

CSE 484 / CSE M 584: Buffer Overflows (continued)

Fall 2022

Franziska (Franzi) Roesner
franzi@cs

Announcements

- Things Due:
 - Homework #1: Due Friday
 - Research Readings (CSE M 584): Due Thursday(s)
- TA office hours scheduled
 - Tue 3:30-4:30pm, CSE (Allen) 218 - Aroosh and Basia
 - Wed 3:30-4:30pm, Zoom (see Canvas for link) - David, Tim, and Wenqing
 - Thurs 10:30-11:30am, CSE (Allen) 218 - Noah and William
 - Fri 2:00-3:00pm, CSE2 (Gates) 152 - Julia and Theo
- Lab 1 out shortly
 - I recommend you tr to create accounts before section tomorrow

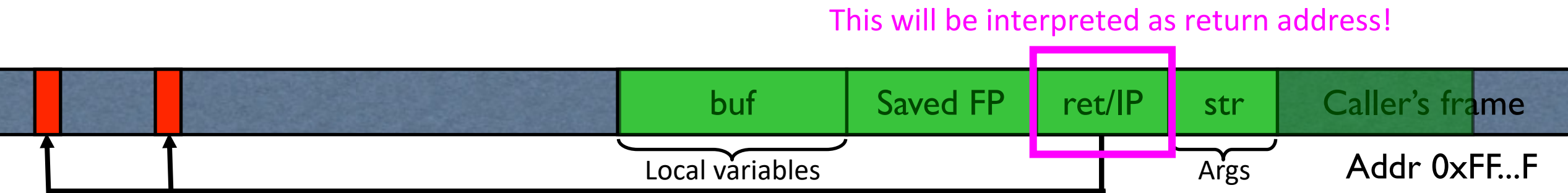
Last Time: “Standard” Buffer Overflows

- Memory pointed to by str is copied onto stack...

```
void foo(char *str) {  
    char buf[126];  
    strcpy(buf, str);  
}
```

strcpy does NOT check whether the string at *str contains fewer than 126 characters

- If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations.

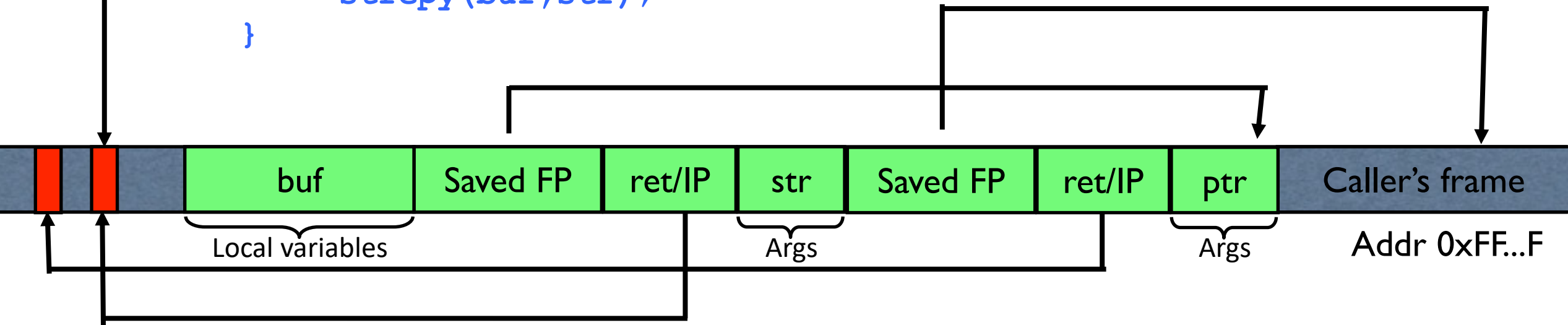


Review: Stack Buffers – bar() calls foo()

```
void bar(char *ptr) {  
    func(ptr);  
    ptr++;  
}  
void foo(char *str) {  
    char buf[126];  
    strcpy(buf, str);  
}
```

Note: Toy example, functions not useful

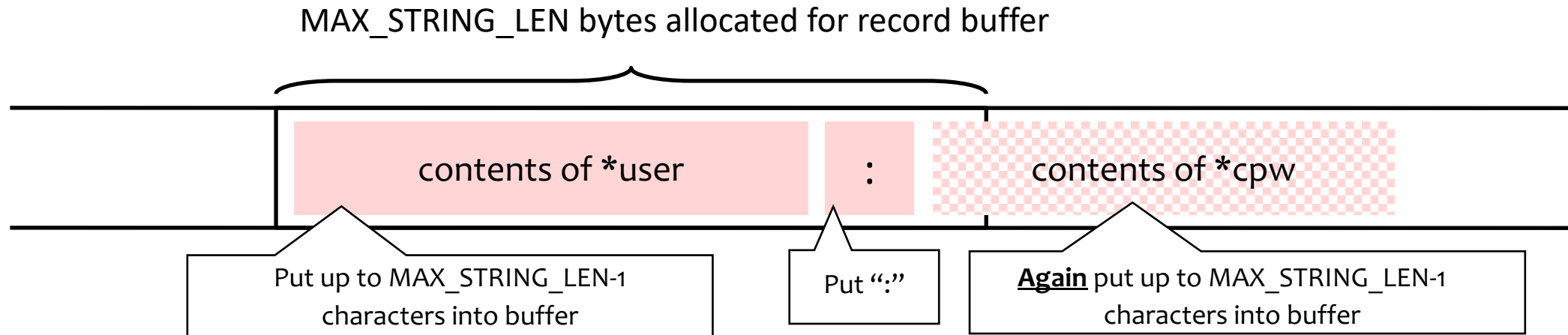
Note: Exact pointer locations may vary by architecture; this description focuses on high-level ideas



Review: Misuse of strncpy in httpasswd “Fix”

- Published “fix” for Apache httpasswd overflow:

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



What About This?

- Home-brewed range-checking string copy

```
void mycopy(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)
        mycopy(argv[1]);
}
```

Off-by-One Overflow

- Home-brewed range-checking string copy

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

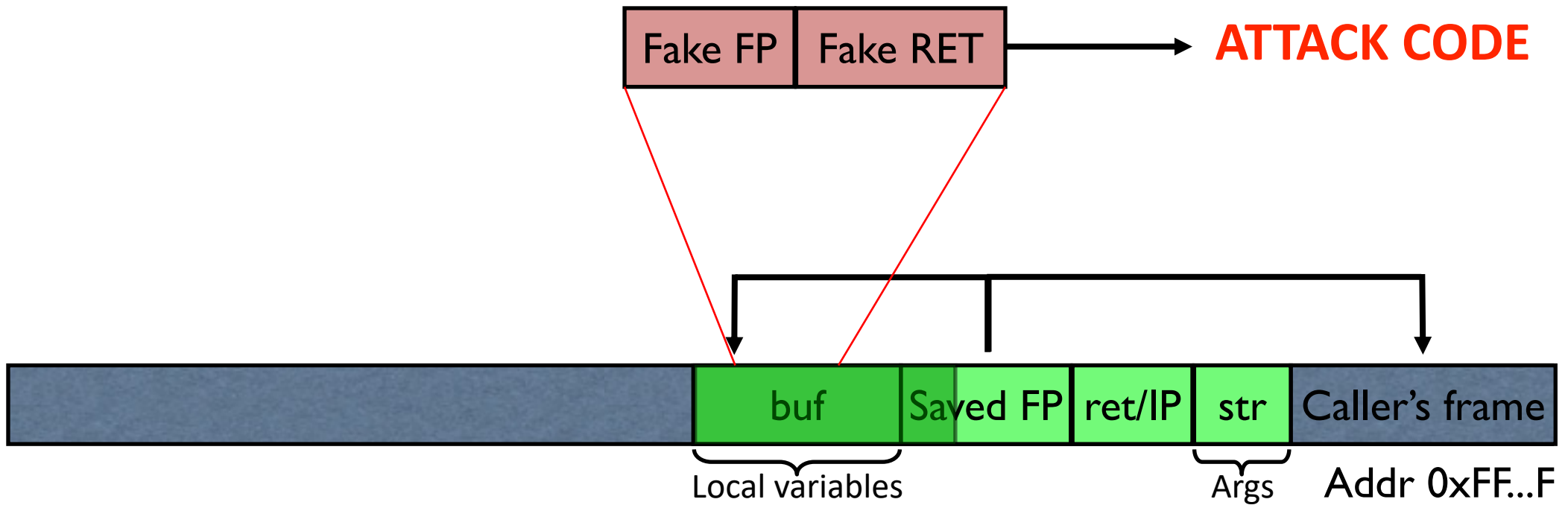
    for (i=0; i<=512; i++)
        buffer[i] = input[i];
}

void main(int argc, char *argv[]) {
    if (argc==2)
        mycopy(argv[1]);
}
```

This will copy **513** characters into buffer. Oops!

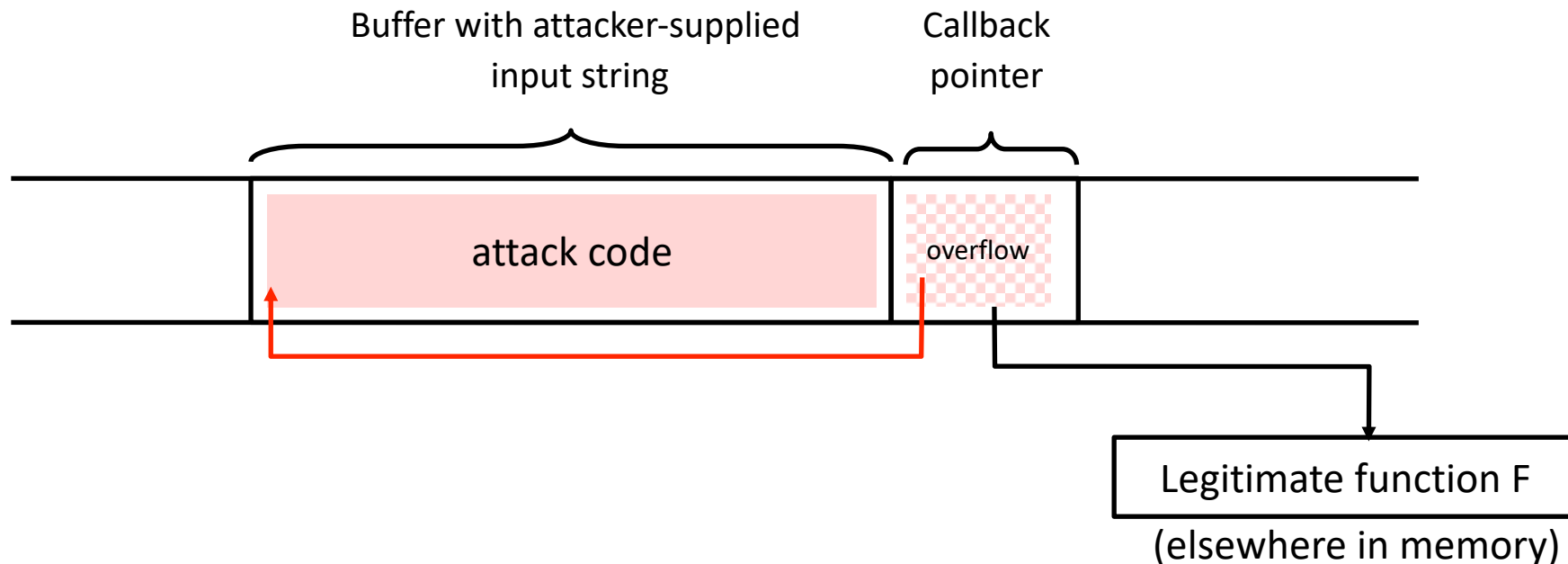
- 1-byte overflow: can't change RET, but can change pointer to previous stack frame...

Frame Pointer Overflow



Another Variant: Function Pointer Overflow

- C uses **function pointers** for callbacks: if pointer to F is stored in memory location P, then one can call F as $(*P)(\dots)$



Other Overflow Targets

- Format strings in C
 - We'll walk through this one next
- Heap management structures used by malloc()
 - More details in section
 - Techniques have changed wildly over time
- These are all attacks you can look forward to in Lab #1 😊

Variable Arguments in C

- In C, can define a function with a variable number of arguments
 - Example: `void printf(const char* format, ...)`
- Examples of usage:

```
printf("hello, world");  
printf("length of '%s' = %d\n", str, str.length());  
printf("unable to open file descriptor %d\n", fd);
```

Format specification encoded by special % characters:

- `%d,%i,%o,%u,%x,%X` – integer argument
- `%s` – string argument
- `%p` – pointer argument (`void *`)
- Several others

Format Strings in C

- Proper use of printf format string:

```
int foo = 1234;  
printf("foo = %d in decimal, %X in hex", foo, foo);
```

This will print:

```
foo = 1234 in decimal, 4D2 in hex
```

- Unsafe use of printf format string:

```
char buf[14] = "Hello, world!";  
printf(buf);  
// should've used printf("%s", buf);
```

What happens if buffer contains format symbols starting with % ???


(Simplified) Implementation of Variable Args

- Special functions `va_start`, `va_arg`, `va_end` compute arguments at run-time

```
void printf(const char* format, ...)
{
    int i; char c; char* s; double d;
    va_list ap; /* declare an "argument pointer" to a variable arg list */
    va_start(ap, format); /* initialize arg pointer using last known arg */

    for (char* p = format; *p != '\\0'; p++) {
        if (*p == '%') {
            switch (*++p) {
                case 'd':
                    i = va_arg(ap, int); break;
                case 's':
                    s = va_arg(ap, char*); break;
                case 'c':
                    c = va_arg(ap, char); break;
            }
            ... /* etc. for each % specification */
        }
    }
    ...

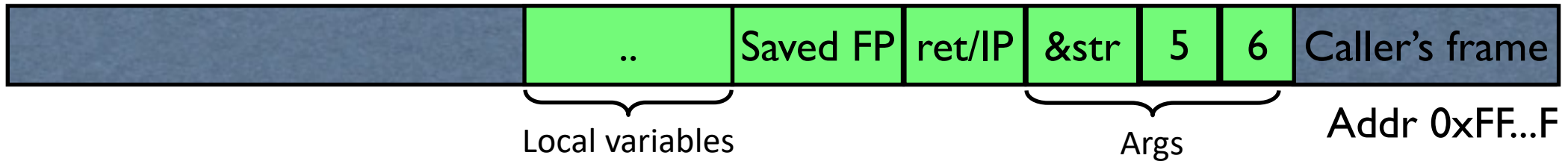
    va_end(ap); /* restore any special stack manipulations */
}
```



Closer Look at the Stack

```
printf("Numbers: %d,%d", 5, 6);
```

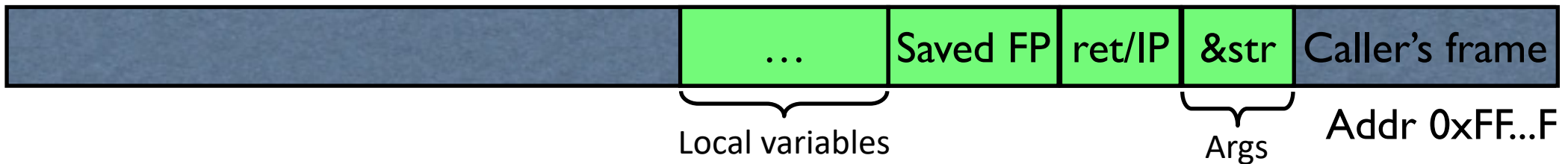
Printf's internal stack pointer starts here



```
printf("Numbers: %d,%d");
```



Printf's internal stack pointer starts here



Format Strings in C

- If the buffer contains format symbols starting with %, the location pointed to by printf's internal stack pointer will be interpreted as an argument of printf.

This can be exploited to move printf's internal stack pointer!

- Unsafe use of printf format string:

```
char buf[14] = "Hello, world!";  
printf(buf);  
// should've used printf("%s", buf);
```

What happens if buffer contains format symbols starting with % ???

Viewing Memory

- `%x` format symbol tells printf to output data on stack

```
printf("Here is an int:  %x",i);
```

- What if printf does not have an argument?

```
char buf[16]="Here is an int:  %x";  
printf(buf);
```

- Or what about:

```
char buf[16]="Here is a string:  %s";  
printf(buf);
```


Viewing Memory

- `%x` format symbol tells printf to output data on stack

```
printf("Here is an int:  %x",i);
```

- What if printf does not have an argument?

```
char buf[16]="Here is an int:  %x";  
printf(buf);
```

- Stack location pointed to by printf's internal stack pointer will be interpreted as an int. (What if crypto key, password, ...?)

- Or what about:

```
char buf[16]="Here is a string:  %s";  
printf(buf);
```

- Stack location pointed to by printf's internal stack pointer will be interpreted as a pointer to a string

Writing Stack with Format Strings

- `%n` format symbol tells `printf` to write the number of characters that have been printed

```
printf("Overflow this!%n", &myVar);
```

- Argument of `printf` is interpreted as destination address
 - This writes `14` into `myVar` (“Overflow this!” has 14 characters)
- What if `printf` does not have an argument?

```
char buf[16]="Overflow this!%n";  
printf(buf);
```

- Stack location pointed to by `printf`'s internal stack pointer will be **interpreted as address** into which the number of characters will be written.

Summary of Printf Risks

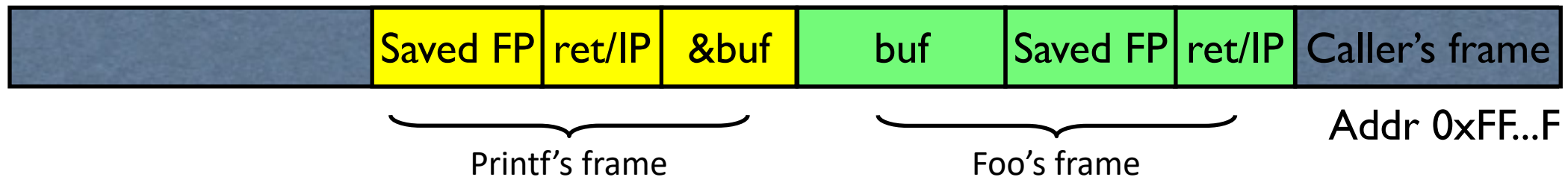
- Printf takes a variable number of arguments
 - E.g., `printf("Here's an int: %d", 10);`
- Assumptions about input can lead to trouble
 - E.g., `printf(buf)` when `buf="Hello world"` versus when `buf="Hello world %d"`
 - Can be used to advance printf's internal stack pointer
 - Can read memory
 - E.g., `printf("%x")` will print in hex format whatever printf's internal stack pointer is pointing to at the time
 - Can write memory
 - E.g., `printf("Hello%n");` will write "5" to the memory location specified by whatever printf's internal SP is pointing to at the time

How Can We Attack This?

```
foo() {  
    char buf[...];  
    strncpy(buf, readUntrustedInput(), sizeof(buf));  
    printf(buf); //vulnerable  
}
```

Note: Different compilers / compiler options / architectures might vary

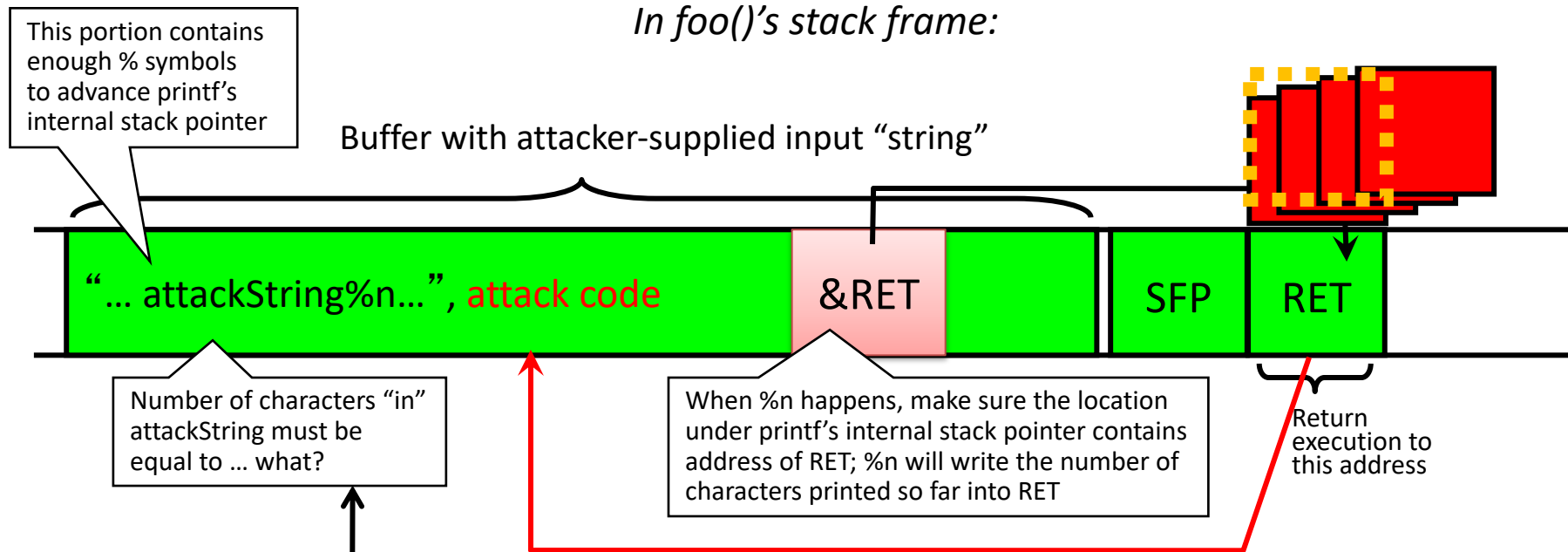
If format string contains % then printf will expect to find arguments here...



What should the string returned by readUntrustedInput() contain??

Canvas -> Quizzes -> Oct 7

Using %n to Overwrite Return Address



Why is "in" in quotes? C allows you to concisely specify the "width" to print, causing printf to pad by printing additional blank characters without reading anything else off the stack.

Example: `printf("%5d%n", 10)` will print three spaces followed by the integer: " 10"

That is, the %n will write 5, not 2.

Key idea: do this 4 times with the right numbers to overwrite the return address byte-by-byte. (4x %n to write into &RET, &RET+1, &RET+2, &RET+3)

Recommended Reading

- It will be hard to do Lab 1 without:
 - Reading (see course schedule):
 - Smashing the Stack for Fun and Profit
 - Exploiting Format String Vulnerabilities
 - Attending section this week and next