# CSci 5271: Introduction to Computer Security

**Exercise Set 1** **due: Wednesday September 25th, 2019**

**Ground Rules.** You may choose to complete these exercises in a group of up to three students. Each group should turn in **one** copy with the names of all group members on it. You may use any source you can find to help with this assignment but you **must** explicitly reference any source you use besides the lecture notes or textbook. An electronic (PDF) copy of your solution should be submitted on Canvas by 11:59pm on Wednesday, September 25th.

**1. Threat models and risk assessment.** (15 pts) Suppose the course instructor has created a database of all the information for this course: homeworks, exams and solutions, 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 ancient personal laptop, which has no network hardware.[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. Remember that net risk reduction is a formula, so you should have numeric estimates of the costs of attacks and defense mechanisms, the rates of attacks, etc. You should justify these estimates for the various incidence rates and costs.

**2. Finding vulnerabilities.** (20 pts) Here are a few code excerpts. For each part, find the vulnerability and describe how to exploit it.

(a) Below is a short POST-method CGI script written in Perl. It reads a line of the form "field-name=value" from the 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 two good ways the code could be changed to avoid the problem.

```
#!/usr/bin/perl
print "Content-Type: text/html\r\n\r\n";
print "<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] This is a hypothetical situation, not reflecting the way the course information is really stored. Of course honest students such as yourselves wouldn't be tempted to attack the course information.

(The Perl operation `` `cmd` ``, pronounced "backticks", passes the string `cmd` to a shell, and returns the output of `cmd` in a string. You can get more detailed documentation under `man perlop`.)

(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 `/highly/critical`. Describe how to exploit a race condition to make the function delete the last byte of `/highly/critical`, assuming that the program has read and write access to the file `/highly/critical` 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 explain why the steps will work. (You can read documentation for Unix/Linux system calls with a command like `man 2 stat` on a Linux machine, or at various places on the web.)

```
void silly_function(char *pathname) {
  struct stat f, we;
  int rfd, wfd;
  char *buf;
  stat(pathname, &f);
  stat("/highly/critical", &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.** (30 pts) This question discusses some defenses against, and variations on, the attack of buffer-overflow stack smashing.

(a) *Reversing the Stack.* When people learn about the stack smashing attack for the first time (such as when $\aleph_1$'s tutorial came out), it often occurs to them to suggest the following defense. On our present systems it's relatively easy for an overflowed buffer to overwrite the return address because the stack grows in the opposite direction as buffers are commonly written to. But if we reversed the direction in which we write the stack, then overflowing the end of a buffer would take you *away* from the location of the return address. Of course this wouldn't be complete protection, because programs can mistakenly write before the beginning of a buffer rather than after the end. But because the return address comes right at the beginning of a stack frame, a procedure could never overwrite its own return address by writing beyond the end of one of its local variables. However there's a more serious limitation of this proposed defense.

Give an example program and attack scenario in which a program's attempt to `strcpy` a long string into a too-short buffer will cause a return address on the stack to be overwritten, 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 through the control flow under the attack, similar to the regular buffer overflow example we discussed in lecture.

(b) Many defenses against stack smashing work by detecting when the return address has been overwritten (like stack canaries), or when the attacker tries to hijack control flow to a new location (like CFI). However there are other ways that a buffer overflow can be used to make a program do the attacker's bidding. Consider the following function from a very simplified payment application:

```
void payment(char *name, double amount_jpy,
             char *purpose, int purpose_len) {
    double amount_usd = amount_jpy / 109.23;
    char memo[32];
    strcpy(memo, "Payment for: ");
    memcpy(memo + strlen(memo), purpose, purpose_len);
    write_check(name, amount_usd, memo);
}
```

Suppose that you as the attacker can control the `purpose` and `purpose_len` arguments, but not `amount_jpy`, on a payment to yourself. (In normal usage, `purpose_len` would be the length of the string pointed to by `purpose`, including a terminating null character.)

Describe how by supplying a carefully crafted `purpose` string, you can increase the amount you get paid, even if stack canaries and CFI are both in use. For concreteness, you can assume a 64-bit platform using IEEE floating point on which local variables are allocated consecutively from higher to lower addresses on the stack in the order they are declared. However, assume that you do not know whether the victim system is little-endian or big-endian, so pick an attack that maximizes your guaranteed return across either little- or big-endian.

**4. Reckless programming.** (20 pts) Let's practice finding bad programming practices that could lead to exploits.

Here's a function that's intended to reverse the order of a subsequence of integers within an array. For instance suppose the array `a` originally contains the integers `1 2 3 4 5 6 7 8 9`. If you call `reverse_range(a, 2, 5)`, then afterwards the array `a` will contain the same integers but with the ones in positions 2 through 5 (counting from zero) in the opposite order. I.e., `a` will be `1 2 6 5 4 3 7 8 9`.

Unfortunately you'll see that this function was not implemented very carefully.

```
/* Reverse the elements from FROM to TO, inclusive */
void reverse_range(int *a, int from, int to) {
    unsigned int *p = &a[from];
    unsigned int *q = &a[to];
    /* Until the pointers move past each other: */
    while (!(p == q + 1 || p == q + 2)) {
        /* Swap *p with *q, without using a temporary variable */
        *p += *q;     /* *p == P + Q */
        *q = *p - *q; /* *q == P + Q - Q = P */
        *p = *p - *q; /* *p == P + Q - P = Q */
        /* Advance pointers towards each other */
        p++;
        q--;
    }
}
```

(a) Describe at least three bad things that could happen when running this function in situations that the programmer probably didn't think of. For each case, identify the programming mistake, the problematic situation, and the bad outcome.

(b) Provide a safer implementation for this function. Note that your new implementation will have to behave differently than the old implementation in some circumstances (probably including, though not necessarily limited to, those situations in which the old one could crash). Think carefully about what behavior would be best, and explain your choices. If your new version has a different interface, explain why this change is needed.

**5. Obscure C behavior.** (15 pts)

(a) With the integers you use in math class, there are only a few pairs you can multiply together to get 18: 1 and 18, -1 and -18, 9 and 2, -9 and -2, 6 and 3, or -6 and -3. However because `int` variables in C have a limited bit width, they behave somewhat differently. Explain how to find, and give, another pair of 32-bit `int`s which multiply together to get 18, even though they wouldn't as mathematical integers. There is a simple example you can write down (e.g., in hex) without any complicated calculation; it may be easiest to see if you think about the fact that multiplying by a power of two is equivalent to shifting left in binary.

(b) With normal integers there isn't any integer you can multiply by 7 to get 18. But again C `int`s are different. Explain how to find, and give, a 32-bit `int` which yields 18 when multiplied by 7. For this it might be easiest to use a computer or at least a calculator for the calculations. You can get the answer, and partial credit, just with a brute-force search through all $2^{32}$ possibilities, but for full credit think of a more clever approach. (If you haven't heard about "modular arithmetic", you might want to look it up. Besides its relevance here, we'll encounter it again later in the course.)

(c) In your previous C programming you've probably already used the formatted output function `printf`, but you may not have used all of its features. For this question, write a `printf` format string that produces the output:

```
$420000000000000000000000 BAREFACED electric 1337
```

when passed the five arguments

```
0, 82, 15707373, "election", 735
```

Your format string should contain 5 conversion specifications (i.e., it should use all five arguments), and be no more than 29 characters long.