CSE 484: Computer Security

Software Security: Buffer Overflow Attacks and More

Spring 2023

David Kohlbrenner
dkohlbre@cs

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

Announcements + Logistics

- Things Due:
  - Reading #1 today!
- Lab 1:
  - **Going out soon**
  - If you haven't filled out the partner survey, we can't help you with finding one!
- Office Hours:
  - TBA

- Next week:
  - Paper discussion at the beginning of class.

Learning new languages/tools/etc

- Security often requires rapidly acclimating to new tech
- You don’t need mastery
- Running into a system/language/construct you don’t know is expected, and expected to be hard!
- Don’t understand a term in class? Ask!

This week's paper

- "Low-level Software Security: Attacks and Defenses"

Last time…

- Threat models
  - Assets
  - Adversaries
  - Vulnerabilities
  - Threats
  - Risks

(SOME OF) SOFTWARE SECURITY

Bugs, Vulnerabilities, and Exploits

- Bug
  – Not working quite right
- Vulnerability
  – A malfunction that can be used for an adversary's goals
- Exploit
  – The mechanical set of operations to make use of a vulnerability

Adversarial Failures

- Software bugs are bad
  - Consequences can be serious
- Even worse when an intelligent adversary wishes to exploit them!
  - Intelligent adversaries: Force bugs into "worst possible" conditions/states
  - Intelligent adversaries: Pick their targets

Aside: The Weird Machine

- An exploit can also be considered a *program* for a *weird machine*
- If you are more formally-inclined, check out:
  - https://www.cs.dartmouth.edu/~sergey/wm/

Many types of vulnerability

- Pollev.com/dkohlbre
- Talk to your neighbors, define one you’ve heard of (or ask about one you don’t know!)

Memory Corruption Bugs

- Buffer overflows bugs: Big class of bugs
  - Normal conditions: Can sometimes cause systems to fail
  - Adversarial conditions: Attacker able to violate security of your system (control, obtain private information, ...)
- Stack, Heap both possibilities

A note on languages

- We're going to be assuming code is written in an *unsafe language*
  - Like C

- Fundamentally, we care about the executed binary
  - So the language is sometimes immaterial

BUFFER OVERFLOWS

A Bit of History: Morris Worm

- Worm was released in 1988 by Robert Morris
  - Graduate student at Cornell, son of NSA chief scientist
  - Convicted under Computer Fraud and Abuse Act,
    - 3 years probation and 400 hours of community service
- Worm was intended to propagate slowly and harmlessly measure the size of the Internet
- Due to a coding error, it created new copies as fast as it could and overloaded infected machines
- $10-100M worth of damage

Morris Worm and Buffer Overflow

- One of the worm's propagation techniques was a buffer overflow attack against a vulnerable version of `fingerd` on VAX systems
  - By sending special string to finger daemon, worm caused it to execute code creating a new worm copy

Overflows remain a common source of vulnerabilities and exploits today!

(Especially in embedded systems.)

Aside: Famous Internet Worms

- Morris worm (1988): overflow in `fingerd`
  - 6,000 machines infected
- CodeRed (2001): overflow in MS-IIS server
  - 300,000 machines infected in 14 hours
- SQL Slammer (2003): overflow in MS-SQL server
  - 75,000 machines infected in **10 minutes** (!!)
- Sasser (2005): overflow in Windows LSASS
  - Around 500,000 machines infected

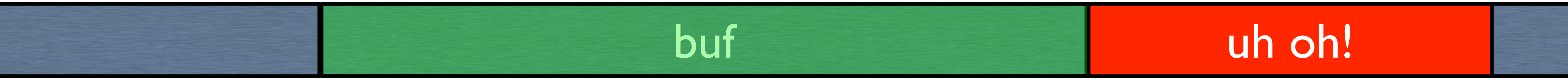
… And More

- Conficker (2008-09): overflow in Windows RPC
  – Around 10 million machines infected (estimates vary)
- Stuxnet (2009-10): several zero-day overflows + same Windows RPC overflow as Conficker
  – Windows print spooler service
  – Windows LNK shortcut display
  – Windows task scheduler
- Flame (2010-12): same print spooler and LNK overflows as Stuxnet
  – Targeted cyperespionage virus
- These days, worms are uncommon

Attacks on Memory Buffers

- Buffer is a pre-defined data storage area inside computer memory (stack or heap)
- Typical situation:
  - A function takes some input that it writes into a pre-allocated buffer.
  - The developer forgets to check that the size of the input isn't larger than the size of the buffer.
  - Uh oh.
    - "Normal" bad input: crash
    - "Adversarial" bad input : take control of execution

Stack Buffers

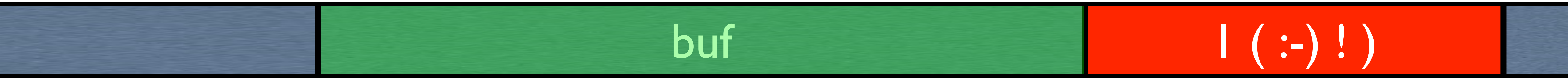


- Suppose Web server contains this function

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

- No bounds checking on strcpy()
- If str is longer than 126 bytes
  - Program may crash
  - Attacker may change program behavior

Example: Changing Flags



- Suppose Web server contains this function

```
void func(char *str) {
        byte auth = 0;
        char buf[126];
        ...
        strcpy(buf,str);
        ...
    }
```

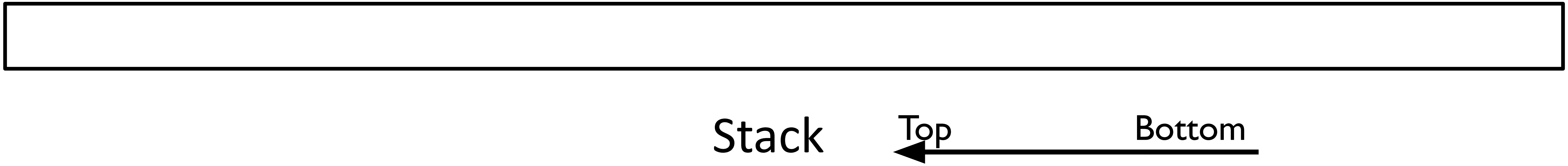
- Authenticated variable non-zero when user has extra privileges
- Morris worm also overflowed a buffer to overwrite an authenticated flag in fingerd

Memory Layout

- Text region: Executable code of the program
- Heap: Dynamically allocated data
- Stack: Local variables, function return addresses; grows and shrinks as functions are called and return

What happens on function call?

What happens on function call?



Stack Buffers

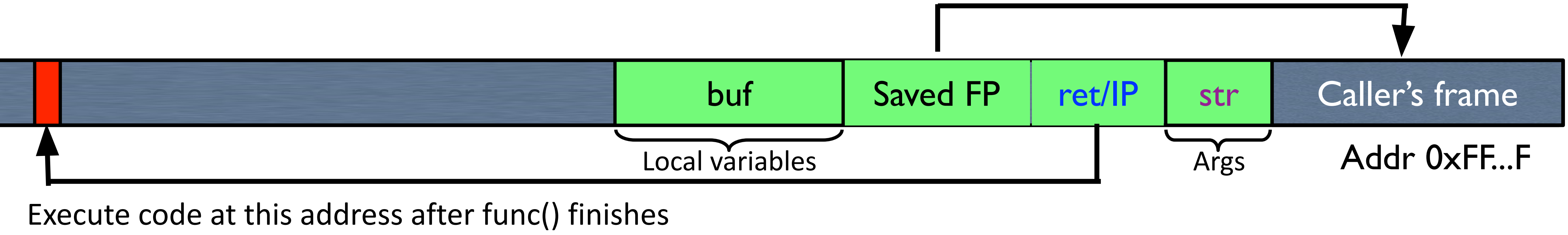
- Suppose Web server contains this function:

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

Allocate local buffer (126 bytes reserved on stack)

Copy argument into local buffer

- When this function is invoked, a new frame (activation record) is pushed onto the stack.



What if Buffer is Overstuffed?

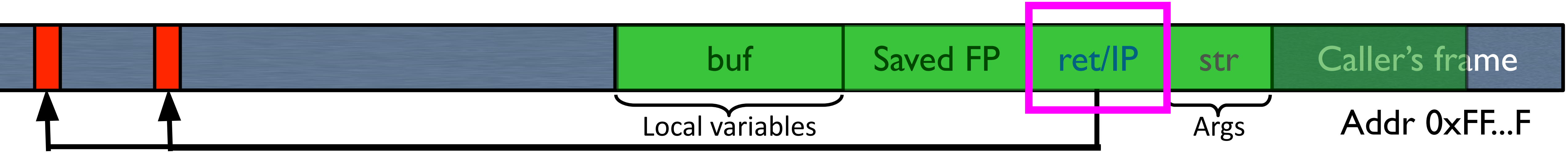
- Memory pointed to by str is copied onto stack…

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

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

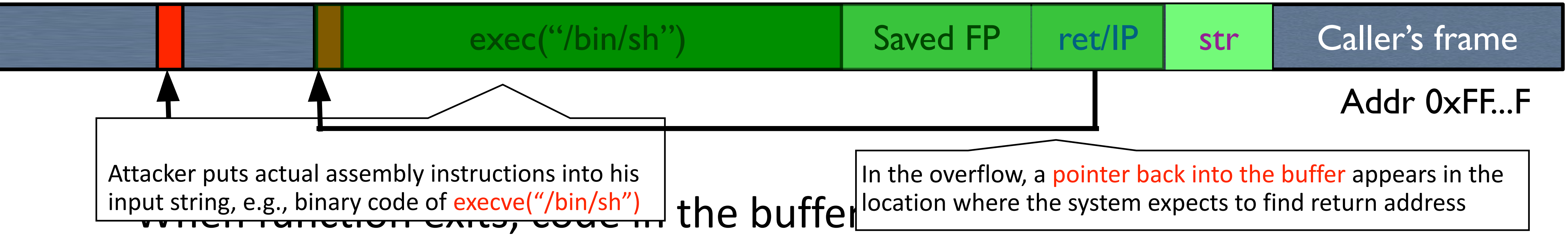
- If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations.

This will be interpreted as return address!



Executing Attack Code

- Suppose buffer contains attacker-created string
  - For example, str points to a string received from the network as the URL



- When function exits, code in the buffer will be executed, giving attacker a shell **("shellcode")**
  - Root shell if the victim program is setuid root

Buffer Overflows Can Be Tricky…

- Overflow portion of the buffer must contain correct address of attack code in the RET position
  - The value in the RET position must point to the beginning of attack assembly code in the buffer
    - Otherwise application will (probably) crash with segfault
  - **Attacker must correctly guess in which stack position his/her buffer will be when the function is called**

Problem: No Bounds Checking

- strcpy does <u>not</u> check input size
  - strcpy(buf, str) simply copies memory contents into buf starting from *str until "\0" is encountered, ignoring the size of area allocated to buf
- Many C library functions are unsafe
  - strcpy(char *dest, const char *src)
  - strcat(char *dest, const char *src)
  - gets(char *s)
  - scanf(const char *format, …)
  - printf(const char *format, …)

Does Bounds Checking Help?

- strncpy(char *dest, const char *src, size_t n)
  - If strncpy is used instead of strcpy, no more than n characters will be copied from *src to *dest
    - Programmer has to supply the right value of n
- 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);
```

Does Bounds Checking Help? Pollev Discussion Time

- strncpy(char *dest, const char *src, size_t n)
  - If strncpy is used instead of strcpy, no more than n characters will be copied from *src to *dest
    - Programmer has to supply the right value of n
- 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);
```

Consider these changes

Apache 1.3 had the following code:

```
strcpy(record, user);
strcat(record, ”:”);
strcat(record, cpw);
```

The published fix:

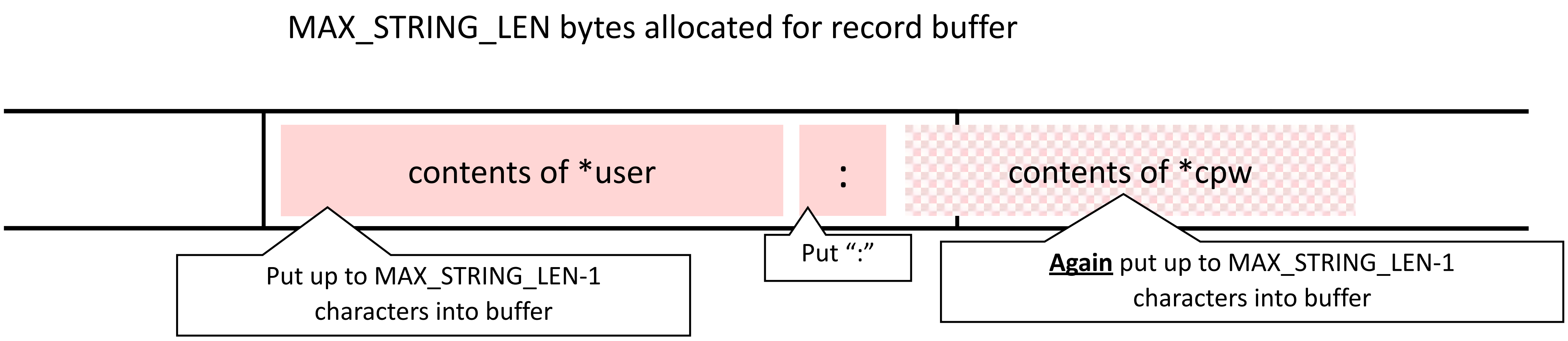
```
strncpy(record, user, MAX_STRING_LEN-1);
strcat(record,”:”);
strncat(record, cpw, MAX_STRING_LEN-1);
```

Is this fix good? If so, why? If not, why not?

Misuse of strncpy in htpasswd "Fix"

- Published "fix" for Apache htpasswd overflow:

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



Consider this homebrewed copy:

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

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

}
```

Consider this homebrewed copy:

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

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

}
```

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

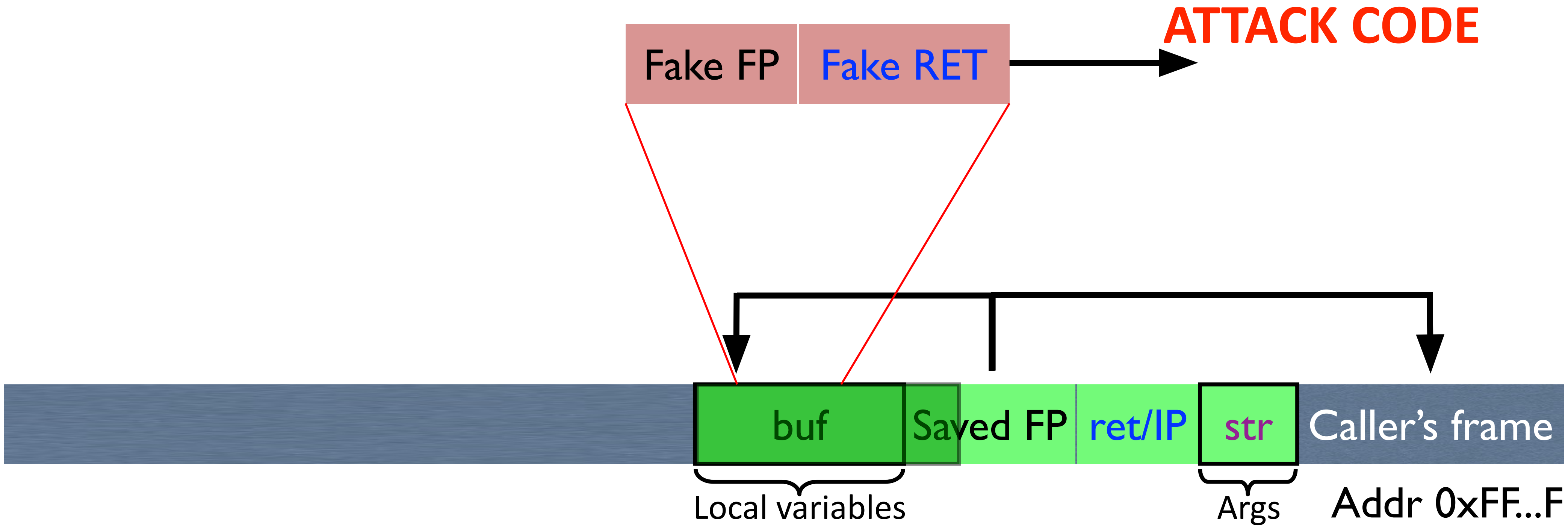
Off-By-One Overflow

- 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]);
}
```

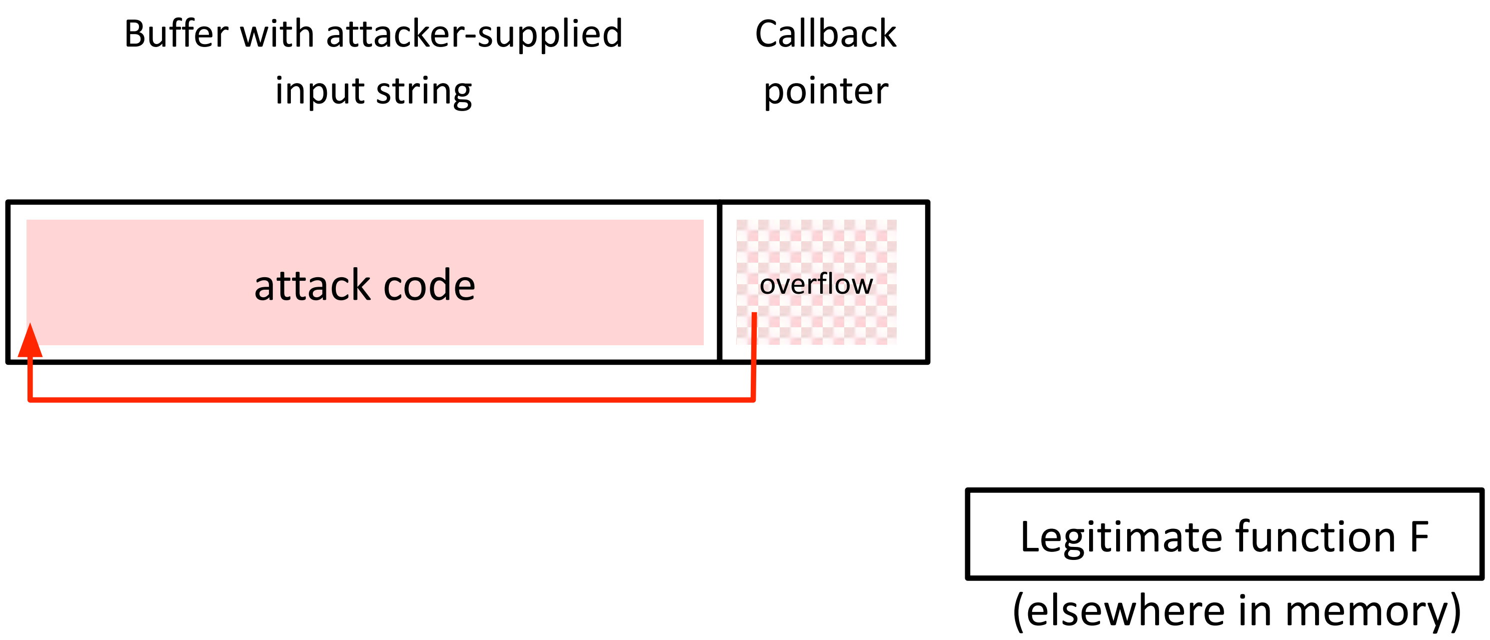
- 1-byte overflow: can't change RET, but can change pointer to previous stack frame…

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)(…)



A note on assembly

- You will need to read some assembly
- Its all x86_32 assembly
- There are two syntaxes (I'm sorry)

Shall we do one live?

Other Overflow Targets

- Format strings in C
  - We'll walk through this one today
- Heap management structures used by malloc()
  - 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
```

- 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 % ???

Implementation of Variable Args

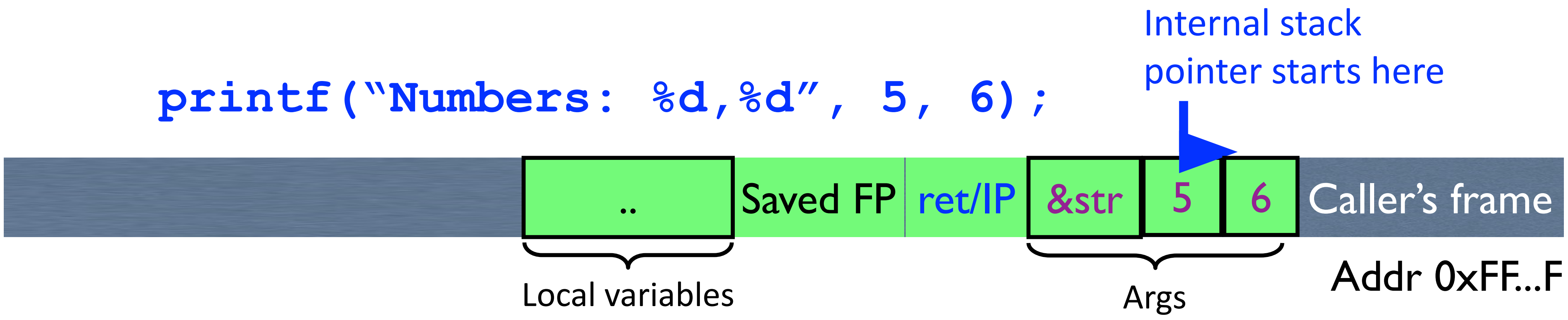
- Special functions va_start, va_arg, va_end compute arguments at run-time

```
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 */
}
```

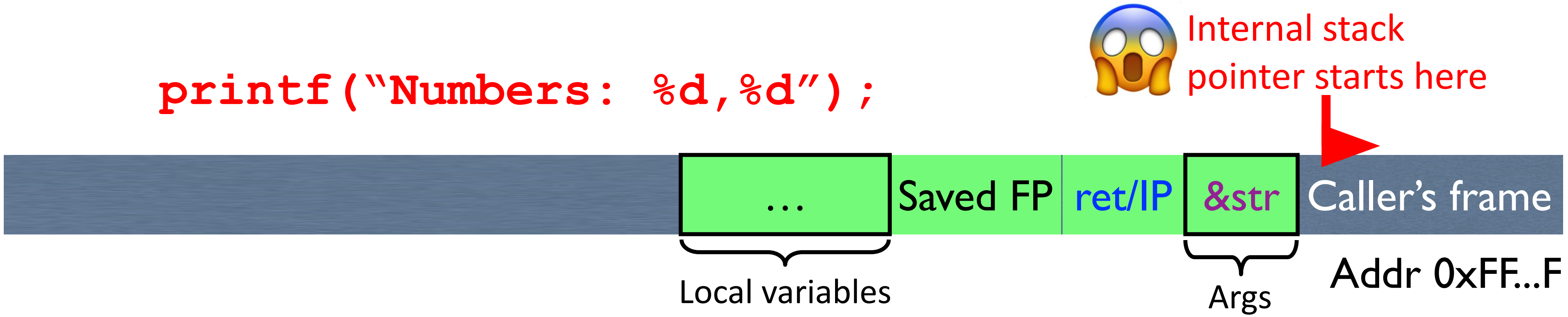
This is simplified code, e.g., handles %d but not %10d

Closer Look at the Stack

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




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



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);
```

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 % ???

Break!

- Back at:

- Think about varargs (printf) calls…

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

Try This At Home

```
#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 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

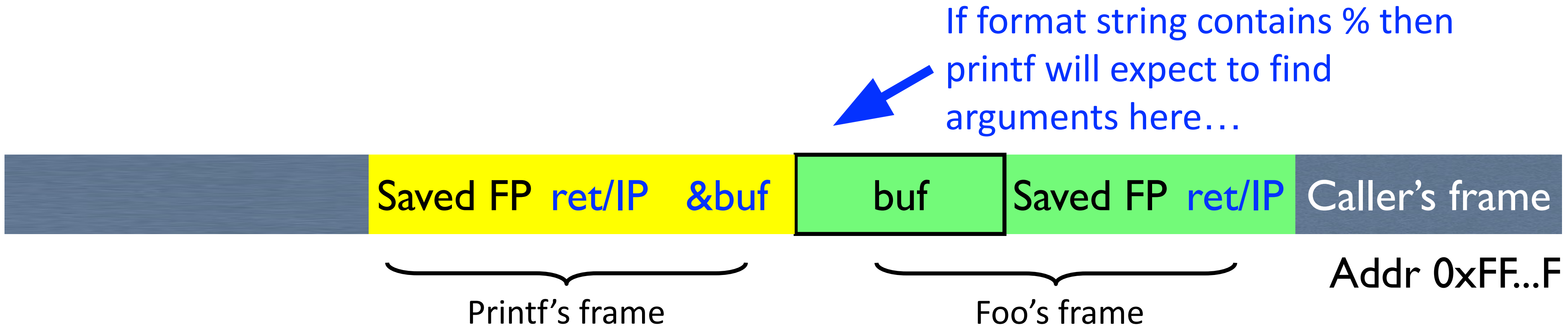
- Printf takes a variable number of arguments
  - E.g., printf("Here's an int: %d", 10);
- Assumptions about input can lead to trouble
  - E.g., printf(buf) when buf="Hello world" versus when buf="Hello world %d"
  - Can be used to advance printf's internal stack pointer
  - Can read memory
    - E.g., 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

“Weird Machines”

- Way of thinking about exploits (the best way ☺️)
- Treat each discrete side-effect as an ‘instruction’
- Synthesize a ‘program’ from these instructions
- This is now your exploit!

How Can We Attack This?

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

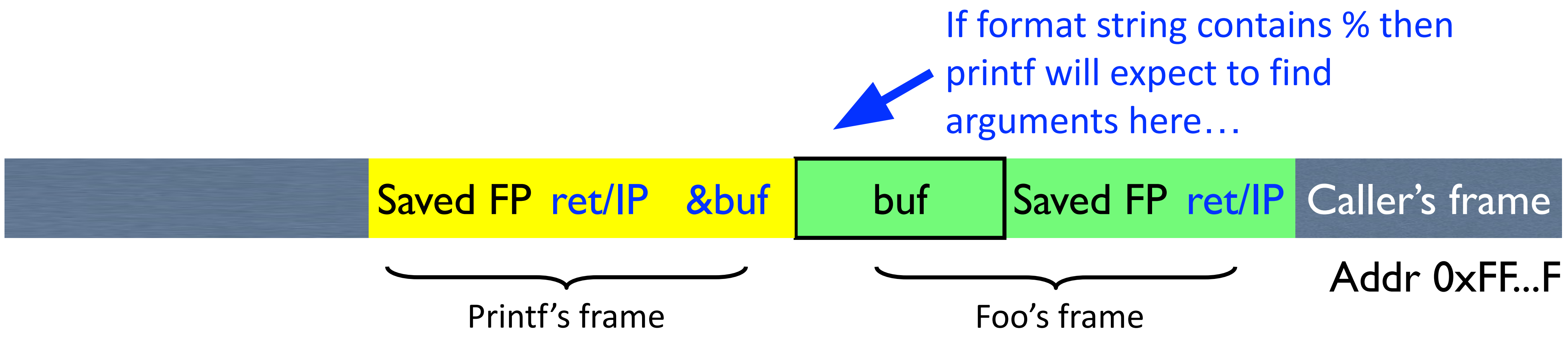


What should the string returned by readUntrustedInput() contain?

Different compilers / compiler options / architectures might vary

Pollev and Discussion Time

```
foo() {
    char buf[2048];
    strncpy(buf, readUntrustedInput(), sizeof(buf));
    printf(buf); //vulnerable
}
```

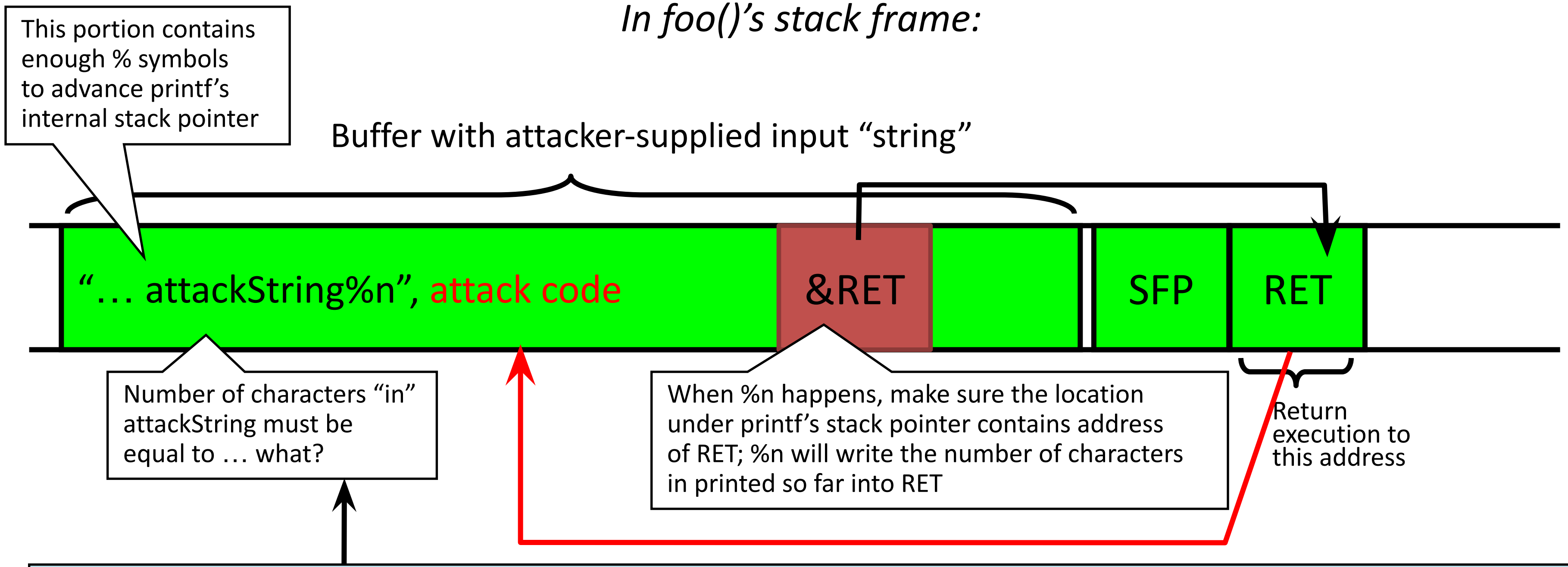


What should the string returned by readUntrustedInput() contain?

Different compilers / compiler options / architectures might vary

Using %n to Overwrite Return Address

In foo()'s stack frame:



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)

Lab 1 will go out soon

- Significant help from doing these readings:
  - Smashing the Stack for Fun and Profit
  - Exploiting Format String Vulnerabilities
- I'll go through partner requests shortly
- Live example of sploit0 next week at the beginning of class