CSCI 5271: Introduction to Computer Security

Exercise Set 1 **due: October 2, 2020**

Ground Rules. You may choose to work with one other student if you wish. Only one submission is required per group, please ensure that both group members names are on the submitted copy. Work must be submitted on canvas by the marked date and time.

1. Threat Models and Risk Assessment. Suppose the course instructor has created a database of all information for this course: homeworks, exams, handouts, and grades. Create a detailed threat model for this database: what should the security goals be? What are reasonable attacks, and who are the potential attackers? What threats should we explicitly exclude from consideration?

Now assume that the database is stored on the instructor's personal laptop, with no network card and no floppy disk drive.[1] Propose at least two security mechanisms that would help counter your threat model (e.g. file or disk encryption, a laptop lock, a safe to store the laptop, a kevlar laptop sleeve, relocation to Fort Knox . . .), and analyze the net risk reduction of both. You should justify your estimates for the various incidence rates and costs. While we do want to see numbers for this part, don't worry about figuring out exact costs or risk reductions, guess at some reasonable numbers but don't spend very long at this part of the assignment

2. Finding vulnerabilities. Here are a few code excerpts. For this exercise, you'll find the vulnerability in each and describe how to exploit it.

(a) Below is a short POST-method CGI script - it reads a line of the form "field-name=value" from standard input, and then executes the `last` command (in the line `$result = ‘last ...‘`) to see if the user name "value" has logged in recently. Describe how to construct an input that executes an arbitrary command with the privileges of the script. Explain how your input will cause the program to execute your command, and suggest how the code could be changed to avoid the problem.

```
#!/usr/bin/perl
print "content-type: text/html\r\n\r\n<HTML><BODY>\n";

($field_name, $username_to_look_for) = split(/=/, <>);
chomp $username_to_look_for;

$result = ‘last -1000 | grep $username_to_look_for‘;
if ($result) {
  print "$username_to_look_for has logged in recently.\n";
} else {
  print "$username_to_look_for has NOT logged in recently.\n";
}
print "</BODY></HTML>\n";
```

[1] *n.b.: this database does not exist; don't waste your time trying to attack it!*

(b) This is a short (and poorly written) C function that deletes the last byte from any file that is not the *extremely* important file `/what/ever`. Describe how to exploit a race condition to make the function delete the last byte of `/what/ever`, assuming the program has read and write access to the file (`/what/ever`) but the user does not. Your description should list what file the fixed string `pathname` refers to at each important point in the exploit, and why it will work. (You can read about the various system calls used below at `http://www.linuxmanpages.com/man2/`)

```
void silly_function(char *pathname) {
  struct stat f, we;
  int rfd, wfd;
  char *buf;
  stat(pathname, &f);
  stat("/what/ever", &we);
  if (f.st_dev == we.st_dev && f.st_ino == we.st_ino) {
    return;
  }
  rfd = open(pathname, O_RDONLY);
  buf = malloc(f.st_size - 1);
  read(rfd, buf, f.st_size - 1);
  close(rfd);

  stat(pathname, &f);
  if (f.st_dev == we.st_dev && f.st_ino == we.st_ino) {
    return;
  }
  wfd = open(pathname, O_WRONLY | O_TRUNC);
  write(wfd, buf, f.st_size - 1);
  close(wfd);
  free(buf);
}
```

3. Overflowing buffers. Let's examine a few proposed solutions to the problem of stack smashing.

(a) *Reversing the Stack.* It is not uncommon, upon reading $\aleph_1$'s tutorial, for someone to propose to eliminate stack-smashing attacks by reversing the direction of stack growth, so that the stack grows in the same direction as buffer addresses. With this arrangement, as long as the return address comes earlier in the stack frame than local variables, a procedure can never overwrite its own return address by manipulating only local variables. Consider the function `func` defined in lecture:

```
void func(char *str) {
  char buf[128];
  strcpy(buf, str);
  do_something();
  return;
}
```

Describe an attack that will exploit this function's use of `strcpy` to hijack control flow, even if the stack grows in the same direction as buffers: A good description will show the contents of the stack at the important points of the attack and walk us through the control flow under the attack. (E.g. at the level of detail in slide 7, lecture 3.)

(b) Properly placed *canaries* can protect return addresses and frame pointers from being modified by simple buffer overflows. Other kinds of stack-based buffer overflows can still be possible, however. For example, imagine the following code that is supposed to ensure that the system has only 100 active connections at any time.

```
int accept_connection(char *user, int *total_connections) {
  int tmp = *total_connections;
  char buf[128];
  strcpy(buf, user);
  if (tmp < 100) {
    strcat(buf, ":Y");
    *total_connections++;
    tmp = 1;
  } else {
    strcat(buf, ":N");
    tmp = 0;
  }
  printf("%s\n", buf);
  return tmp;
}
```

Describe an input that overflows `buf` so that even if there are 100 connections, and even if canaries are used, the system will accept additional connections. Your input should not write beyond the local variables of `accept_connection`. (Hint: different inputs are required for different endian-nesses)

4. Nondefensive programming. Let's practice finding "bad programming practices" that could lead to exploits. There are at least five issues with each of these.

(a) Here's a chunk of code that's intended to "zip" two strings together. Find all of the potential bugs in the code and suggest a better implementation. Notice that there are some input cases that are specifically not defined by this code: it is up to you to pick an interpretation for these cases and explain your rationale.

```
char *zip(char *a, char *b) {
  char *result;
  int len, i;
  len = strlen(a);
  result = malloc(2*len);
  for(i = 0; i <= len; i++) {
    result[2*i] = a[i];
    result[2*i+1] = b[i];
  }
  return result;
}
```

(b) The very, very buggy code below is intended to merge, in place, two sorted, linked lists into a single sorted, linked list: it is supposed to take two lists as input and destructively modify them so that at the end, they form a single, sorted, linked list in memory. Find as many potential bugs in the code as you can:

```
typedef struct link {
  char *data;
  struct link *next;
} list;

list *merge(list *input1, list *input2) {
  list *output = NULL, *tail, *next;
  do {
    if(strcmp(input1->data, input2->data) <= 0) {
      next = input1;
      input1 = input1->next;
    } else {
      next = input2;
      input2 = input2->next;
    }
    if (output == NULL) {
      output = tail = next;
    } else {
      tail->next = next;
      tail = tail->next;
    }
  } while (input1 != NULL && input2 != NULL);
  if (input1 != NULL) tail->next = input1;
  else tail->next = input2;
  return output;
}
```