CS342 Computer Security
Prof. Lyn Turbak
Wellesley College

Handout # 3
September 8, 2014
Revised September 14, 2014

Problem Set 1
Due: 11:59pm, Thursday September 18

Revisions: Sep 14: In Problem 2, clarify that (1) search for "vulnerabilities" should be case insensitive and (2) symbolic links should note be followed.

Overview:

The purpose of this assignment is to get you to get you in the mindset of thinking about computer security and getting you familiar with some concepts and tools you will need.

Working Together:

The basic rule of collaboration in this class is that you can talk with anyone about how to solve any problem but you can't take away anything (notes, programs, etc.) from such a discussion and must write up all programs and solutions completely on your own. This is essentially Yoshi Kohno's "Gilligan's Island Rule" described at

https://www.cs.washington.edu/education/courses/cse484/12sp/administrivia/overview.html,

whose purpose is to ensure that you truly understand everything you write up. You must also explicitly list your collaborators on every problem (Kohno's "Freedom of Information Rule").

Submission:

You will submit all your work for this pset in a single *PS1* Google doc linked from your CS342 *Portfolio* doc that you create in Problem 1. Follow these guidelines:

1. In your course *Blog* (see Problem 1) you should summarize any issues you encountered while working on the problems, insights you gained, resources you consulted, etc.
2. At the beginning of each problem, indicate the time you spent on that problem.
3. For Problem 2, submit a transcript of the Linux command and its results.
4. For Problem 3, submit the the decoded T-shirt message and explain how you decoded it.
5. For Problem 4, submit the Unicode codepoint sequence and an image of how the characters should look. Also explain how you determined these.
6. For Problem 5, submit (1) a transcript of your REPL interaction that culminates with the desired result and (2) a companion explanation of why you chose the commands you did.
7. For Problem 6, submit the report described in the problem description.

Problem 1 [5]: Creating your portfolio doc and starting your course blog

Create a Google doc entitled **your name CS342 Portfolio** that you share with me with commenting privileges. You will use this *Portfolio* doc to submit all your work for this course. This doc will simply contain a list of links to docs of your submitted work. For starters, it will have a link to your *Blog* doc (see below) and your *PS1* doc (see above.)

Create a second Google doc entitled ***your name* CS342 Blog** that you share with me with commenting privileges and that you link from your *Portfolio* doc. You will use this *Blog* doc to record your journey through the course. Every day you work on the course, you should add a dated entry to the top of this page summarizing your work for the day, emphasizing insights you gained, difficulties you encountered, resources you discovered, and security-related news/articles you read. Include images when appropriate. Put all your entries in a single document in reverse chronological order.

cat	gunzip	popd	telnet
cd	gzip	ps (-ef)	top
chmod (-R)	info	pushd	touch
chown (-R)	less	pwd	umount
chgrp (-R)	ln (-s)	rm (-rf)	useradd
cp (-R)	ls (-al)	scp (-r)	usermod
df	mkdir	ssh	wc
du	more	source	which
echo	mount	su (-)	whoami
find	nice	tar (-cvf, -xvf,	xargs
grep	passwd (and the file	-cvzf, -xvzf,	
	/etc/password)	-cvjf, -xvjf)	

Figure 1: Some Linux commmands with which you should be familiar.

Problem 2 [15]: Linux Skills

In this course, you will be doing a lot of work with Linux. So it's important to gain familiarity with lots of Linux commands. Figure 1 lists some of the commands you should be familiar with. The list is by no means exhaustive! Many of the commands have lots of options; some of the common options ones are listed. To get documentation on this commands, use `man` and `info`, browse on-line resources, and refer to the many Linux books on the security bookshelves in SCI 160A.

You should also play with pipes (`|`) and input/output redirection (`<`, `>`, `>>`) and learn a little bit about shell scripts (which we will cover soon in a lab or lecture). You can learn about Linux commands and shell scripts by consulting the Linux resources on the CS342 **Resources** page.

For this problem, you should play around with Linux commmands and shell programming. The only thing you need to submit for this problem is the following:

> In your account on cs.wellesley.edu (tempest), write a single Linux command that lists every line containing the word "vulnerabilities" in every publicly readable text (non-binary) file in the directory `/home/cs342`. Each line should have the form *filename*:*line*, where *line* is the contents of the line containing the word and *filename* is the name of the file containing that line. The `find` and `grep` commands are very useful for this problem. The `2>` error redirection mechanism is helpful for suppressing annoying error messages.
>
> Your command should ignore case. E.g., it should list lines containing strings like "Vulnerabilities", "VULNERABILITIES", and "vulneraBiliTies".
>
> Your command should also *not* follow symbolic links. This will greatly reduce the number of matching lines. Using the `find` command in conjunction with `grep` is helpful in this regard.
>
> Submit a transcript of the command and the results of executing it.

Problem 3 [15]: Decoding a T-shirt

Margaret Ligon '14, a student in the 2012 CS342 class, designed a T-shirt for the Wellesley participants in the Lincoln Laboratory Capture the Flag (CTF) contest. The back of the T-shirt is shown in figure 2. The binary digits and decimal numbers on the shirt stand for a message. What is the message?

Guidelines/hints

- An ASCII table such as the one at `http://www.asciitable.com/` is very helpful in this problem.
- For full credit, you must not only specify the message, but describe the details by which you decoded it.

Figure 2: Design on the 2012 Lincoln Lab CTF T-shirt worn by Wellesley CS342 students.

Problem 4 [15]: Decoding a message

You have been sent a message that consists of the following bytes (expressed in hex):

49 E2 99 A5 CF 80 21

You have been informed that this message consists of Unicode characters that have been encoded into a byte sequence via the UTF-8 encoding.

a [12] What is the sequence of Unicode codepoints in this UTF-8 encoded message? For example,

codepoint `U+03B1` is the Greek letter α (http://unicode.org/charts/PDF/U0370.pdf). Explain how you determined the codepoints.

b [3] What would this message look like when displayed in a Unicode-enabled application?

Guidelines/hints

- Consult the following on Unicode: http://www.joelonsoftware.com/articles/Unicode.html, http://en.wikipedia.org/wiki/Unicode
- Consult the following on UTF-8: http://en.wikipedia.org/wiki/UTF-8.

Problem 5 [15]: Exploiting a C program

This problem illustrates the fundamental insecurity of arrays in C and hints at the role that character buffers play in stack buffer exploits.

Figure 3 is a C program `arrays.c` available in `~cs342/download/ps1`. When compiled and executed, it enters a read/eval/print loop (REPL) that responds to commands. For example, the `display` command shows the addresses and contents of all slots in the three integer arrays `a`, `b`, and `c` manipulated by the program:

```
[cs342@jay ps1] gcc -o arrays arrays.c
[cs342@jay ps1] ./arrays
Enter one of these three commands: display, setb, quit:
> display
bfbca1d8 a[0]: 1 (int); 1 (hex)
bfbca1dc a[1]: 2 (int); 2 (hex)
bfbca1cc b[0]: 3 (int); 3 (hex)
bfbca1d0 b[1]: 4 (int); 4 (hex)
bfbca1d4 b[2]: 5 (int); 5 (hex)
bfbca1bc c[0]: 6 (int); 6 (hex)
bfbca1c0 c[1]: 7 (int); 7 (hex)
bfbca1c4 c[2]: 8 (int); 8 (hex)
bfbca1c8 c[3]: 9 (int); 9 (hex)
Enter one of these three commands: display, setb, quit:
>
```

Your task is to enter a sequence of commands that ends with a `display` command that prints exactly the following array contents:

```
Enter one of these three commands: display, setb, quit:
> display
bfbca1d8 a[0]: 1 (int); 1 (hex)
bfbca1dc a[1]: 17 (int); 11 (hex)
bfbca1cc b[0]: 3 (int); 3 (hex)
bfbca1d0 b[1]: 23 (int); 17 (hex)
bfbca1d4 b[2]: 5 (int); 5 (hex)
bfbca1bc c[0]: 42 (int); 2a (hex)
bfbca1c0 c[1]: 7 (int); 7 (hex)
bfbca1c4 c[2]: 8 (int); 8 (hex)
bfbca1c8 c[3]: 9 (int); 9 (hex)
Enter one of these three commands: display, setb, quit:
> quit
```

For this problem, submit (1) a complete transcript of your REPL interaction that ends with the desired result and (2) a companion explanation of why you chose the commands you did.

Guidelines/hints

- Run this program on one of the micro-focus Linux machines (e.g., `cardinal`, `jay`, `lark`, `thrush`, etc.); you can connect to these remotely via `ssh`.
- Study `arrays.c` carefully to understand exactly how it works. But you are not allowed to change it in any way!
- It is straightforward to use the `setb` command to change `b[1]`, but how the heck you change `a[1]` and `c[0]`? That's the crux of this problem, and it requires thinking like a hacker.
- The addresses of all integer array slots and character array slots are displayed for a reason. Study them to understand the key to your exploits. Of course, in a typical program they wouldn't be printed, but you'd have to imagine they were there. You'd also have to do a lot more experimentation without them.
- An unexpected command string can be used in a nefarious way!

Problem 6 [35]: Hacking the Tanner Photo Contest Sites

In 2010, I implemented a website that allows members of the Wellesley community to submit photos to the annual Tanner photo contest. The purpose of the website is to allow members of the Wellesley community (and no one else) to submit up to three photos to the contest, none of which should be larger than 5MB. Users should be able to view the photos that they've uploaded, but no one else's.

In this problem, you will experiment with an early version of the 2010 website
http://cs.wellesley.edu/~tanner-test
and the current 2014 version at
http://cs.wellesley.edu/~tanner-photos.
In the "hacker curriculum" spirit, your goal in this problem is to find out as much as you can about the implementation of these websites, and see whether you can coax the websites into doing some unexpected things. Here are some of the questions that you should investigate (but do not be limited by these!):

- Can a user without Wellesley email or domain name access the system?
- Can you view the photo files uploaded by others? How about the photo metadata (title, location, notes) and personal data (name, email, position at Wellesley)?
- Can you change the photo files uploaded by others? How about their photo metadata and personal data?
- Can you find the source code for the website? Does that help you find other vulnerabilities?
- Can you upload more than 3 files?
- Can you upload files other than photos?
- Can you upload files larger than 5MB?
- Can you use the website to view or modify resources of user `tanner-test` or `tanner-photos`?
- Can you do anything else you consider surprising?

Write up a report describing the experiments you performed on both websites, summarizing all the vulnerabilities or surprising behaviors you discovered. Where possible, suggest ways in which vulnerabilities could be removed or made less damaging. If they can't be removed, explain why. Also, explain the rationale of security design decisions that were made in these sites, and compare the approaches of the two sites. E.g., the 2010 site requires registering with an email address, and then the user follows a link to their submission page. Why is this done? What does the 2014 site do differently? **Bonus points will be awarded for finding vulnerabilities in the 2014 tanner-photos site!**

You are welcome to talk about and/or perform experiments with others, but the write-up must be done on your own.

```
# include <stdio.h> // I/O operations
# include <string.h> // string operations

// Display the word addresses and integer contents of len slots of given array
void display_array(char* name, // string name for array
                   int* array, // array of ints is pointer to (word address of) int in 0th slot
                   int len) {  // need to pass length of array separately
  int i = 0; // initialize index to 0
  int* end = array + len; // end is address of word after last array slot
  for (; array < end; array++) { // Loop iterates through word addresses of array slots.
                                 // Incrementing adds 4 to the address b/c array is int pointer
    printf("%x %s[%i]: %i (int);\t%x (hex)\n", array, name, i, *array, *array);
      // *array is contents of current slot
    i++; // increment index in sync with address of next slot
  }
}

// Display the byte addresses and character contents of all slots in given string
void display_chars(char* str) { // str is pointer to char in 0th slot
  while (*str != 0) { // Loop while terminating null byte hasn't been reached
    printf("%x: %c (char), %x (hex)\n", str, *str, *str);
    str++; // Incrementing adds 1 to address b/c str is a char pointer
  }
}

// Return the maximum of two integers
int max (int a, int b) { if (a > b) return a; else return b; }

// Program entry point
int main (int argn, char** argv) { // argn and argv are ignored in this program

  int a[2] = {1,2}; // Allocate integer arrays on stack
  int b[3] = {3,4,5};
  int c[4] = {6,7,8,9};
  char command[8]; // Stack space allocated for command string character buffer.
                   // No one would type more than 8 characters, would they? ;-)
  int index; // Stack space allocated for "setb" command index
  int value; // Stack space allocated for "setb" command value

  // Read/eval/print loop: read a command from user, perform action, and repeat
  while (1) { // 1 is how "true" is written in C; "infinite" loop exited via "quit" command
    printf("Enter one of these three commands: display, setb, quit:\n> "); // Prompt for command
    scanf("%s", &command); // Read command into character buffer
    if (strcmp(command,"display")==0) { // strcmp compares strings; 0 result means they're equal
      // Display the addresses and contents of slots in all arrays
      display_array("a", a, 2);
      display_array("b", b, 3);
      display_array("c", c, 4);
    } else if (strcmp(command,"setb")==0) { // Change slots in array b
      printf("Choose an index for array b: "); // Prompt for index of array b to change
      scanf("%i", &index); // Store it into index variable
      index = max(index,0); // Don't allow negative indices; convert negative index to 0
      printf("Choose a new value for b[%i]: ", index); // Prompt for new value at index
      scanf("%i", &value); // Store it into value variable
      b[index] = value; // Change the indexth slot of b to new value
    } else if (strcmp(command,"quit")==0) {
      return 0; // Exit loop by returning from main function
    } else { // Case for unrecognized commands
      printf("\"%s\" is not a recognized command.\n");
      display_chars(command);  // Display the addresses and contents of characters in command
      printf("Try again.\n");
    }
  }
}
```

Figure 3: A C program manipulating some arrays.