CSci 5271: Introduction to Computer Security

Exercise Set 2 **due: Wednesday February 27th, 2019**

Ground Rules. You may choose to complete these exercises in a group of up to two 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 the course Canvas page by 11:59pm on Wednesday, February 27th.

1. Buffer overflows and invariants. (25 pts) As an exercise in a C programming class, students were asked to implement a certain transformation on character strings. Words inside slashes should have underscores put around each letter `_l_i_k_e_ _t_h_i_s_` to simulate underlining, words inside square brackets should be made upper-case, and words inside curly braces should be ROT13 encrypted. Also the output should end with a space and the word `end` (followed by the usual null terminator). The output of the function is a limited-sized buffer, so some input characters might be discarded, but to avoid causing syntax errors, we want to include the appropriate closing delimiters in the output. Here's an implementation by your friend Eric of that specification. (The function `rot_char`, not shown, implements a Caesar cipher on a single character.)

```
void transform(char *in_buf, char *out_buf, int out_size) {
    char *p = in_buf;
    char *bp = out_buf;
    char *buflim = &out_buf[out_size - 8];
    char c;
    int in_ul, last_ul, rot_amt, skipping;
    int brack_lvl, brace_lvl;
    in_ul = brack_lvl = brace_lvl = last_ul = rot_amt = skipping = 0;

    while ((c = *p++) != '\0') {
        if (brack_lvl > 0)
            c = toupper(c);
        c = rot_char(c, rot_amt);
        if (c == '/')
            in_ul = !in_ul;
        skipping = (bp >= buflim);
        if ((unsigned)c - (unsigned)'[' < 3u && c != '\\') {
            int i = (c & 2) ? 1 : -1;
            if (brack_lvl + i >= 0 && !skipping) {
                brack_lvl += i;
                buflim -= i;
            }
        }
        if (c == '{') {
            if (!skipping) {
                brace_lvl++;
            }
```

```
            rot_amt += 13;
            if (rot_amt == 26) {
                rot_amt = 0;
                buflim -= 2;
            }
        }
        if (c == '}' && brace_lvl > 0) {
            if (!skipping) {
                brace_lvl--;
                buflim++;
            }
            rot_amt -= 13;
            if (rot_amt < 0)
                rot_amt = 0;
        }
        if (in_ul && isalpha(c) && !last_ul && !skipping)
            *bp++ = '_';
        if (c != '/' && !skipping)
            *bp++ = c;
        if (in_ul && isalpha(c)) {
            if (!skipping)
                *bp++ = '_';
            last_ul = 1;
        } else {
            last_ul = 0;
        }
    }
    while (brack_lvl-- > 0)
        *bp++ = ']';
    while (brace_lvl-- > 0)
        *bp++ = '}';
    *bp++ = ' '; *bp++ = 'e'; *bp++ = 'n'; *bp++ = 'd';
    *bp++ = '\0';
}
```

(a) Unfortunately, this code has a buffer overflow bug. Give an example of an input that will cause an overflow if the output buffer is of size 20. (You may find it easier to do either this part of the question or the next part first, or consider working on them together. We've also posted a compilable version of the code on the course web site if you'd like to experiment with it.)

(b) Eric wasn't too far from implementing this function correctly: he recognized that he needed to limit the number of characters written to the output buffer, and he designed mechanisms to try to stop writing when there would not be enough space left. However the logic he implemented for maintaining and checking these limits is not quite correct.

Use invariants to think about how to code this function safely. An invariant for this function

is a relationship between the values of one or more variables that should always hold at a particular point in the program; even better are invariants that always hold, except perhaps in the middle of updating the variables. Formulate some good invariants over the variables of this function. It should be easy to see from your invariants that if the invariants are maintained, the code won't have a buffer overflow. And the invariants should also be related to the variables in a way that explains why the variables change when they do. Because of the bug, your invariants won't all hold in the original version of the code, but suggest a minimal code change that will make the invariants hold, and so make the code safe. If you want to test out your invariants, you can add them as `assert` statements in the code.

2. Error trade-offs with passwords. (15 pts) Many biometric authentication schemes produce a "confidence" value that allows a tradeoff between false positive and false negative errors. Password schemes are not typically considered in this light. List some reasons why you think this might be. We could change the way we check passwords to produce a confidence value; for example, the edit distance between a login attempt and the stored password. This might make passwords easier to use, but it would also have an effect of the security that passwords provide. List some of the security concerns raised by this approach as compared with the standard use of passwords. Make a proposal for how should we choose the best confidence cut-off to balance convenience for users with security against relevant kinds of attacks.

3. Reference monitor without hardware support. (15 pts) Alice is a developer for a toy company. One day her boss Cindy rushes up to her desk excitedly and says "we are going to develop a toy computer with an operating system and everything." Alice is really excited about the prospect of developing an operating system until she finds out that Cindy has already purchased processors that have no access control mechanisms at all: neither a supervisor bit nor a MMU. On the plus side, they are really fast and she has tons of RAM. Alice thinks for a bit longer and decides she can solve this problem in a pretty straightforward way. Sketch out her solution, in enough details to convince a fellow student it will be secure.

4. Sharing files in Unix. (25 pts) Alice wants to be able to share read and write access to some of her files (on a Unix system) with dynamically changing sets of users. Since she is not root, she can't just construct new groups for each file, nor can she turn on the optional ACL feature available on some systems. So she decides to use setuid programs that will implement ACLs for sharing files with her friends. Alice's design calls for two setuid-Alice, world-executable programs (i.e., programs that anyone can run, and which execute with her privileges) named `alice-write` and `alice-read`. She specifies that the programs should operate as follows:

- `alice-write [in] [out]` first checks a permission file written by Alice to make sure that the real uid of the process (that of the calling user) is allowed to write to the file `out`. If so, then the program reads the file `in` and writes it over `out`.
- `alice-read [in] [out]` first checks a permission file written by Alice to make sure that the calling user is allowed to read the file `in`. If so, the the program reads `in` and writes it to the file `out`.

Alice sat in on the first few weeks of 5271, so she also knows to be careful about implementing programs like this. She knows there should be no buffer overflows in `alice-read` and `alice-write`, that the permissions file should be uniquely named in the program and modifiable only by her, and that the programs should only accept files listed by full paths in the permissions file. Before she goes off to hire someone to implement her design, she asks you to critique it.

Point out some remaining security problems with Alice's design. For instance, suppose Bob can read and write some of Alice's files but not others; can he use `alice-write` and `alice-read` to gain access to files he shouldn't? Are there potential attacks that could allow third parties to read/write Alice's files? Does any security-relevant part of Alice's design seem vague or unclear?

To avoid the problems you've identified, suggest design changes to the interface and/or the implementation of `alice-write` and `alice-read`.

5. Multilevel-secure classification. (20 pts) Bob is setting up an MLS operating system for his company. His boss has told him that they will be using a multi-level classification system with three ranks: public $<$ managers $<$ c-level, and one specialized compartment, `hr`. Every user will hold a clearance according to this system.

Suppose Alice has current clearance (managers, $\varnothing$). ($\varnothing$ is the set of specialized compartments she is a member of, namely none.) Draw the lattice of classifications in this system (there are 6 classifications). Mark with an "R" each classification that Alice should be able to read under the BLP policy and with a "W" each classification that Alice should be able to write to under the BLP policy.