CSE 484 / CSE M 584: Computer Security and Privacy

Software Security: Buffer Overflow Attacks

(continued)

Autumn 2018

Tadayoshi (Yoshi) Kohno
yoshi@cs.Washington.edu

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

Admin

- Lab 1 out, discussed in quiz section
- Thanksgiving: no class Wednesday
  - Alternate video assignment

- Looking forward
  - Today + Next week: More buffer overflows + defenses
  - Next week: Transition to crypto

Wednesday: Peter Ney

- Glimpse at some aspects of computer security research
  - Measurement
  - Analysis / attack exploration
- Other types of research
  - Building secure systems
- Also connected to threat modeling, and to buffer overflows

Last Time: Basic Buffer Overflows

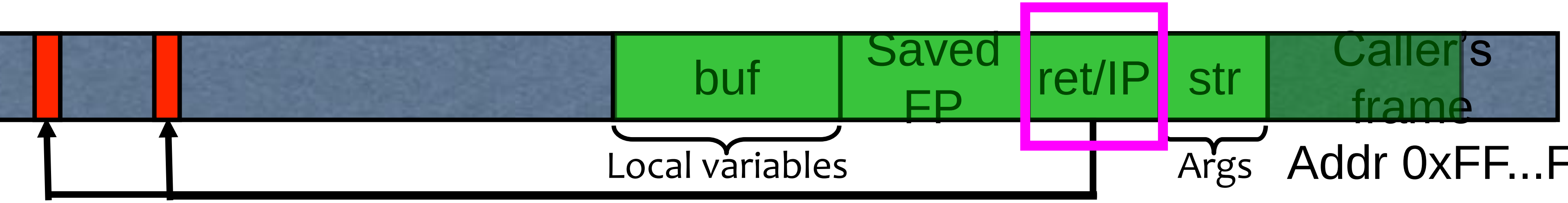
- Memory pointed to by str is copied onto stack…

```
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!



Off-By-One Overflow

- Home-brewed range-checking string copy

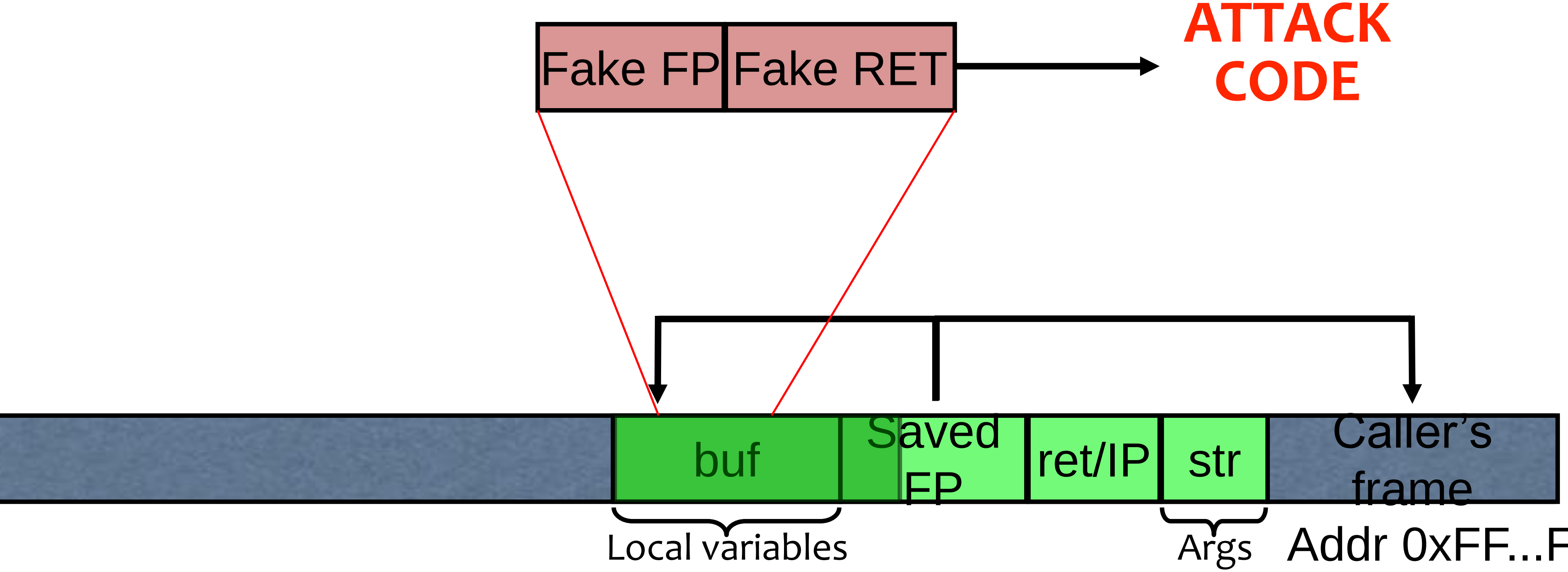
```
 void mycopy(char *input) {
       char buffer[512]; int i;

       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



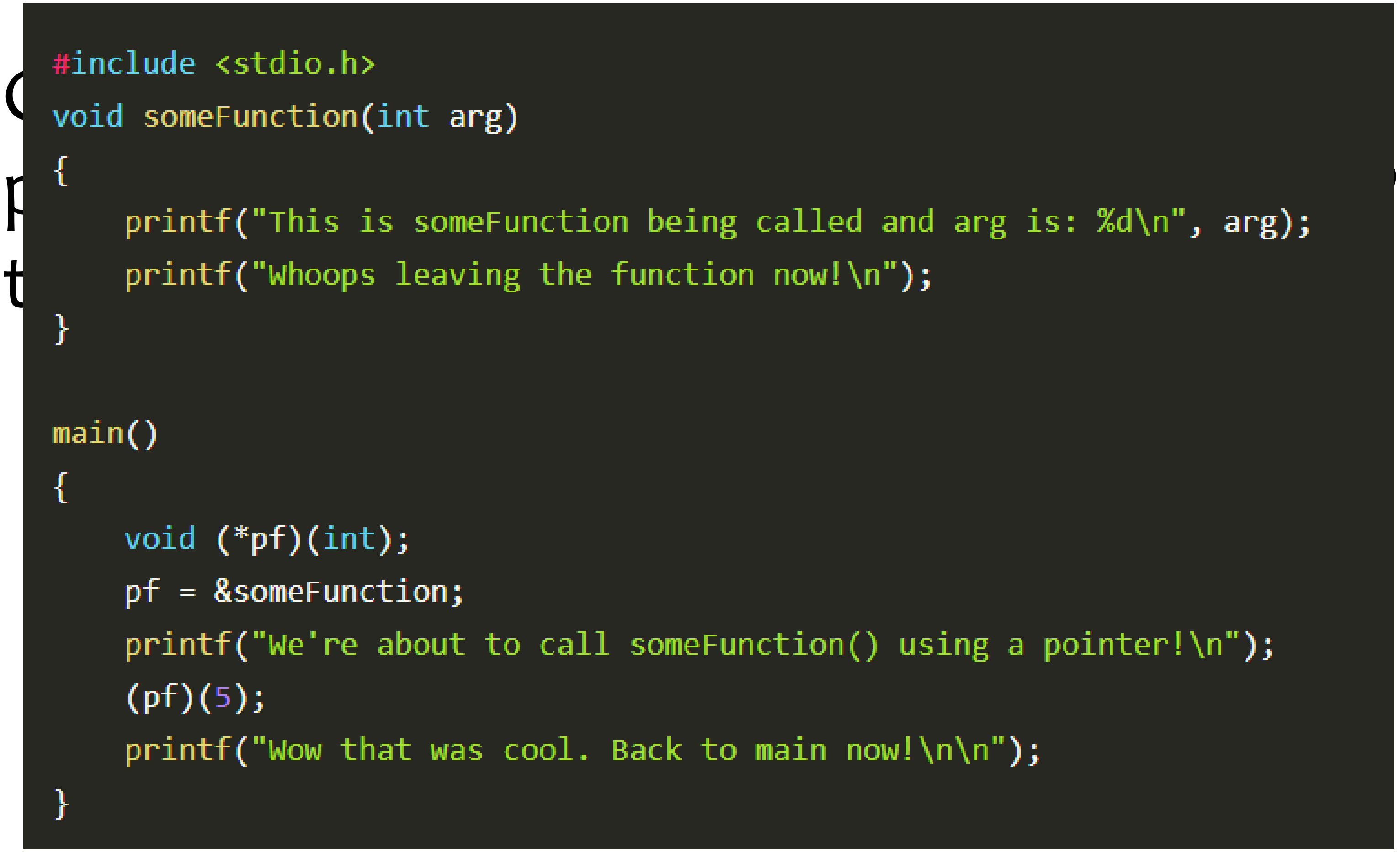
Stepping Back

- This class: Broad tour of key concepts in security
  - Key principles
  - Foundations / historical perspective
  - Lab 1 doesn't have all modern defenses / compiler options enabled
- But you'll still experiment with other variants
  - E.g., one target in lab 1 doesn't save frame pointer on stack

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)(…)

Another Variant: Function Pointer Overflow



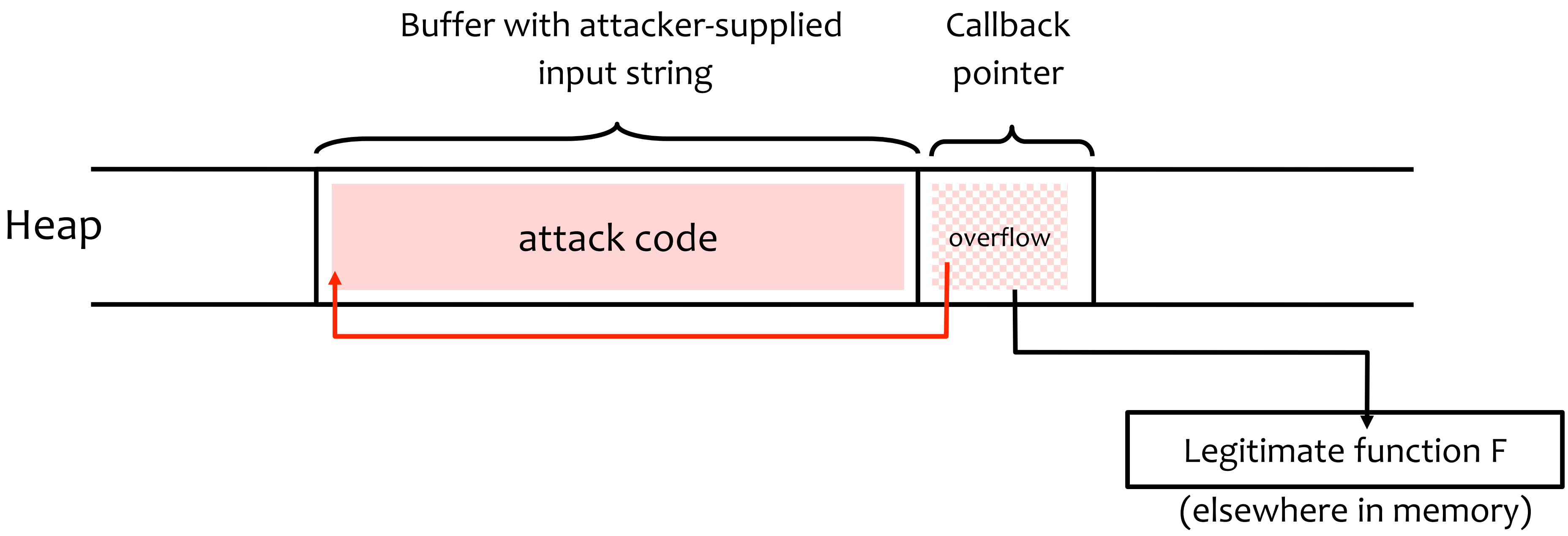
```
#include <stdio.h>
void someFunction(int arg)
{
    printf("This is someFunction being called and arg is: %d\n", arg);
    printf("Whoops leaving the function now!\n");
}

main()
{
    void (*pf)(int);
    pf = &someFunction;
    printf("We're about to call someFunction() using a pointer!\n");
    (pf)(5);
    printf("Wow that was cool. Back to main now!\n\n");
}
```

https://www.learn-c.org/en/Function_Pointers

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)(…)



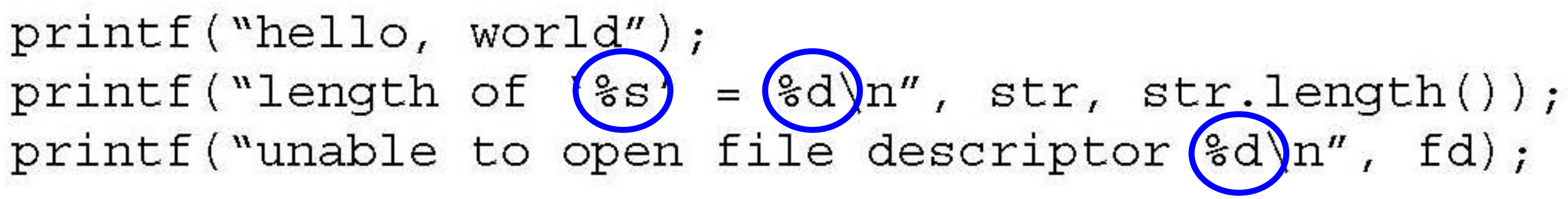
Other Overflow Targets

- 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);
```

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
```

- Sloppy use of printf format string:

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

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

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

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);
// should’ve used printf(“%s”, buf);
```

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

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 cryptographic 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

Tested on Wednesday, on barb.cs

```
#include <stdio.h>

int main()
{
    char *buf = "%08x\t%08x\t%08x\t%08x\n";
    printf(buf);
}
```

`Compiled with gcc`

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

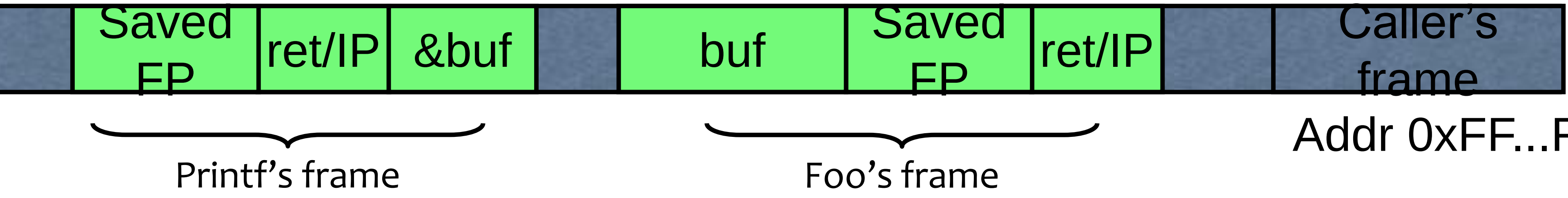
- What if printf does not have an argument?

```
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.

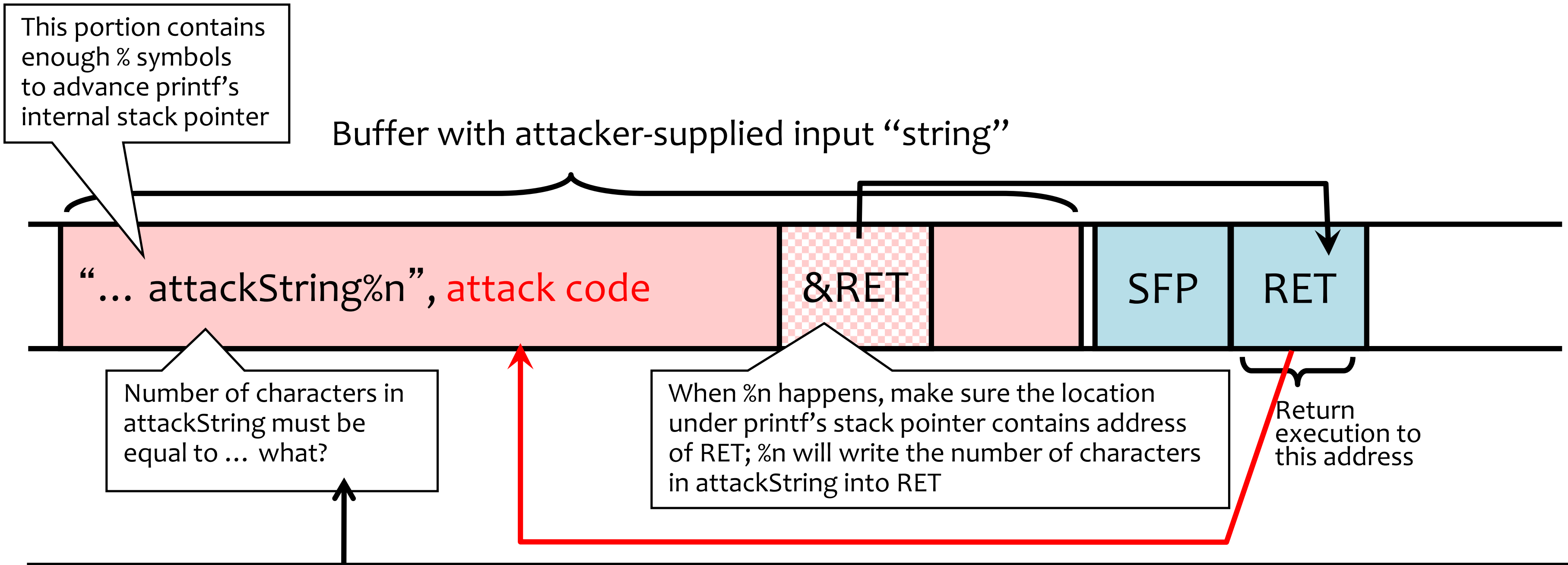
How Can We Attack This?

```
foo() {
      char buf[…] = "attackString";
      printf(buf); //vulnerable
}
```



What should "attackString" be??

Using %n to Overwrite Return Address



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

Buffer with attacker-supplied input “string”

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:
  - Smashing the Stack for Fun and Profit
  - Exploiting Format String Vulnerabilities
- Links to these readings are posted on the course schedule