CS342 Computer Security **Handout # 3**
Prof. Lyn Turbak **January 27, 2016**
Wellesley College

Lab 1: Introduction to Hacking in C

Reading:

- *Hacking*, 0x100, 0x210 – 0x240, 0x260 – 0x280
- Scott Anderson's *C and C++ for Java Programmers* (linked from the `cs342` Resources page)

Overview

Later in the course, we will study software vulnerabilities like buffer overflow attacks, format string exploits, and viruses. We need to know a lot of skills, tools, and low-level details in order to understand/launch such exploits.

In today's lab, we will begin to study C programming, which is an essential tool in our journey to understand exploits.

Getting the code

In this lab, you will need two programs, `sos.c` and `reps.c`. These files can be found on any of the CS dept linux machines (which all share the same filesystem) in `/home/cs342/download/intro-to-c`. Copy them into your own account as follows:

```
cd ~
mkdir cs342
cd cs342
scp -r ~cs342/download/intro-to-c .
```

These commands (1) create a local directory named `cs342` in your account and copy the remote directory `cs342/download/intro-to-c` into this new local directory. The `-r` flag to `scp` indicates to recursively copy the contents of a directory.

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

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

#include <stdio.h>

/* 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@localhost intro-to-c] gcc -o sos sos.c

[cs342@localhost intro-to-c] sos 3 4
sos(3,4)=25

[cs342@localhost intro-to-c] sos -9 -10
sos(-9,-10)=181

[cs342@localhost intro-to-c] sos foo bar
sos(0,0)=0

[cs342@localhost intro-to-c] sos 3.1 4.9
sos(3,4)=25

[cs342@localhost intro-to-c] 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.

- Can you explain the following results?

```
[lynux@localhost intro-to-c]$ sos 40000 1
sos(40000,1)=1600000001

[lynux@localhost intro-to-c]$ sos 50000 1
sos(50000,1)=-1794967295
```

C Types and Their Representations

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

```
/* Contents of the file reps.c */
#include <stdlib.h>
#include <stdio.h>

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@localhost intro-to-c] gcc -o reps reps.c

    [cs342@localhost intro-to-c] 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, 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);
/* Note: *s is treated as an 8-bit value because s has type char*.
   To treat it as a 32 bit value, we need to cast it to an int*,
   as in *((int*)s), which is done below. */

/* (int*) s casts s to an integer pointer, which points to a 32-bit value */
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));

/* Both 2[a] and a[2] are treated like *(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):[1]

```
------------------------------------------------------------
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); * (string);
# h is ASCII 104 = x68 (showing word bytes are little endian); can you explain the string?
*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 endian interpretation of "abcd".
(100*256*256*256)+(99*256*256)+(98*256)+97=1684234849; # little endian 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);
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);
# hex ef = octal \357.
# Alternatively: -273 = -256 - 17; 256-17 = 239 = octal \357.
*(a+2) = nan (floating point); 1.591489e-314 (scientific notation);
# nan stands for a ''not a number'' error in the IEEE floating point standard.
2[a] = \357 (char); -273 (signed int); 4294967023 (unsigned int); fffffeef (hex);
2[a] = nan (floating point); 1.591489e-314 (scientific notation);
```

[1]Your results may vary from these, even on different runs on the same machine. Why?

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.[2] Can you find where the variables are stored?

```
bffffbf3c: 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);
```

[2]Again, your results may vary from these, even on different runs on the same machine. Why?