CSci 5271: Introduction to Computer Security

Exercise Set 1 **due: Wednesday September 29th, 2021**

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 copy of your solution should be submitted online by 11:59pm on Wednesday, September 29th.

For this assignment, we are going to use Gradescope's free-text online submission style. Therefore, your answers to each question should be plain text. You may find it helpful to write your answers in a separate text editor or word processor, but you will need to cut and paste your answer to each part (10 in total) into boxes on Gradescope.

1. Threat models and risk assessment. (20 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 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. Even if you have to be somewhat imprecise, you should justify these estimates for the various incidence rates and costs.

[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.

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";
```

(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. (20 pts) This question discusses a variation on the attack of buffer-overflow stack smashing.

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_cny,
             char *purpose, int purpose_len) {
    double amount_usd = amount_cny / 6.466;
    char memo[40];
    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_cny`, 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, give separate attacks for the cases when the victim system is little-endian or big-endian, since this may affect your attack's return.

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, within the constraint of keeping the same interface (i.e., function signature). Your new implementation should be a drop-in replacement for the original version in cases where the original version worked sensibly. But it 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).

(c) Lifting the same-interface constraint from the previous part, provide a safer interface and corresponding implementation for this functionality (i.e., a function with a different signature). This implementation should still be usable as a replacement for the original version: explain how code that calls the function should be modified.

5. Obscure C behavior. (20 pts)

(a) With the integers you use in math class, there are only a few pairs you can multiply together to get 20: 1 and 20, -1 and -20, 10 and 2, -10 and -2, 5 and 4, or -5 and -4. 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 20, 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 20. But again C `int`s are different. Explain how to find, and give, a 32-bit `int` which yields 20 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:

```
$100000000000000000000000000 spearheaded evacuation 31337
```

when passed the five arguments

```
0, 104, 962029, "evacuate", 13023
```

Your format string should contain 5 conversion specifications (i.e., it should use all five arguments), and be no more than 30 characters long.