CSci 5271: Introduction to Computer Security

Exercise Set 2 **due: Wednesday October 20th, 2021**

Ground Rules. You may choose to complete these exercises in a group of up to three students. Each group should turn in **one** set of answers, designating all group members using the group submission feature of Gradescope. 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, October 20th.

For this assignment, we will use Gradescope's template-based submission style. We will supply a template document with spaces to insert your answers in, in your choice of LaTeX or Google Docs formats. Prepare your answers by filling in this document and converting it to a PDF in a way that preserves the template formatting, and submit this to Gradescope. Note that we still ask that you type your written answers, rather than hand-writing and scanning them, so that they are easy to read. But you can you can use either a drawing program or hand-drawing for the picture in question 5.

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. A heap-related vulnerability. (20 pts) There is something incorrect and unsafe about how the following program uses `malloc` and `free`. First, describe what the bug is. Then, think about how this bug might be useful to an attacker. Suppose that an attacker is supplying the input commands to this program, and their goal is to cause the execution of the function `shellcode`, which stands in for some shellcode that the program wouldn't normally execute. How can an attacker make this happen? Given an example of an input that would be a successful attack of this sort, and explain why it works. You can assume that the `shellcode` function has the address shown in a comment, that the platform is x86-64, and that the `malloc` library reuses the space for objects of the same size in a last-in-first-out order (like a stack). You should be able to answer this question without having to run this code, but if you would like to experiment with a real program, we have supplied the source code on the course web site.

```
typedef void (*toes_func)(void);
void even_toes(void) { printf("with even-toed hoofs"); }
void odd_toes(void)  { printf("with odd-toed hoofs"); }
/* Assume this function has the address 0x4012ce */
void shellcode(void) {
    printf("Uh-oh, this looks like some sort of attack\n");
    exit(42); }

struct herbivore { struct herbivore *next; toes_func func; };
struct herbivore *herbivore_list = 0;
struct carnivore { struct carnivore *next; long num_teeth; };
struct carnivore *carnivore_list = 0;

#define NUM_ANIMALS 256
void *animals_by_num[NUM_ANIMALS]; /* Starts as all null pointers */

void new_herbivore(long x) {
    int loc = x & (NUM_ANIMALS - 1);
    struct herbivore *hp = malloc(sizeof(struct herbivore));
    printf("Allocating herbivore at %p\n", hp);
    if (!hp) exit(1);
    hp->func = ((x & 1) ? odd_toes : even_toes);
    hp->next = herbivore_list; herbivore_list = hp;
    animals_by_num[loc] = hp;
}
void new_carnivore(long x) {
    int loc = x & (NUM_ANIMALS - 1);
    struct carnivore *cp = malloc(sizeof(struct carnivore));
    printf("Allocating carnivore at %p\n", cp);
    if (!cp) exit(1);
    cp->num_teeth = x;
    cp->next = carnivore_list; carnivore_list = cp;
    animals_by_num[loc] = cp;
}
```

```
void release_animal(long x) {
    int loc = x & (NUM_ANIMALS - 1);
    if (!animals_by_num[loc]) {
        fprintf(stderr, "Attempt to release non-existant animal\n");
        exit(1); }
    free(animals_by_num[loc]);
    animals_by_num[loc] = 0;
}
void list_animals(void) {
    struct herbivore *hp;
    struct carnivore *cp;
    for (hp = herbivore_list; hp; hp = hp->next) {
        printf("A herbivore "); (hp->func)(); printf("\n"); }
    for (cp = carnivore_list; cp; cp = cp->next)
        printf("A carnivore with %ld teeth\n", cp->num_teeth);
}
void syntax_error(void) { fprintf(stderr, "Unrecognized syntax\n");
                          exit(1); }

int main(int argc, char **argv) {
    for (;;) {
        int c = getchar();
        long x;
        while (isspace(c)) c = getchar();
        if (c == EOF) return 0;
        switch (c) {
        case 'h':
            if (scanf(" %li", &x) != 1) syntax_error();
            new_herbivore(x); break;
        case 'c':
            if (scanf(" %li", &x) != 1) syntax_error();
            new_carnivore(x); break;
        case 'r':
            if (scanf(" %li", &x) != 1) syntax_error();
            release_animal(x); break;
        case 'l': list_animals(); break;
        case 'q':
            return 0;
        default:
            fprintf(stderr, "Unrecognized command %c\n", c); exit(1);
        }
    }
}
```

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. (20 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.