CS342 Computer Security **Handout # 21**
Profs. Daniel Bilar and Lyn Turbak **November 8, 2006**
Wellesley College

Problem Set 5
Due: Midnight Wed. November 15

Reading:

- Aleph One, "Smashing the Stack for Fun and Profit" (can be found at `http://cs.wellesley.edu/~security/papers/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`).
- (Optional) Jon Erickson, *Hacking: The Art of Exploitation*, Chapter 2.

Working Together:
You should work on both problems on this assignment with your lab buddies on your Linux workstation in the Security Lab. You may use the same "loosely coupled" strategy we allowed for PS3 — i.e., you need not do all work with all team members working at the console, but you need to work in a way that guarantees that all team members completely understand the solutions to the problems.

Tutorial:
I will be giving (perhaps more than one instantiation of) a tutorial involving hands-on hacking, including some coverage of format string vulnerabilities, which we didn't get to in class. Please watch for emails on the scheduling of this tutorial.

Problem 1 [Assembly Code]:

The goal of this problem is to make sure that you are familiar with assembly code, stack organization, and calling conventions. You will need detailed knowledge of these in order to succeed with the exploits in Problem 2.

You should begin this problem by studying the `fact-by-hand.s` assembly language program from Handout #14 and its associated stack trace. Do not proceed with this problem until you understand the details of this example. (You can find a copy of `fact-by-hand.s` and all other code files mentioned in this problem set can be found on puma in the directory `~cs342/download/ps5`.)

a. : Recursive Fibonacci

Recall the recursive definition of the Fibonacci function:

$$fib(n) = \begin{cases} n, \text{ if } n <= 1 \\ fib(n-1) + fib(n-2), \text{ otherwise} \end{cases}$$

Following the form of the assembly code for the recursive factorial function in `fact-by-hand.s`, write assembly code for the recursive Fibonacci function in a file named `fibSlow.s`. You should obey all the normal procedure calling conventions. You do not need to include the base case call to `print_stack`. Compile your function via `gcc -o fibSlow fibSlow.s` and show that it works on the following inputs:

```
[cs342@puma ps5] fibSlow 5
fibSlow(5)=5
[cs342@puma ps5] fibSlow 10
fibSlow(10)=55
[cs342@puma ps5] fibSlow 15
fibSlow(15)=610
[cs342@puma ps5] fibSlow 20
fibSlow(20)=6765
[cs342@puma ps5] fibSlow 25
fibSlow(25)=75025
[cs342@puma ps5] fibSlow 30
fibSlow(30)=832040
[cs342@puma ps5] fibSlow 35
fibSlow(35)=9227465
[cs342@puma ps5] fibSlow 40
fibSlow(40)=102334155
[cs342@puma ps5] fibSlow 45
fibSlow(45)=1134903170
```

Note: Your function should take a noticeably long time for the input 45.

b. : A Faster Recursive Fibonacci

The reason that `fibSlow` is so slow on inputs ≥ 40 is that it unnecessarily recomputes many intermediate values. For example, `fibSlow(45)` computes `fibSlow(44)` and `fibSlow(43)`, while `fibSlow(44)` repeats the computation of `fibSlow(43)`. All the unnecessary repeated computations slow the process up considerably.

It is possible to avoid the repeated computations by having the recursive Fibonacci function return *two* values for the input n: (1) the Fibonacci of n and (2) the Fibonacci of $n-1$. This is illustrated by the following OCAML code in which `fib'` returns a pair of this form and `fib` returns the first component of the pair returned by `fib'`:

```
let rec fib n =
  let (fib_n, fib_n_minus_1) = fib'(n)
   in fib_n

and fib' n =
  if n <= 1 then
    (n, n-1)
  else
    let (fib_n_minus_1, fib_n_minus_2) = fib'(n-1)
     in (fib_n_minus_1 + fib_n_minus_2, fib_n_minus_1)
```

Your goal in this part is to write assembly code in the file `fibFast.s` for a version of the recursive Fibonacci function named `fibFast` based on the above idea. In order to make your `fibFast` function particularly efficient, you should observe the following conventions in place of the usual procedure calling conventions:

- the parameter `n` to `fibFast` should be passed in the `%eax` register rather than on the stack.
- the two results of calling `fibFast` should be returned in the registers `%eax` (the Fibonacci of `n`) and `%ebx` (the Fibonacci of `n-1`).
- the only thing that `fibFast` needs to push on the stack is the return address for a nested call to `fibFast`. In particular, unlike in the usual calling convention, there is no need to change the base pointer in any of the calls to `fibFast`.

Compile your function via `gcc -o fibFast fibFast.s` and show that it works on the same inputs used for `fibSlow`. You should notice that `fibFast 45` is extremely fast compared to `fibSlow 45`.

Problem 2: Game Exploits

Figs. 1–3 show a number-guessing game program written in C. (This program is in the `ps5` download directory.) The game prompts the user to guess a randomly generated integer. It is nearly impossible to win if played "honestly". Fortunately (!?), the game is vulnerable to several kinds of buffer overflow and format string exploits that allow a wily hacker to beat it.

Below are some sample interactions with the game:

```
[cs342@puma ps5] game
Guess a number> 123
123 is not the secret number.
Guess a number> 0x2a
42 is not the secret number.
Guess a number> done
You did not guess the secret number.
Bye! Play again soon!
[cs342@puma ps5] game "foo\tbar\nbaz> "
foo bar
baz> 1\x372
172 is not the secret number.
foo bar
baz> 4\t5
4 is not the secret number.
foo bar
baz> quit
You did not guess the secret number.
Bye! Play again soon!
[cs342@puma ps5] cat guesses.txt
123
4
6\x37
\xFF
[cs342@puma ps5] game "guess> " < guesses.txt
guess> 123 is not the secret number.
guess> 4 is not the secret number.
guess> 67 is not the secret number.
guess> You did not guess the secret number.
Bye! Play again soon!
[cs342@puma ps5]
```

Note the following:

- The game ends when (a prefix of) the guess cannot be parsed as an integer.
- Guessed numbers are normally entered in decimal, but can be specified in hex via a leading `0x` or `0X`.
- Escape characters (e.g., `\t` for tab, `\n` for newline, `\x`*NN* for ASCII character with hex value *NN*) can be used in guesses. For example, `\x37` is ASCII 55, which stands for the digit '7', so `1\x372` is another way to write `172`.
- The escape characters `\t` and `\n` can be used in the optional user-supplied prompt string.
- As demonstrated in the last example, gueses can be supplied from a file.

To do the first two subproblems, you will need to know about format string vulnerabilities, as described in the "Exploiting Format String Vulnerabilities" paper. Read the first 3 sections of this paper. You will *not* need to use any of the complicated techniques mentioned in paper, only simple ones.

a. By supplying an appropriate prompt string, you can examine the stack and determine what the secret number is. Describe how you can do this, and show a transcript in which you win the game using this technique. *Hint:* What happens if you try the following?

```
game "%08x %08x %08x %08x\n> "
game "%12u %12u %12u %12u\n> "
game "%012u %012u %012u %012u\n> "
```

(The numbers written before a specifier like `x` or `u` indicate the width of the printed field. So 8 specifies an 8-character field and 12 specifies a 12-character field. If the number begins with a 0, then "empty" slots of the field will be filled with 0; otherwise, they will be left blank.)

The above examples can also be written

```
game "`perl -e 'print "%08 "x4 . "\n> ";'`"
game "`perl -e 'print "%12u "x4 . "\n> ";'`"
game "`perl -e 'print "%012u "x4 . "\n> ";'`"
```

(The outer double quotes are essential to making the examples work as desired.) The `perl` versions are handy for replicating a string *many* times.

These are just examples of what you can do; you will need to do more to find the secret number.

b. By supplying an appropriate prompt string, you can change the secret number to be any nonnegative number of your choosing. Explain how to do this, and show transcripts in which you win the game using this technique with (1) the most convenient number; (2) the number 42 and (3) the number 12345.

Hints:

- Where is the address of the `secret` pointer on the stack? You should be able to tell this by parsing the stack trace from part a. The value of this address may change between games, but its relative position will be constant.
- Once you know the relative position of the `secret` pointer, you can overwrite it using the `%n` format string specifier. There is one "natural" value to overwrite it with that depends on where `%n` appears in the format string. This is the "most convenient number".
- You can generalize the convenient number to any nonnegative integer. It helps to use the direct parameter access notation mentioned in Section 4.3 of the "Exploiting Format String Vulnerabilities" paper. For example, executing the the C statement

  ```
  printf("%i,%5i,%05i\n%2$i,%3$5i,%2$05i\n", 10, 20, 30);
  ```

  yields the following result:

  ```
  10,   20,00030
  20,   30,00020
  ```

  Normally, a format specifier refers to the next argument (i.e., value on the stack), which is why the three specifiers `%i,%5i,%05i` refer to 10, 20, and 30, respectively. But a specifier beginning with `%`*n*`$` refers to the *n*th argument (i.e., value on the stack), regardless of how many arguments have been processed so far. So `%2$i` and `%2$05i` both refer to 20 and `%3$5i` refers to 30.

Because the $ symbol is special in both Linux and Perl, care must be taken to properly escape it when using it in a string argument to `game`. For example, what would be written as `%3$5i` in C must be written as `%3\$5i` in the Linux command line and as `%3\\\$5i` in string within a backquoted Perl expression in the Linux command line (because Perl will treat \\ as \ and \$ as `$`, and Linux requires \ before `$`.)

c. By overflowing the `buff` buffer, you can change the return address of the call to `make_guess` so that it returns to the part of `main` that declares you won the game. Explain how to do this, and show a transcript in which you win the game using this technique.

Hint: Use the `gdb disassemble` command to determine the correct return address.

Notes:

- If you get a segmentation fault after "winning", that will receive only partial credit. To get full credit, you must "win" without getting any segmentation fault.
- Because of stack randomization, the values in certain stack slots may change on every execution of `game`. This should not affect your ability to solve the problem. However, if you find the randomization annoying, you can turn it off by executing the following command as `root`:

  ```
  echo 0 > /proc/sys/kernel/randomize_va_space
  ```

  (You can turn stack randomization back on by changing the `0` to `1`.)
- You may want to modify `game.c` to use calls to `print_stack` or your own stack-printing code to debug any problems you encounter. Debugging code needs to be carefully place so as not to interfere with your exploit . To compile the game with `print_stack`, use

  ```
  gcc -o game print_stack.o game.c
  ```

d. By using the technique for overwriting the return address in part c, you can make the `game` program spawn a shell and execute arbitrary commands in that shell. If the `game` program is setuid `root`, then a nonroot player of the game will be able to execute commands as root! Demonstrate that this is possible by (1) making the `game` binary setuid `root` and (2) as `guest`, launch a buffer overflow exploit on the `game` program that spawns a root shell.

Notes:

- In order to execute shellcode on the stack, you will need to turn off ExecShield by executing the following as root:

  ```
  echo 0 /proc/sys/kernel/exec-shield
  ```

  (You can turn it back on by changing the `0` to `9`.)
- You do not need stack randomization turned off for this exploit, but it may be more convenient if you do.
- The shellcode developed on Handout #19 was specialized for user `cs342`. You will need to tweak it so that it works for `root` instead.
- Because `buff` is small (16 characters), the shellcode cannot be located in the same way as shown in class. Where can you put it instead? How can you find the address of the shellcode to jump to

it? (*Hint:* Parsing a relatively small area near the top of the stack gives you all the information you need.)

- You may want to modify game.c to use calls to print_stack or your own stack-printing code to debug any problems you encounter. Debugging code needs to be carefully place so as not to interfere with your exploit. To compile the game with print_stack, use

```
gcc -o game print_stack.o game.c
```

e. All of the above problems come from poorly written code. Describe all the buffer overflow and format string vulnerabilities in game.c and how you could eliminate them by modifying the program.

```
// Almost impossible game in which the user must guess a random number.
// Luckily, the program is poorly written and has security holes via
// which a savvy player can win (or spawn a shell!).

// Compile this program via "gcc -o game game.c"
// If you include a call to print_stack for debugging,
//   compile the program via "gcc -o game print_stack.o game.c"

#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include "print_stack.h"

// Forward declarations; see definition below.
int make_guess (char *guess_prompt, int* secret_ptr);
unsigned char hexval (unsigned char c);
char * handle_escapes (char *s);
void readline (char* tgt);

int main (int argc, char** argv) {
  //int main () {
  int result = 17; // arbitrary value that will be overwritten
  int secret;
  char *prompt;
  // set the random number generator seed to the current time.
  srand((unsigned int) time (NULL));
  secret = rand(); // secret is a random number.
  prompt = "Guess a number> "; // default prompt
  if (argc >= 2) {
     prompt = handle_escapes(argv[1]); // user can supply the prompt.
  }
  while ((result = make_guess(prompt, &secret))==0) {
    // plays until user guesses or gives up.
  }
  if (result == 1)
    printf("%u is the secret number -- you won!\n", secret);
  else
    printf("You did not guess the secret number.\n");
  printf("Bye! Play again soon!\n");
}
```

Figure 1: The `game` program, part 1.

```
// Reads a line of input from the user, using guess_prompt as a prompt.
// If the user input is all digits, checks if the corresponding number
//   is equal to the number in location secret_ptr, and prints
//   a message about whether or not it is equal.
// Returns:  1 if guess is a correct number;
//           0 if guess is an incorrect number;
//           -1 if guess is not a number.
int make_guess (char *guess_prompt, int* secret_ptr) {
  char buff[16];
  int guess, scan_result;
  printf(guess_prompt);
  readline((char *)buff);
  scan_result = sscanf(buff, "%i", &guess);
  // sscanf(buff, "%i", &guess) reads the string in buff as an integer and stores the
  //   resulting integer in guess. Strings beginning "0x" or "0X" are read as hex.
  //   scan_result is 1 if the string in buff is parsable as an integer an 0 otherwise.
  if (scan_result == 0) // guess was not a number
    return -1;
  else if (guess == *secret_ptr) {
    return 1;
  } else {
    printf("%u is not the secret number.\n", guess);
    return 0;
  }
}

// Convert a hex character to a value 0-15.
// Return 0 for any non-hex character.
unsigned char hexval (unsigned char c) {
  if (('0' <= c) && (c <= '9')) {
    return c - '0';
  } else if (('a' <= c) && (c <= 'f')) {
    return 10 + (c - 'a');
  } else if (('A' <= c) && (c <= 'F')) {
    return 10 + (c - 'A');
  } else {
    return 0;
  }
}
```

Figure 2: The game program, part 2.

```
// Handle escapes in given string.
// Return pointer to given string
char * handle_escapes (char *s) {
  char *src = s;
  char *tgt = s;
  unsigned char hex;
  while (*src != '\0') {
    if (*src != '\\') {
      *tgt++ = *src++; // copy source char to target char
    } else { // it's an escape sequence
      src++; // go to next char after slash
      if (*src == 't') {
        *tgt++ = '\t'; src++;
      } else if (*src == 'n') {
        *tgt++ = '\n'; src++;
      } else if (*src == 'r') {
        *tgt++ = '\r'; src++;
      } else if (*src == '\'') {
        *tgt++ = '\''; src++;
      } else if (*src == '"') {
        *tgt++ = '"'; src++;
      } else if (*src == '\\') {
        *tgt++ = '\\'; src++;
      } else if (*src == 'x') { // two character hex code follows
        src++;
        hex = 16*(hexval (*src++)); // read first hex character
        hex += hexval (*src++); // read second hex character
        *tgt++ = hex;
      }
    }
  }
  *tgt = '\0'; // terminate target with NUL
  return s; // return given pointer
}

// Read line up to newline or EOF into buffer TGT.
// Assume TGT is big enough to store results.
// Handle escaped characters as in handle_escapes.
void readline (char* tgt) {
  char line[2048]; // Assume a large buffer
  char* lineptr = (char *) line;
  char c = getchar();
  while ((c != EOF) && (c != '\n')) { // Note: EOF is -1
    *lineptr++ = c;
    c = getchar();
  }
  *lineptr = '\0'; // Terminate string
  handle_escapes(line);
  strcpy(tgt,line);
}
```

Figure 3: The game program, part 3.