CSE 484 / CSE M 584: Computer Security and Privacy

Software Security: Buffer Overflow Attacks

(continued)

Spring 2020

Franziska (Franzi) Roesner
franzi@cs.washington.edu

Thanks to Dan Boneh, Dieter Gollmann, Dan Halperin, Yoshi Kohno, Ada Lerner, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials ...

Announcements

- Participation and Breakout Groups
  - We'll be using in-class activities for participation (see email)
  - Sign up via Canvas if you'd like a specific breakout group
  - Also using Canvas groups for assignment groups (new group set per assignment, to support changing groups)
- TA Office Hours
  - See course website; Zoom links on Canvas
- Lab 1
  - **Group signup instructions will be released today (SSH)**
  - Lab access granted starting mid-week
  - Checkpoint (4/17) and Final (4/29) deadlines
- Feedback re: online course logistics? Survey sent Friday

Last Time: Basic Buffer Overflows

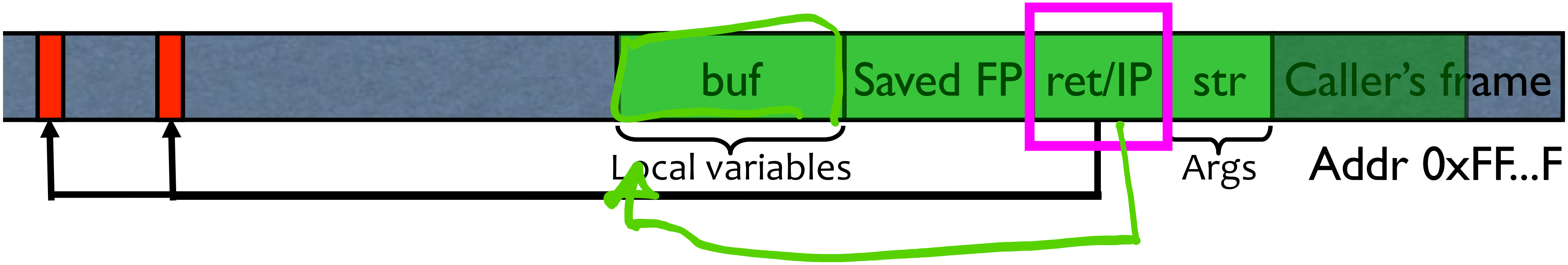
- Memory pointed to by str is copied onto stack…

```
void func(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.

This will be interpreted as return address!



What About This?

- Home-brewed range-checking string copy

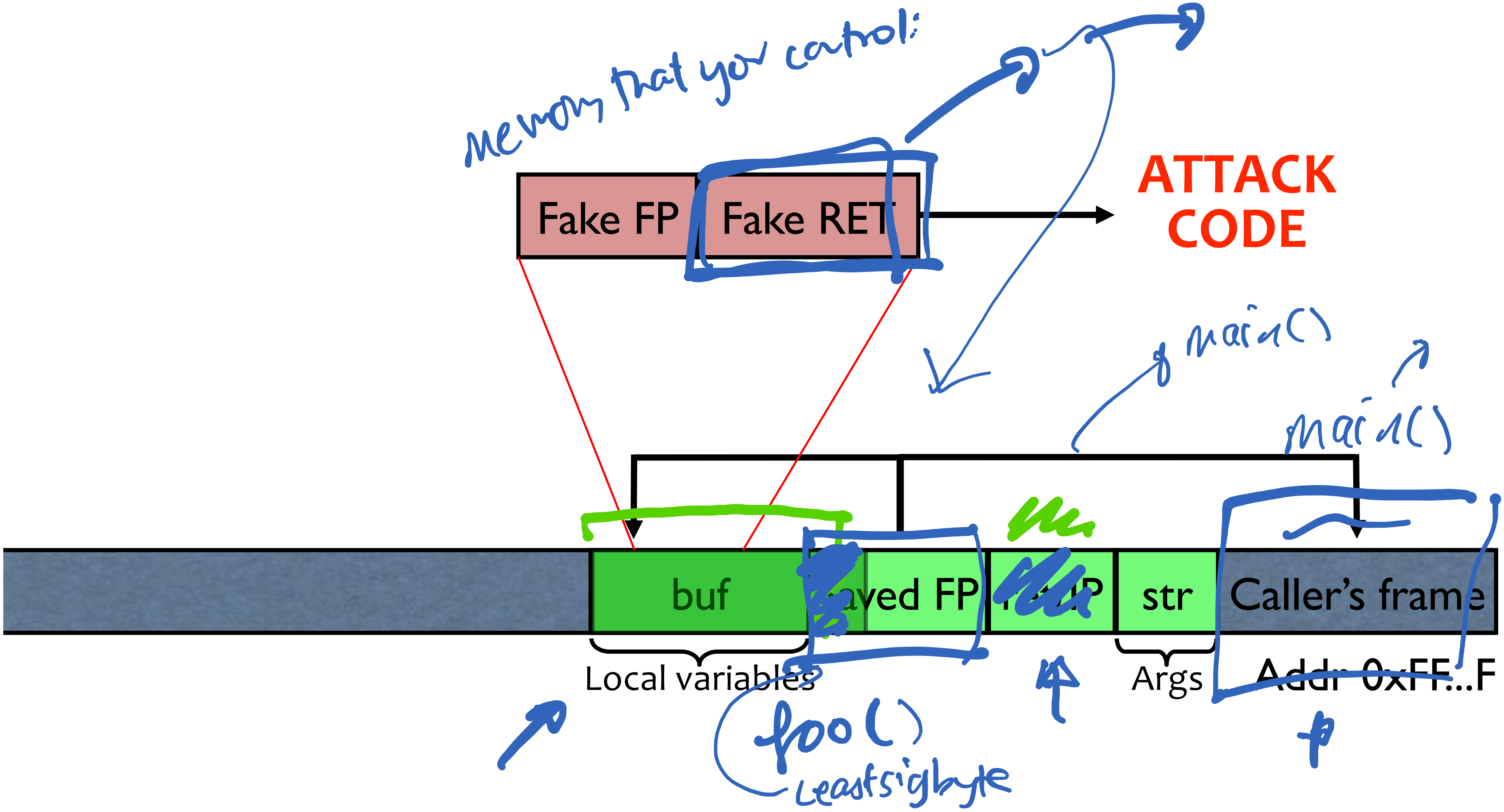
```
void mycopy(char *input) {
     char buffer[512]; int i;

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

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

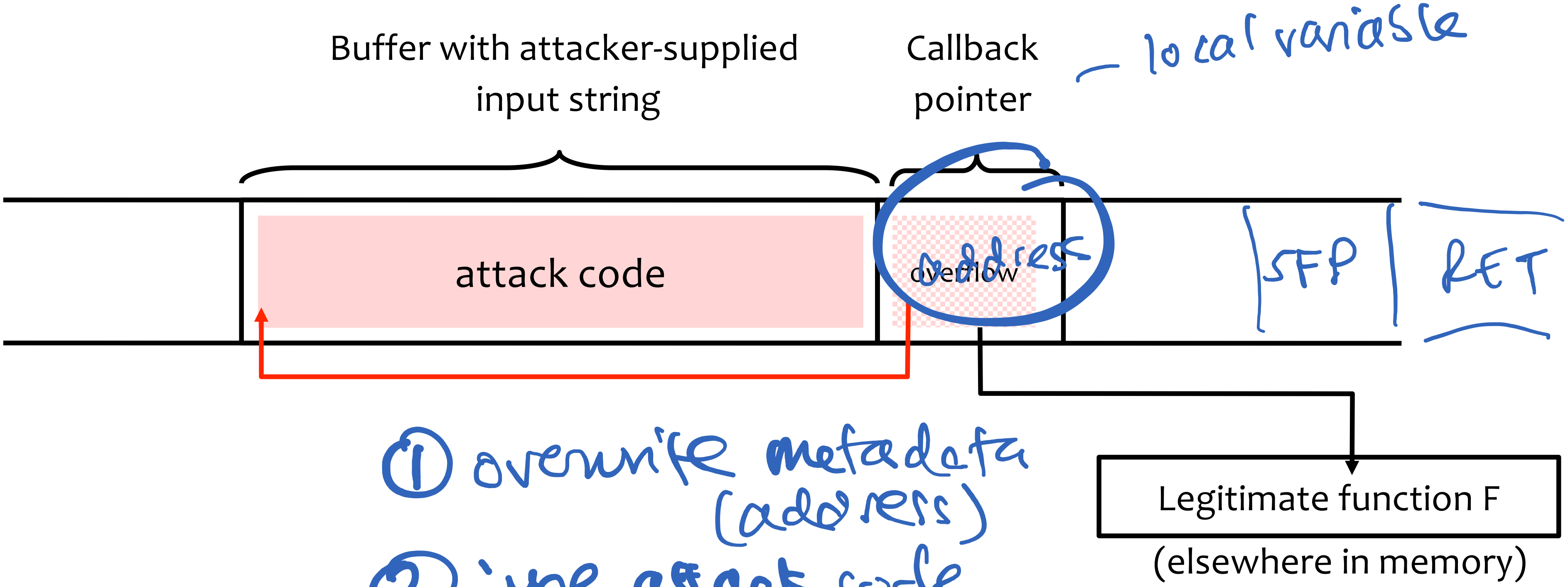
- 1-byte overflow: can't change RET, but can change pointer to previous stack frame…

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)(…)



Buffer with attacker-supplied input string

Callback pointer

attack code

overflow

Legitimate function F

(elsewhere in memory)

① overwrite metadata (address)

② jump attack code

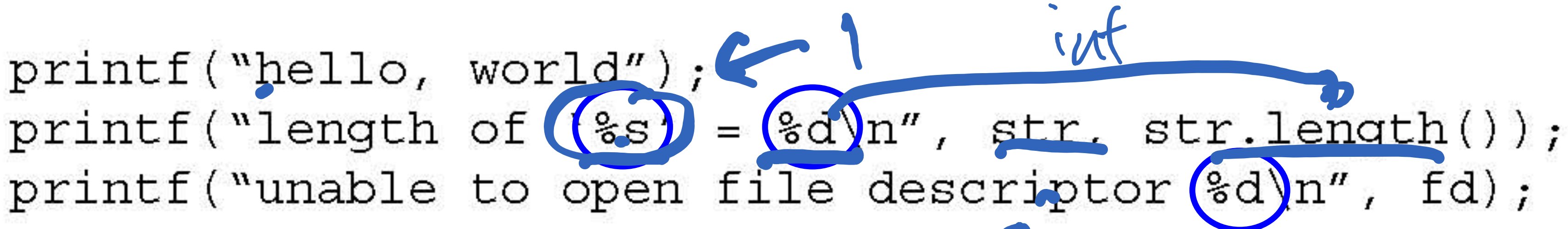
Other Overflow Targets

- Format strings in C
  – More details today
- Heap management structures used by malloc()
  – More details in section

- 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);
```

1

int

Format specification encoded by special % characters

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

string

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
```

- Sloppy use of printf format string:

```
char buf[14] = “Hello, world!”;
printf(buf); ← assuming it's just a string
// should’ve used printf(“%s”, buf);
```

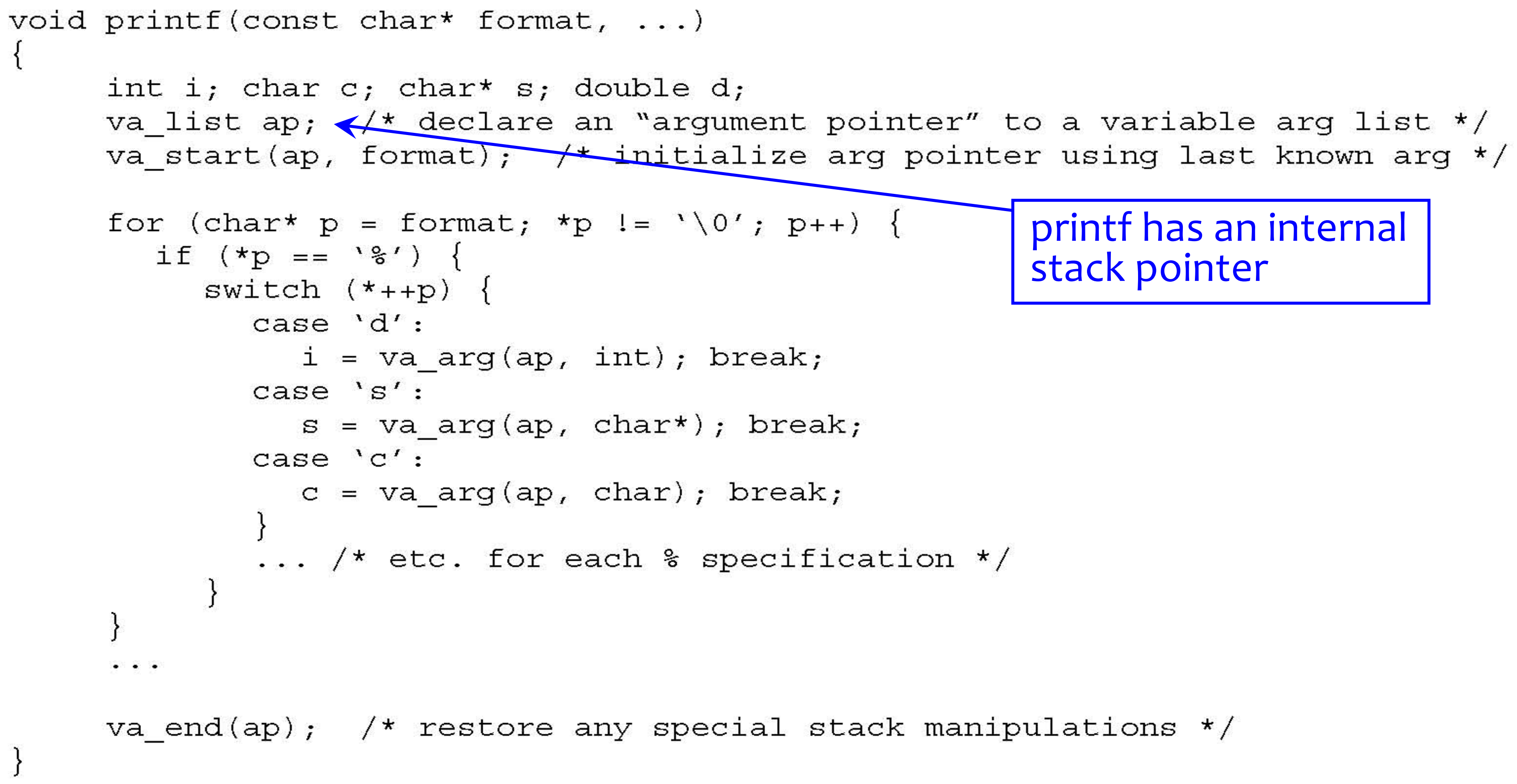
from network

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

“Hello %x”

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 */
}
```

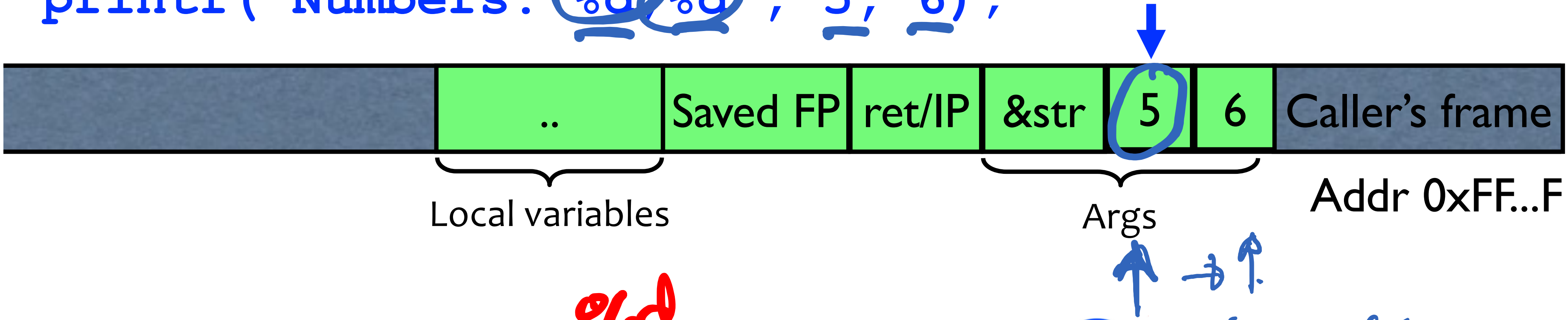
printf has an internal stack pointer

Closer Look at the Stack

null terminated

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

Internal stack pointer starts here

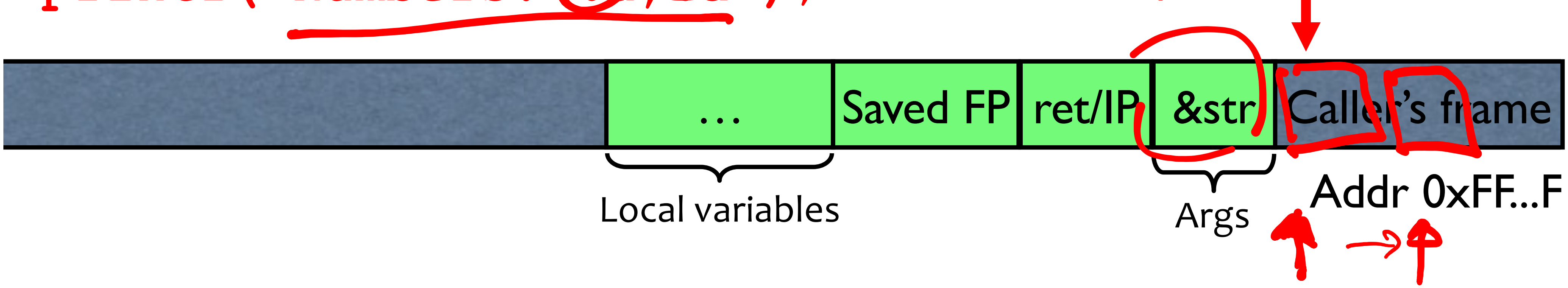


%d

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

Internal stack pointer starts here

internal stack p.



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!

- Sloppy 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

interpreted as address

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

14

myVar

  - Argument of printf is interpeted as destination address
  - This writes 14 into myVar ("Overflow this!" has 14 characters)

- What if printf does not have an argument? %d 5

```
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.

?

internal sp

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

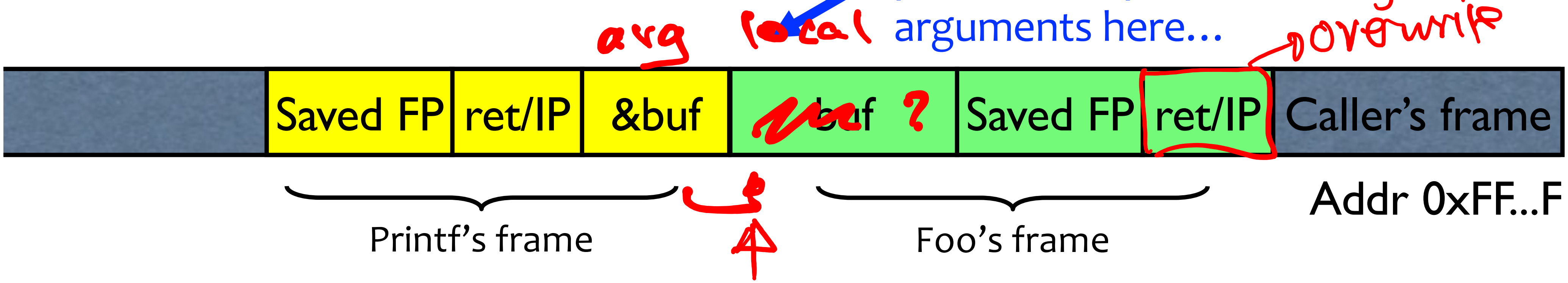
How Can We Attack This?

attacker controls

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

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

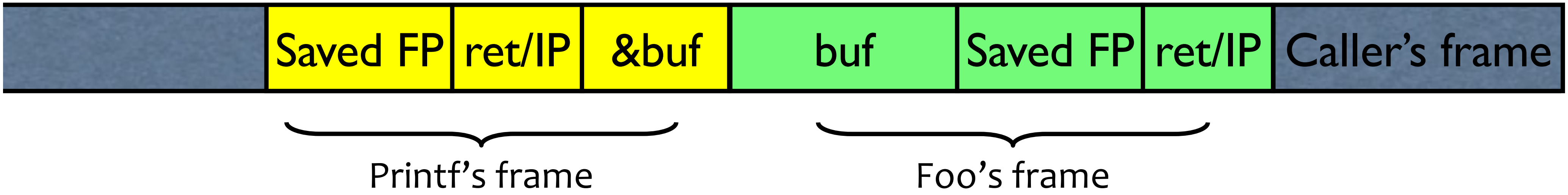
goal: overwrite



start looking here

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

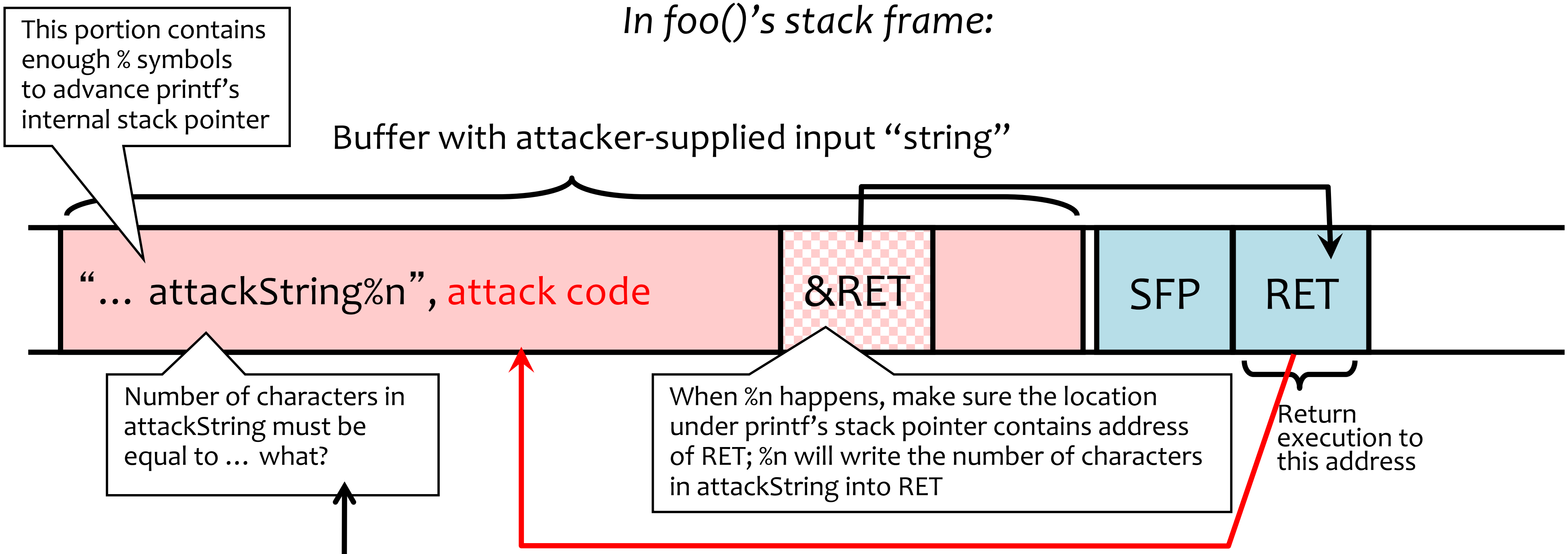
Go to Canvas Quiz for April 6!



Saved FP
ret/IP
&buf
buf
Saved FP
ret/IP
Caller's frame
Printf's frame
Foo's frame

Using %n to Overwrite Return Address

In foo()'s stack frame:



This portion contains enough % symbols to advance printf's internal stack pointer

Buffer with attacker-supplied input "string"

"… attackString%n", attack code

&RET

SFP

RET

Number of characters in attackString must be equal to … what?

When %n happens, make sure the location under printf's stack pointer contains address of RET; %n will write the number of characters in attackString into RET

Return execution to this address

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", 10) will print three spaces followed by the integer: "   10"

That is, %n will print 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