CS349 Computer Security
Prof. Lyn Turbak
Wellesley College

Handout # 4
October 29, 2004

Homework 4

Due: Anytime Friday November 5

This is the final version of HW4.

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`).

Working Together:

- Problem 1 is about choosing a final project. You may choose to work on a final project individually or with a partner. If you choose a partner, it need not be your sysadmin partner.
- In Problems 2 and 3, you may work alone or in pairs. In these problems you may work with a partner who is *not* your sysadmin partner. This will give you a chance to work with someone new in the class if you desire.
- In Problems 2 and 3, you may work on any CS Linux machine, not just the ones in the security lab. Nothing on this assignment endangers other computers.

Problem 1: Choosing a Final Project Topic

This course culminates in a final project in which you (either individually or with a partner) work on a security-related project of your choice. The project should have the following properties:

- It must be in the area of computer security;
- You must be passionate enough about the topic to (1) invest significant time and energy exploring it; (2) give a class presentation on it; and (3) write a report on it.
- Ideally, the project should involve more than just becoming familiar with the literature on a topic. It should have a hands-on technical component in which you implement programs, write exploits, experiment with software, etc.

In this task, your goal is to think about what you want to do for your final project. For this task, you should do the following:

- Explore security-related topics to find ones that interest you. A good starting point is the list in Fig. 1. You should also skim the resources on the bookshelves in the security lab.
- Talk to your classmates about their interests. If you and a classmate share an interest, consider a pair project.
- Have one or more discussions with Lyn about topics in which you're interested.
- Submit a paragraph explaining what you want to work on for your final project and why.

Here are the critical stepping stones in the project:

1. **Fri. Nov. 5**: Submit a paragraph explaining your choice of topic.
2. **Fri. Nov. 19**: Submit (1) a summary of your progress on the project so far and (2) an outline of your project in the form of a (not-yet-fleshed out) skeleton of your final report.
3. **Last four classes meetings (Tue. Nov. 30 – Fri. Dec. 10)**: Give a 30-or-so minute presentation on your project in class.
4. **Mon. Dec. 13**: Submit a draft of your final report. I will get you feedback by Wed. Dec. 15.
5. **Tue. Dec. 21**: Final reports due by 4:30pm.

- Evaluate the security measures for one or more of the electronic systems you use at Wellesley: the CWIS, FirstClass, puma, wilbur, on-line course registration, and on-line SEQs.
- Implement a honeypot and report on what you find.
- Research security holes that have been fixed since Red Hat 7.3 and write exploits that take advantage of the security holes in Red Hat 7.3.
- Design and possibly implement a security protocol for a problem of interest to you.
- Complete some of the hacker challenges in books like *Hacker's Challenge* and *Hacker's Challenge 2* or on sites like `http://happyhacker.org/`. Report on your experiences.
- Report on issues in electronic voting.
- Study information warfare – how vulnerable are companies and countries?
- Write your own rootkit or experiment with an existing one.
- Study denial of service attacks and countermeasures.
- Experiment with firewalls.
- Report on the security features/holes of Java and/or C#.
- Study/implement security features for e-commerce.
- Experiment with cryptographic protocols.
- Investigate the security issues in Microsoft's Next-Generation Secure Computing Base architecture.
- Study/evaluate biometric authentication.
- Report on video surveillance.
- Study security issues associated with wireless systems.
- Study security issues associated with radio frequency identification (RFID) systems.
- Study and report on Windows/MAC vulnerabilities and how they differ from those in Linux.
- Study watermarking and/or steganography.
- Study systems for prevent digital piracy and means for circumventing these systems.
- Study viruses and/or virus protection programs.
- Evaluate an existing virus protection program.
- Study a class of viruses/worms in depth.
- Study spyware and/or spyware protection programs.
- Evaluate and compare security tools: e.g. tools for hardening Linux (Bastille, LIDS, Openwall patches), scan detectors, etc.
- Report on electronic money.
- Report on zero-knowledge proofs.
- Explore security issues in network file systems like NFS and AFS.
- Experiment with Linux exploits/tools not covered in class.
- Study MULTICS and compare its security features to LINUX.
- Study one or more of the security models discussed in Bishop and show how they are used in practice.
- Study information flow systems.
- Investigate security issues involved in sharing software like PC Anywhere.
- Investigate the benefits/drawbacks of smart cards.

Figure 1: Some final project ideas. These are only suggestions; you are welcome to choose any idea you wish.

Problem 2 [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 3. All code files mentioned in this problem can be found on puma in the directory `~security/download/hw4`.

a. : Recursive Factorial

Fig. 2 shows hand-written assembly code for a recursive factorial function. The `print_stack` function mentioned in the base case is the stack-displaying utility that I presented in lecture.

1. What are the advantages and disadvantages of writing assembly code for recursive factorial by hand?
2. Compile the code via

   ```
   gcc -o fact print_stack.o fact.s
   ```

   Invoke it on the input 5 by executing `fact 5`. In addition to displaying the factorial of 5, the call to `print_stack` will display the contents of the stack when the base case is reached. Annotate a transcript of the frames in this printout explaining the purpose of each frame.

b. : 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.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:

```
[security@wampeter hw4] fibSlow 5
fibSlow(5)=5
[security@wampeter hw4] fibSlow 10
fibSlow(10)=55
[security@wampeter hw4] fibSlow 15
fibSlow(15)=610
[security@wampeter hw4] fibSlow 20
fibSlow(20)=6765
[security@wampeter hw4] fibSlow 25
fibSlow(25)=75025
[security@wampeter hw4] fibSlow 30
fibSlow(30)=832040
[security@wampeter hw4] fibSlow 35
fibSlow(35)=9227465
[security@wampeter hw4] fibSlow 40
fibSlow(40)=102334155
[security@wampeter hw4] fibSlow 45
fibSlow(45)=1134903170
```

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

```
        .section        .rodata
        .align 32
.fmt:
        .string        "fact(%d)=%d\n"
.text
        .align 4
fact:
        pushl   %ebp
        movl    %esp, %ebp
        cmpl    $0, 8(%ebp)
        jg      factGenCase
        call    print_stack     # base case: show the stack state
        movl    $1, %eax
        jmp     factRet
        .align 4
factGenCase:
        movl    8(%ebp), %eax   # retrieve n
        subl    $1, %eax
        pushl   %eax
        call    fact            # call fact(n-1)
        imull   8(%ebp), %eax   # ans <- n*ans
        .align 4
factRet:
        leave
        ret
        .align 4
.globl main
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp        # space to store n
        movl    12(%ebp), %eax  # argv pointer
        addl    $4, %eax        # pointer to argv[1]
        pushl   (%eax)          # pointer in argv[1]
        call    atoi            # convert string to int n
        movl    %eax, -4(%ebp)  # save n for later printf
        pushl   %eax            # push for fib1 call
        call    fact            # call fact(n)
        pushl   %eax            # push result of fact(n) for printf
        movl    -4(%ebp), %eax
        pushl   %eax            # push n for printf
        pushl   $.fmt           # push format string for printf
        call    printf
        leave
        ret
```

Figure 2: Hand-written assembly code for a recursive factorial function (contents of the file `fact.s`).

c. : 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 `%edx` (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 3: Game Exploits

Figs. 3–5 show the number-guessing game program written in C that we have discussed in class. 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. Note that the game handles escaped characters in both the (optional) user-supplied prompt as well as the guessed numbers. Also note that the guesses can be supplied from a file.

```
[security@wampeter hw4] game
Guess a number> 123
guess=123
Guess a number> 45
guess=45
Guess a number> done
guess=done
You did not guess the secret number.
Bye! Play again soon!
[security@wampeter hw4] game "foo\tbar\nbaz>"
foo bar
baz>1\x323
guess=123
foo bar
baz>4\t5
guess=4 5
You did not guess the secret number.
Bye! Play again soon!
[security@wampeter hw4] cat guesses.txt
123
4
6\x37
\xFF
[security@wampeter hw4] game "guess> " < guesses.txt
guess> guess=123
guess> guess=4
guess> guess=67
guess> guess=\377
```

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.

b. By supplying an appropriate prompt string, you can change the secret number to be a number of your choosing. Explain how to do this, and show a transcript in which you win the game using this technique.

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. (Use the `gdb disassemble` command to determine the correct return address.) Explain how to do this, and show a transcript in which you win the game using this technique.

d. Another way to overwrite the return address of the call to `make_guess` is to supply an appropriate prompt string. Explain how to do this, and show a transcript in which you win the game using this technique.

e. By using one of the previous two techniques for overwriting a return address, you can make the `game` program spawn a shell and execute arbitrary commands in that shell. If the `game` program were setuid root, you would be able to execute commands as root! Show how to make the game program spawn an arbitrary shell. You may want to use the approach shown in `exploit4` of the stack smashing paper.

f. All of the above problems come from poorly written code. Make a version of `game.c` called `mygame.c` in which all buffer overflow and format string vulnerabilities are fixed.

```
// 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 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 i;
  int secret;
  int result = 17; // arbitrary value (helps attacker find this spot on stack.)
  char *prompt;
  srand((unsigned int) time (NULL));
    // set the random number generator seed to the current time.
  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 == 0) {
    printf("You won by guessing the secret number: %d.\n", secret);
  } else {
    printf("You did not guess the secret number.\n");
  }
  printf("Bye! Play again soon!\n");
}
```

Figure 3: 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 either (1) the guess is correct or (2) the input
//   is not a number (indicating the game is over).
// Otherwise returns 0 (indicating the game is not over).
int make_guess (char *guess_prompt, int* secret_ptr) {
  char buff[16];
  char* ptr;
  char c;
  int all_digits;
  // print_stack();  // uncomment this line to see full stack
  printf(guess_prompt);
  readline((char *)buff);
  // Check if all chars in buff are digits
  ptr = buff;
  all_digits = 1;
  while (*ptr != '\0') {
    c = *ptr++;
    all_digits = all_digits && ('0' <= c) && (c <= '9');
  }
  printf("guess=%s\n", buff);
  // end of input
  // printf("all_digits=%d, i=%d\n", all_digits, i);
  if (all_digits) {
    return (atoi(buff) != *secret_ptr);
  } else {
    return -1; // game is over
  }
}

// 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 4: The `game` program, part 2.

```
// 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;
  }
}

// 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
}
```

Figure 5: The `game` program, part 3.