf60: 48544150 [HTAP]
bfffdf64: 73752f3d [su/=]
bfffdf68: 616a2f72 [aj/r]
bfffdf6c: 6a2f6176 [j/av]
bfffdf70: 6b647332 [kds2]
bfffdf74: 2e342e31 [.4.1]
bfffdf78: 726a2f30 [rj/0]
bfffdf7c: 6c702f65 [lp/e]
bfffdf80: 6e696775 [nigu]
```

```
bfffdf84: 3833692f [83i/]
bfffdf88: 736e2f36 [sn/6]
bfffdf8c: 454c0034 [EL^@4]
bfffdf90: 504f5353 [POSS]
bfffdf94: 7c3d4e45 [|=NE]
bfffdf98: 7273752f [rsu/]
bfffdf9c: 6e69622f [nib/]
bfffdfa0: 73656c2f [sel/]
bfffdfa4: 70697073 [pips]
bfffdfa8: 68732e65 [hs.e]
bfffdfac: 00732520 [^@s% ]
bfffdfb0: 50534944 [PSID]
bfffdfb4: 3d59414c [=YAL]
bfffdfb8: 61636f6c [acol]
bfffdfbc: 736f686c [sohl]
bfffdfc0: 30313a74 [01:t]
bfffdfc4: 4700302e [G^@0.]
bfffdfc8: 4f52425f [ORB_]
bfffdfcc: 5f4e454b [_NEK]
bfffdfd0: 454c4946 [ELIF]
bfffdfd4: 454d414e [EMAN]
bfffdfd8: 00313d53 [^@1=S]
bfffdfdc: 2f2e3d5f [/.=_]
bfffdfe0: 74636166 [tcaf]
bfffdfe4: 2d79622d [-yb-]
bfffdfe8: 646e6168 [dnah]
bfffdfec: 662f2e00 [f/.^@]
bfffdff0: 2d746361 [-tca]
bfffdff4: 682d7962 [h-yb]
bfffdff8: 00646e61 [^@dna]
bfffdffc: 00000000
~~~~~~~~: 00000000
c0000000: 00000000
-------------------------------BOTTOM-OF-STACK-------------------------------
fact(3)=6
```

A Program to Be Hacked

Challenge: By being clever with the inputs to the following program, how many different answers can you get it to return?

```
/* A program that hints at issues involving software exploits */
/* Compile this as: gcc -o hackme print_stack.o hackme.c */

int sq (int x) {
  return x*x;
}

int getelt (int* a) {
  char c;
  int i;
  int prev = 0;
  printf("Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): ");
         scanf("%c", &c);
  while (c != 'r') {
    if (c == 'p') { /* print stack */
      print_stack();
    } else if ((c != 'g') && (c != 's')) {
      printf("unrecognized character '%c'\n", c);
    } else {
      printf("Enter an index: ");
      scanf("%i", &i);
      if (c == 'g') { /* get element at a[i] */
printf("getting a[%i]: %i\n", i, a[i]);
      } else if (c == 's') {
printf("setting a[%i] to %i\n", i, prev);
        a[i] = prev; /* set element at a[i] to previous value */
      }
      prev = a[i];
    }
    printf("Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): ");
    scanf("\n%c", &c); /* \n consumes newline from index entry */
  }
  return a[0]; /* always returns a[0] */
}

int process (int* a) {
  return sq(getelt(a));
}

int main () {
  int a[3] = {5,10,15};
  printf("***** ANS = %i *****\n", process(a));
}
```