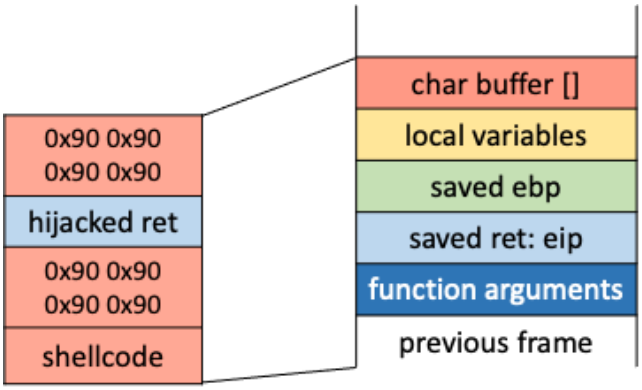
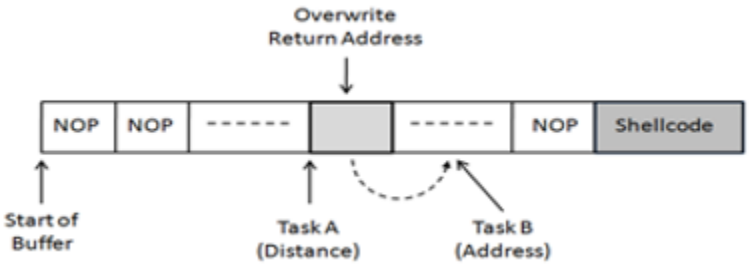
CS 88: Week 2: Class 5: Software Attacks

Q1. Draw out a stack diagram and build your very own shellcode attack sandwich.
Information you are given:

- buffer to overflow:
  - `char buffer[50]`
  - `&buffer[0] = 0xffffd88c`
  - `Saved 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.



Q2. 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); …
```

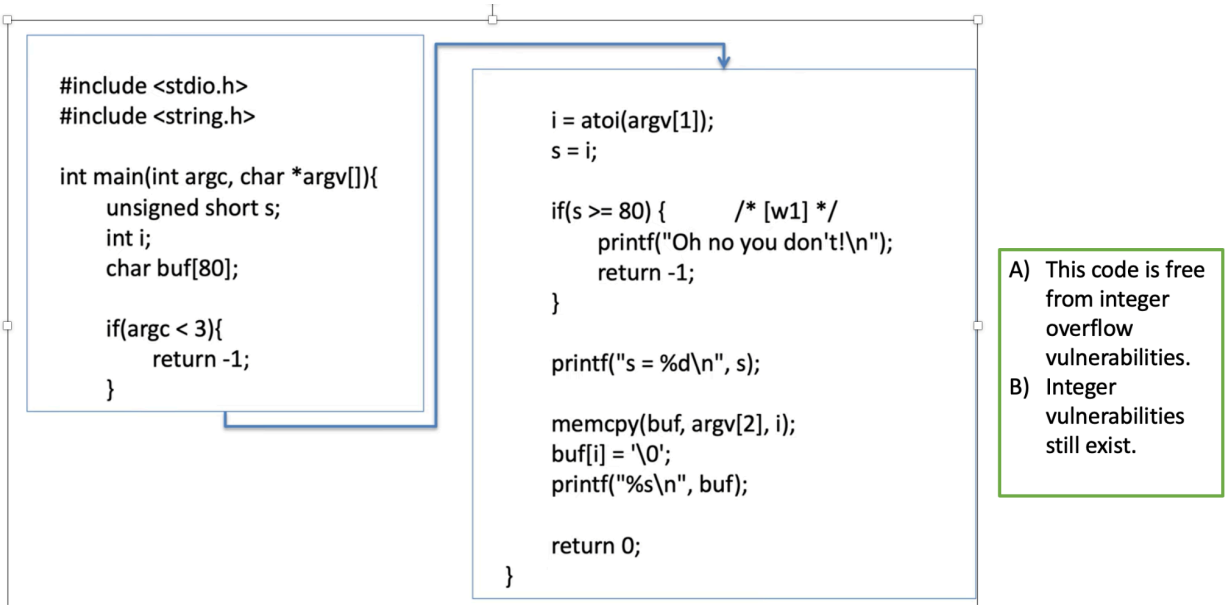
Copies username ("user") into buffer ("record"), then appends ":" and hashed password ("cpw")

- **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.

Q3. 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.

Q3.

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;
```

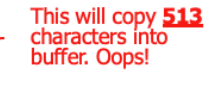
A. Nothing
B. Buffer overflow
C. Integer overflow
D. Race Condition

Off-By-One Overflow

Home-brewed range-checking string copy

```
void notSoSafeCopy(char *input) {
        char buffer[512]; int i;

    for (i=0; i<=512; i++)
        buffer[i] = input[i];
}
void main(int argc, char *argv[]) {
    if (argc==2)
       notSoSafeCopy(argv[1]);
}
```



A vulnerable password checking program

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

int main (int argc, char *argv[]){
     int allow_login = 0;
     char pwdstr[12];
     char targetpwd[12] = “mypwd123”;
     gets(pwdstr);
     if (strncmp(pwdstr, targetpwd, 12) == 0)
          allow_login = 1;
     if (allow_login == 0)
          printf(“Login request rejected”);
     else
          printf(“Login request allowed”);
}
```

Put check marks next to the lines of code that access addresses on main’s stack frame

What vulnerabilities are present in the code? (list at least 3)

How we safeguard against buffer overflows as a software engineer?

A. Make buffers (slightly) longer than necessary

B. Safe string manipulation functions (other checks we can do?)

C. Don't write in C. It's the root of all evil!

D. As a software programmer there's only so much we can do… there's no fix.