CS 88: Week 2: Class 4: Buffer Overflow Attacks

Q1. The threat model is victim code handling input that comes from across a security boundary. What are some examples of this that you can provide? (Here are some examples to get you started.)

- A computer login screen that accepts a username and password typed by the attacker.
- An email server or email clients accepting, storing, and rendering email messages from an attacker.

Q2. A Buffer Overflow can exploit...

A. pointer assignment & memory allocation, de-allocation
B. function pointers
C. calls to library routines
D. general purpose registers
E. format strings

Q3. Consider the following code:

```
char buf[8];
int authenticated = 0;
void vulnerable() {
    gets(buf);
}
```

Note that both char buf[8] and authenticated are defined outside of the function, so they are both located in the static part of memory. In C, static memory is filled in the order that variables are defined, so `authenticated` is at a higher address in memory than buf (since static memory grows downward and buf was defined first, buf is at a lower memory address).

Imagine that elsewhere in the code, there is a login routine that sets the authenticated flag only if the user proves knowledge of the password. What damage could you do?

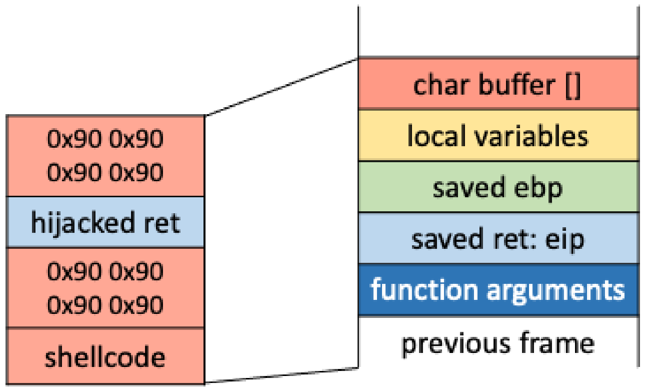
Let's consider a modification of the above code:

```
char buf[8];
int (*fnptr)();
void vulnerable() {
    gets(buf);
}
```
`fnptr` is a *function pointer*. In memory, this is a 4-byte value that stores the address of a function. In other words, calling `fnptr` will cause the program to dereference the pointer and start executing instructions at that address.

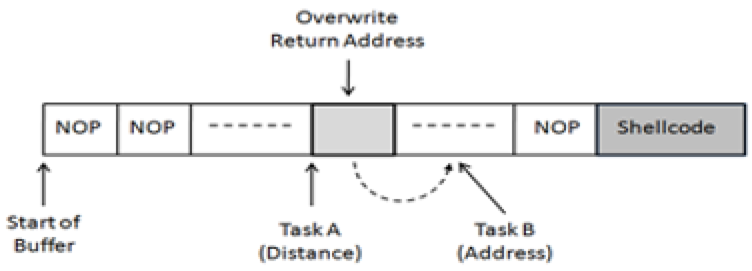
Like `authenticated` in the previous example, `fnptr` is stored directly above `buf` in memory. Suppose the function pointer `fnptr` is called elsewhere in the program (not shown). What possible damage can you do now?

Q4. Draw out a stack diagram and build your very own shellcode attack sandwich.
Information you are given:

- buffer to overflow:
  - char buffer[100]
  - &buffer[0] = 0xffffd88c
  - $eip = 0xffffd8bc
  - shellcode = 20 bytes



Task A: Figure out the distance from the start of the buffer to the saved eip value.
Task B: Figure out where you want to point your saved eip to, in the NOP sled you've created.



Q5. We've seen that the cause of the vulnerability is often no range checking (i.e., string functions in C do not check input size). Assess whether the following range checking will help:

- Potential overflow in htpasswd.c (Apache 1.3):

```
… strcpy(record,user);
  strcat(record,":");
  strcat(record,cpw); …
```

Copies username ("user") into buffer ("record"), then appends ":" and hashed password ("cpw")

- Published fix:

```
strncpy(record,user,MAX_STRING_LEN-1);
strcat(record,":");
strncat(record,cpw,MAX_STRING_LEN-1); …
```

A. The fix ensures that there are no vulnerabilities
B. The vulnerabilities are still present.

Q6. Now consider the following code. Do you think it is free from integer overflow vulnerabilities?

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]){
        unsigned short s;
        int i;
        char buf[80];

        if(argc < 3){
                return -1;
        }
```

```
        i = atoi(argv[1]);
        s = i;

        if(s >= 80) {          /* [w1] */
                printf("Oh no you don't!\n");
                return -1;
        }

        printf("s = %d\n", s);

        memcpy(buf, argv[2], i);
        buf[i] = '\0';
        printf("%s\n", buf);

        return 0;
}
```

A) This code is free from integer overflow vulnerabilities.
B) Integer vulnerabilities still exist.

Q7.

What’s wrong with this code?

```
#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable()
{
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if(len > BUF_SIZE) {
        printf(“Too large\n”);
        return;
    }
    memcpy(buf, p, len);
}
```

```
void *memcpy(void *dest, const void *src, size_t n);
            typedef unsigned int size_t;
```

A. Nothing
B. Buffer overflow
C. Integer overflow
D. Race Condition