Assignment 0

Ricardo Andres Calvo Mendez

Problem 1

Exercise A

```
int main(int argc, char **argv) {
    char buf[10];
    strcpy(buf, argv[0]);
    ....
}
```

Analysis

Here, the program copies the command-line argument (`argv[0]` is always the program name) into the buffer `buf` without checking its size.

Failing scenario:

When `strlen(argv[0])` > 10, the buffer overflow occurs.

Exercise B

```
size_t s;
char *p;
scanf("%lu", &s);
p = (char*)malloc(s + 4);
if (p) {
    strcpy(p, "HDR");
    fgets(p + 3, s, stdin);
} else {
    printf("Out of memory!\n");
    return -1;
}
```

Analysis

Here, the function `malloc()` is used to allocate memory of size `s + 4`, where `s` is provided by user input. The program does not validate if `s + 4` overflows the maximum value representable by `size_t`. If an overflow occurs, `malloc()`

would allocate less memory than intended, leading to a buffer overflow when `fgets()` is called.

Failing scenario:

When the user provides as input a value between $2^{64} - 4$ and $2^{64} - 1$ (assuming `size_t` is 64 bits)

Excercise C

```
static void webize(char *str, char *dfstr, int dfsize) {
    char *cp1;
    char *cp2;
    for (cp1 = str, cp2 = dfstr; *cp1 != '\0' && cp2 - dfstr
< dfsize - 1; ++cp1, ++cp2) {
        switch (*cp1) {
        case '<':
            *cp2++ = '&';
            *cp2++ = 'l';
            *cp2++ = 't';
            *cp2 = ';';
            break;
        case '>':
            *cp2++ = '&';
            *cp2++ = 'g';
            *cp2++ = 't';
            *cp2 = ';';
            break;
        default:
            *cp2 = *cp1;
            break;
        }
    }
    *cp2 = '\0';
}
```

Analysis

Here, the function `webize()` copies characters from the input string `str` to the output buffer `dfstr`, replacing `<` and `>` with their HTML entity equivalents. However, the function does not properly account for the increased length of the output string when these replacements occur. Each `<` or `>` character is replaced by a 4-character sequence (`<` or `>`), which can lead to a buffer overflow if there are enough such characters in the input string.

Failing scenario:

Let $a := \texttt{number of '<' and '>' characters in } str$.

When `strlen(str)` $+3a \geq$ `dfsize`, the buffer overflow occurs.

Excercise D

```
int *p;
int q[20];
unsigned s;
...
memset(q, 0, sizeof(q));
...
p = malloc(s);
if (p != NULL){
    memset(p, 'A', sizeof(p));
} else {
    return -1;
}
```

Analysis

Here, The problem is in the second `memset()` call. the return value of `sizeof(p)` is not the size of the allocated memory, it is the size of the pointer itself (constant 8 bytes).

Failing scenario:

When `s` ≤ 8, the buffer overflow occurs.

Exercise E

```
char format[20];
// Read format to display the log string.
scanf("%19s", format);
...
// Print the log_str in required format.
printf(format, log_str);
...
```

Analysis

Here, The problem is the `format` input string, the user can introduce a format string to print out a given amount of bytes, probably exceeding the allocated buffer size for `log_str`.

Failing scenario:

When `format` is intended to print more bytes than `log_str` can hold.

Exercise F

```
class base {
    public:
        base() {
        }
        ~base() {
        }
}

class sub : public base {
    public:
        sub() {
        }
        ~sub() {
        }
}

int main() {
    base *b = new sub();
    ....
    delete b;
}
```

Analysis

Here, There is a parent class `base` and a child class `sub`. The issue arises when a pointer of type `base` is used to delete an object of type `sub`, which can lead to resource leaks because the destructor of the child class is not called.

To fix the problem either `b` should be of type `sub` or the base class destructor should be made virtual.

Exercise G

```
char fl;
....
int ret = sscanf(buf, %s, &fl);
if (ret != 1) {
    printf("Read Error\n");
    return -1;
}
```

Analysis

Here, the problem is that the size of `fl` is not being checked before using it in `sscanf`. This can lead to buffer overflows if the input exceeds the allocated size.

Failing scenario:

When `fl` points to a buffer smaller than the input being read.

Problem 2

A. If we avoid storing return address on runtime stack then stack-based buffer overflows do not cause any security issues (especially, control-flow hijacking)

Answer

No, In general that no prevent ANY security issues, It prevents only control-flow hijacking attacks because that addresses cannot be used to redirect execution flow.

B. We can always prove that a given program does not have any security vulnerabilities.

Answer

As demonstrated in class, the answer to this question is the same as the halting problem: it is undecidable in general.

C. Exhaustive testing proves that the a given program does not have any bugs.

Answer

No, In general that falls in the halting problem again, you can prove most of the inputs but not all of them.

Problem 3

A. A process can know physical addresses of its virtual addresses. Justify your answer in either case.

Answer

No, the virtual address is converted to physical address by Operating system. The program does not have knowledge of physical addresses. Only kernel-like processes can access physical addresses directly.

B. A process can read and write memory that belong to the operating system kernel. Justify your answer in either case.

Answer

No, the operating system prevents processes from accessing memory that does not belong to them. The typical error that occurs when a process tries to access memory outside its allocated space is a segmentation fault, which results in the termination of the process.

C. Operating system should always sanitize (i.e., verify) addresses given by a user process. Why? E.g., Destination address provided for read/write syscall.

Answer

Yes, the operating system should always ensure that user programs operate within their allocated memory space and do not interfere with each other's memory.

D. Is there any security issue in the following code? Justify your answer in either case.

```
unsigned gl;
char flag_buf[4];
...
unsigned i;
if (!copy_from_user(&i, buf, sizeof(i))) {
    if (i<4) {
        if (!copy_from_user(&gl, buf, sizeof(gl))) {
            flag_buf[gl] = 0;
        }
    }
}
...
```

Answer

Yes, a buffer overflow can occur when it is accessing using `gl` as index `flag_buf[gl] = 0`. `gl` is a value read from the user (I assume `copy_from_user()` is used to read this value from a human). the value of `gl` is not being validated