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

Winter 2022

Tadayoshi (Yoshi) Kohno
yoshi@cs

UW Instruction Team: David Kohlbrenner, Yoshi Kohno, Franziska Roesner. Thanks to Dan Boneh, Dieter Gollmann, Dan Halperin, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials ...

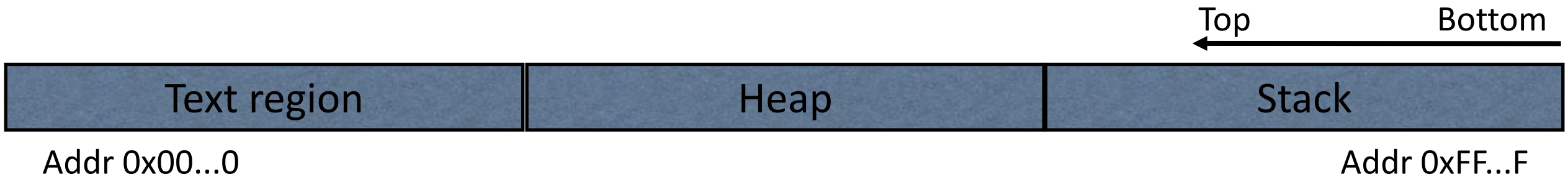
Announcements

- Things Due:
  - Homework #1: Due Thursday
  - Research Readings (CSE M 584): Due Thursday (and every Thursday thereafter)
- Lab 1: Start forming groups of up to 3 people now (strongly encouraged to have groups of 3)

First, Review slides from Friday

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



Stack Buffers

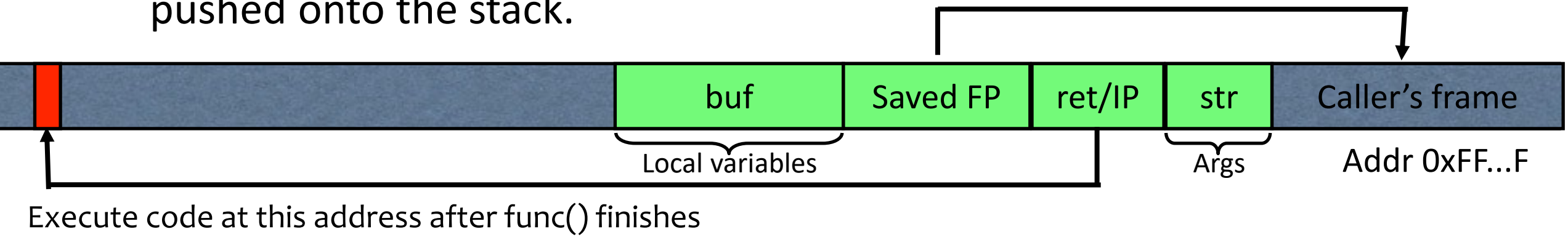
- Suppose Web server contains this function:

```
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?

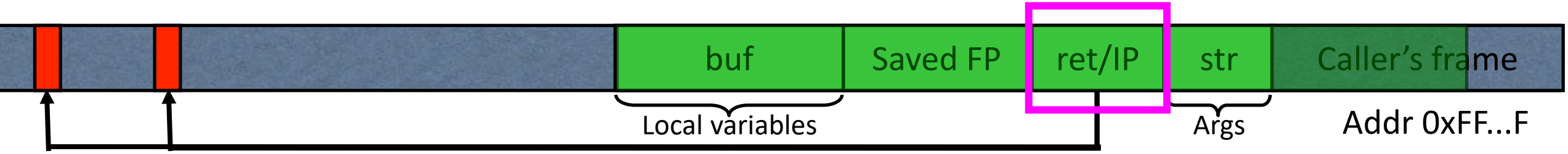
- Memory pointed to by str is copied onto stack…

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

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

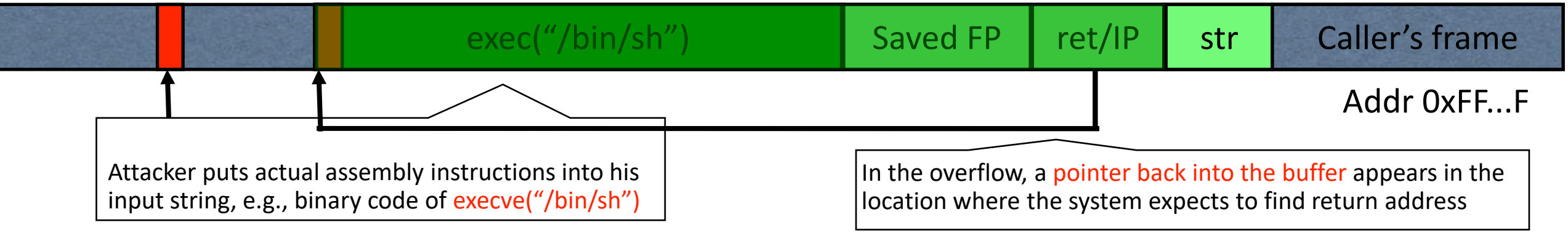
- If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations.

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



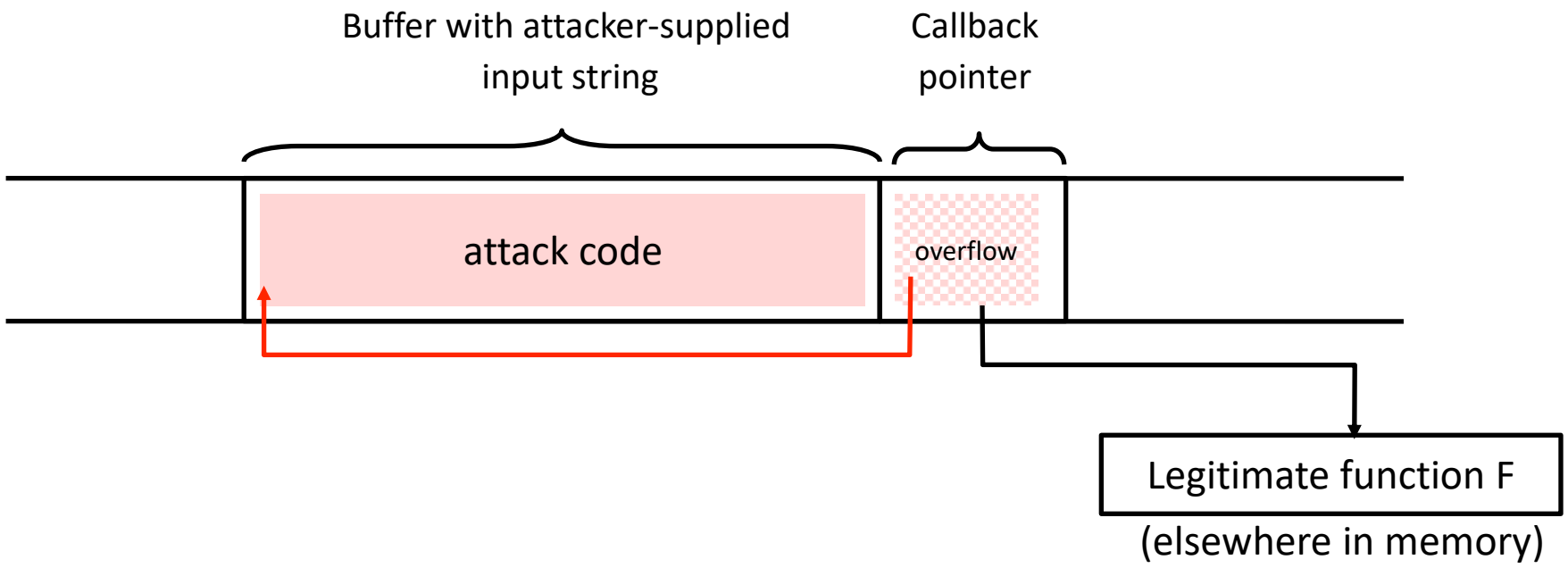
- When function exits, code in the buffer will be executed, giving attacker a shell **("shellcode")**
  - Root shell if the victim program is setuid root

Review: “End”

- “End” in quotes because although those were past slides, we have not ended the review – we will continue to revisit the contents of the earlier slides, just in slightly different ways

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
  - 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 % ???

Implementation of Variable Args

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

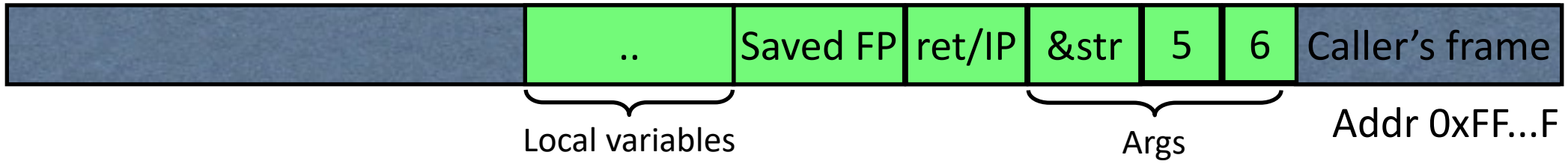
```
void printf(const char* format, ...)
{
     int i; char c; char* s; double d;
     va_list ap;  /* declare an "argument pointer" to a variable arg list */
     va_start(ap, format);  /* initialize arg pointer using last known arg */

     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

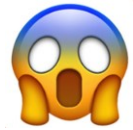
Closer Look at the Stack

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

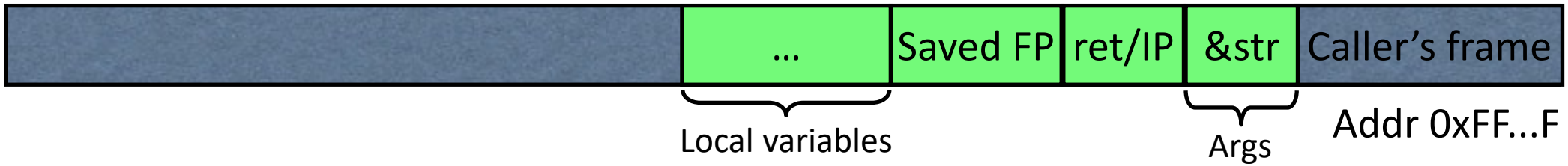
Printf's internal stack pointer starts here




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



Printf's internal stack pointer starts here



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);
```

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!

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

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

- Unsafe use of printf format string:

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

- 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

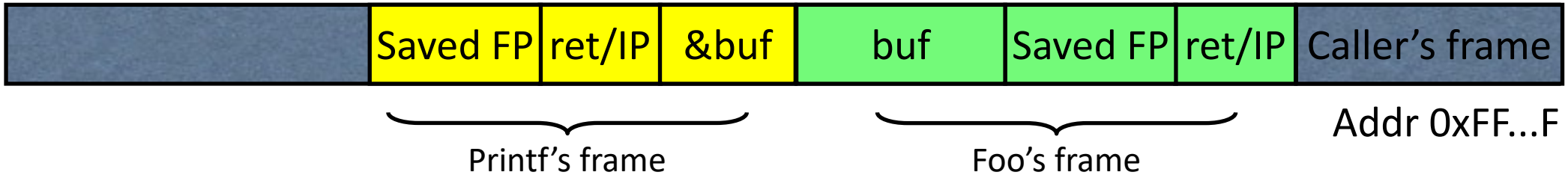
“Weird Machines”

- 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? Breakout -> In-Class Activity

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

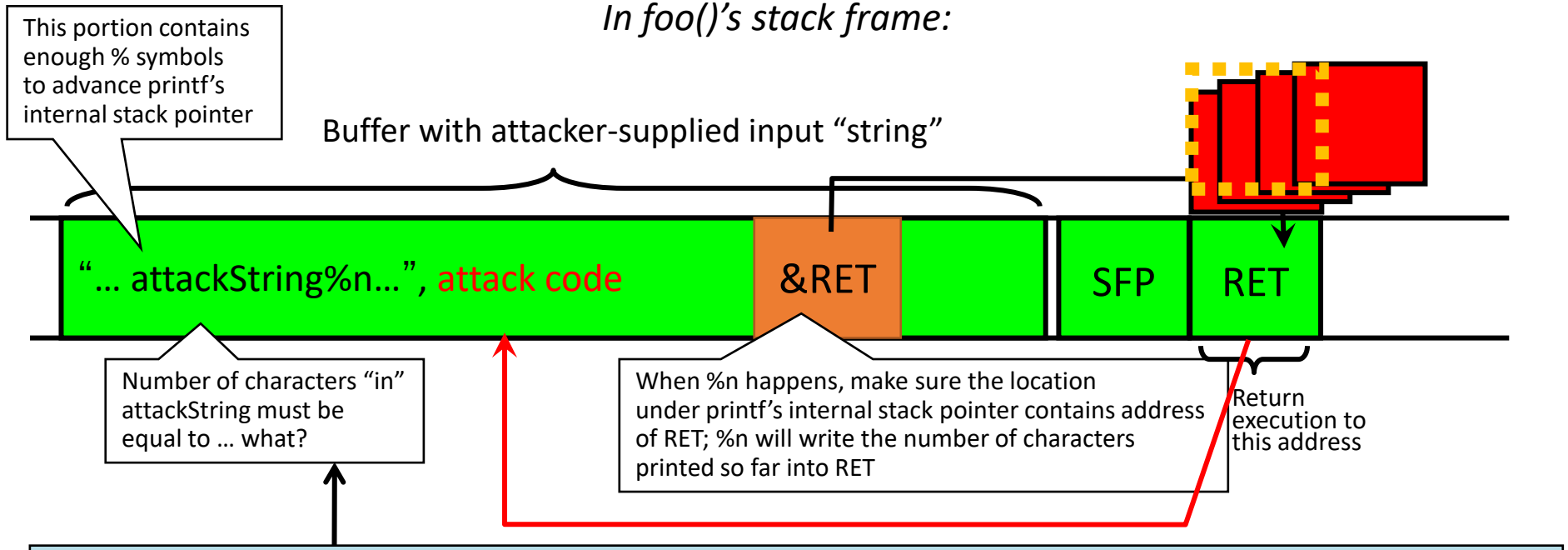
If format string contains % then printf will expect to find arguments here…



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

Different compilers / compiler options / architectures might vary

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

Buffer Overflow: Causes and Cures

- Classical memory exploit involves code injection
  - Put malicious code at a predictable location in memory, usually masquerading as data
  - Trick vulnerable program into passing control to it

- Possible defenses:
  1. Prevent execution of untrusted code
  2. Stack "canaries"
  3. Encrypt pointers
  4. Address space layout randomization
  5. Code analysis
  6. …

Defense: Executable Space Protection

- Mark all writeable memory locations as non-executable
  - Example: Microsoft's Data Execution Prevention (DEP)
  - **This blocks many code injection exploits**
- Hardware support
  - AMD "NX" bit (no-execute), Intel "XD" bit (executed disable) (in post-2004 CPUs)
  - Makes memory page non-executable
- Widely deployed
  - Windows XP SP2+ (2004), Linux since 2004 (check distribution), OS X 10.5+ (10.4 for stack but not heap), Android 2.3+

Canvas In-Class Activity

- What might an attacker be able to accomplish even if they cannot execute code on the stack?

What Does "Executable Space Protection" Not Prevent?

- Can still corrupt stack …
  - … or function pointers
  - … or critical data on the heap

- **As long as RET points into existing code, executable space protection will not block control transfer!**
  → return-to-libc exploits

return-to-libc

- Overwrite saved ret (IP) with address of **any library routine**
  - Arrange stack to look like arguments

- Does not look like a huge threat
  - …

return-to-libc

- Overwrite saved ret (IP) with address of **any library routine**
  - Arrange stack to look like arguments

- Does not look like a huge threat
  - …
  - We can call *any* function we want!
  - Say, exec ☺

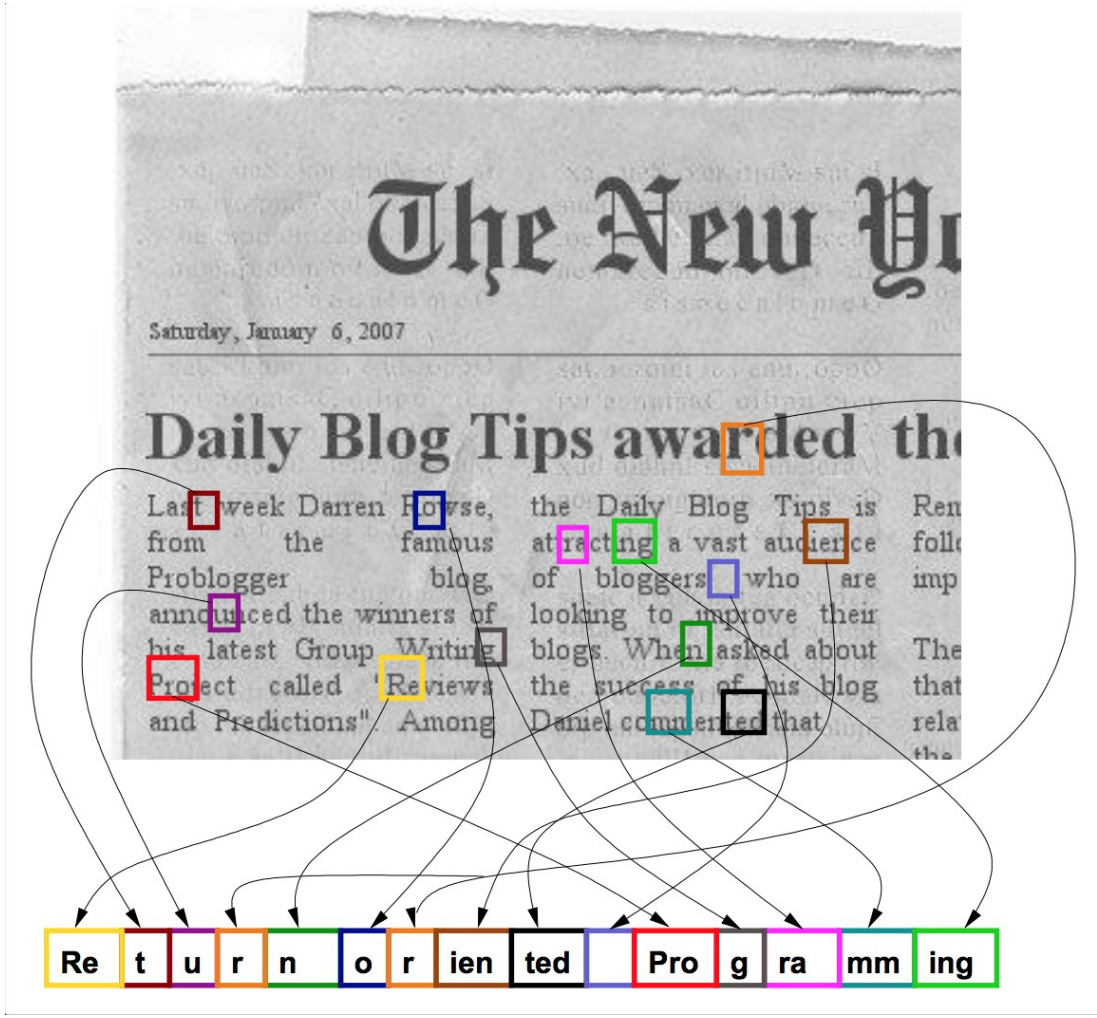
return-to-libc++

- Insight: Overwritten saved EIP need not point to the *beginning* of a library routine
- **Any** existing instruction in the code image is fine
  - Will execute the sequence starting from this instruction
- What if instruction sequence contains RET?
  - Execution will be transferred… to where?
  - Read the word pointed to by stack pointer (SP)
    - Guess what? Its value is under attacker's control!
  - Use it as the new value for IP
    - Now control is transferred to an address of attacker's choice!
  - Increment SP to point to the next word on the stack

Chaining RETs

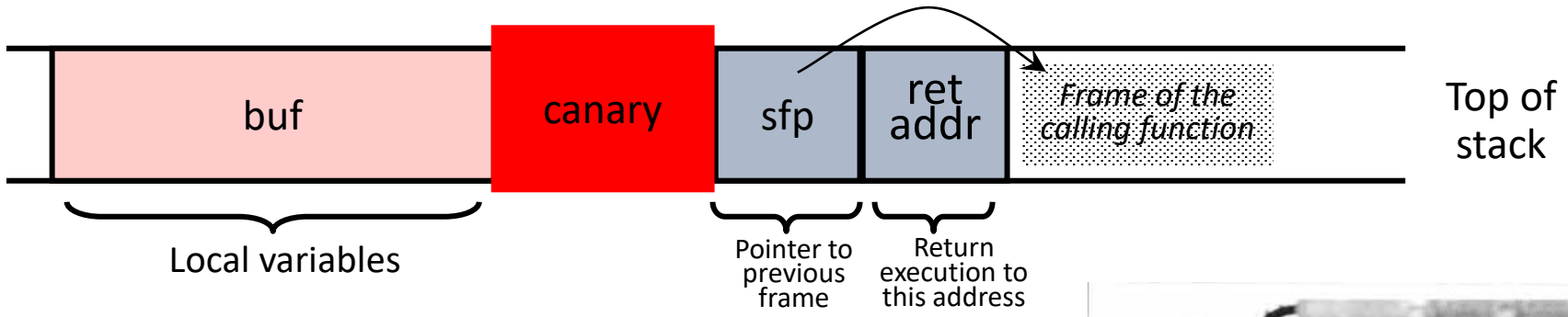
- Can chain together sequences ending in RET
  - Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique" (2005)
- What is this good for?
- Answer [Shacham et al.]: everything
  - Turing-complete language
  - Build "gadgets" for load-store, arithmetic, logic, control flow, system calls
  - Attack can perform arbitrary computation using no injected code at all – return-oriented programming
- Truly, a "weird machine"

Return-Oriented Programming



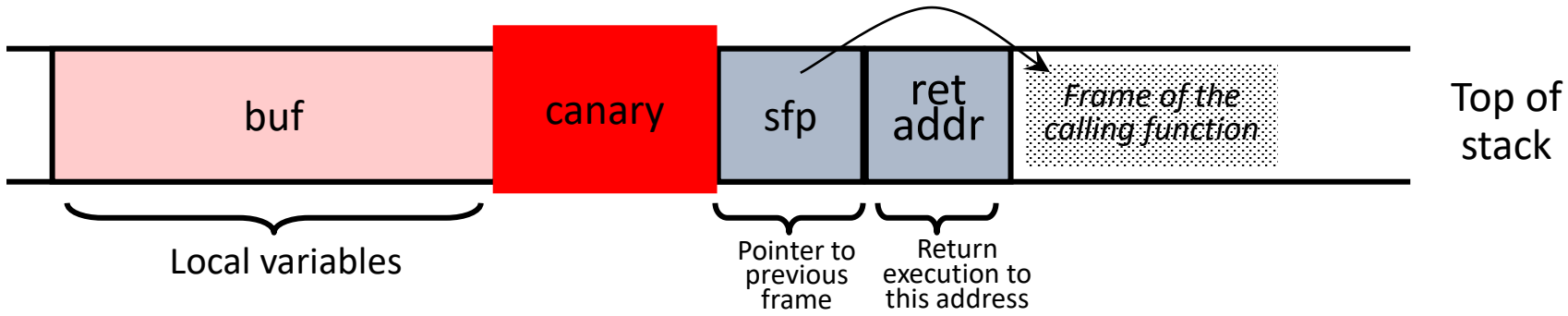
Defense: Run-Time Checking: StackGuard

- Embed "canaries" (stack cookies) in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary



Defense: Run-Time Checking: StackGuard

- Embed "canaries" (stack cookies) in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary



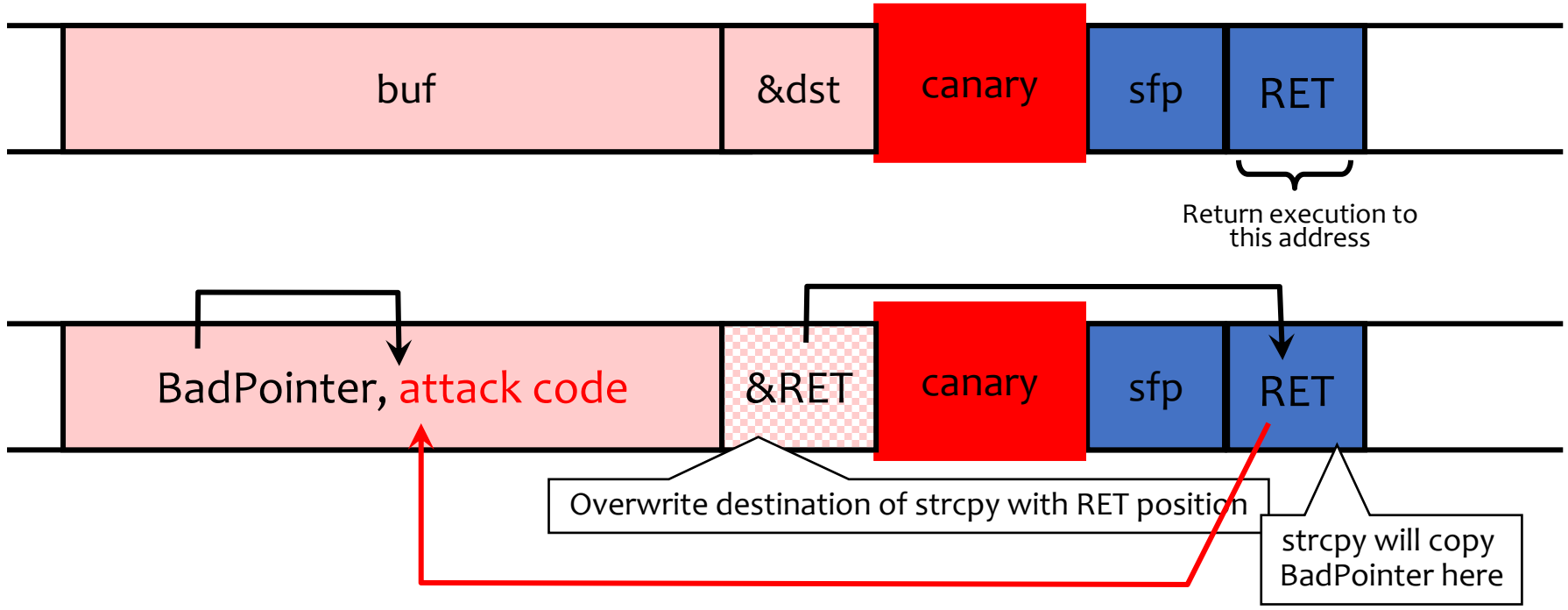
- Choose random canary string on program start
  - Attacker can't guess what the value of canary will be
- Canary contains: "\0", newline, linefeed, EOF
  - String functions like strcpy won't copy beyond "\0"

StackGuard Implementation

- StackGuard requires code recompilation
- Checking canary integrity prior to every function return causes a performance penalty
  - For example, 8% for Apache Web server at one point in time

Defeating StackGuard

- StackGuard can be defeated
  - A single memory write where the attacker controls both the value and the destination is sufficient
- Suppose program contains copy(buf,attacker-input) and copy(dst,buf)
  - Example: dst is a local pointer variable
  - Attacker controls both buf and dst



ASLR: Address Space Randomization

- Randomly arrange address space of key data areas for a process
  - Base of executable region
  - Position of stack
  - Position of heap
  - Position of libraries
- Introduced by Linux PaX project in 2001
- Adopted by OpenBSD in 2003
- Adopted by Linux in 2005

ASLR: Address Space Randomization

- Deployment (examples)
  - Linux kernel since 2.6.12 (2005+)
  - Android 4.0+
  - iOS 4.3+ ; OS X 10.5+
  - Microsoft since Windows Vista (2007)
- Attacker goal: Guess or figure out target address (or addresses)
- ASLR more effective on 64-bit architectures

Attacking ASLR

- **NOP sleds** and **heap spraying** to increase likelihood for adversary's code to be reached (e.g., on heap)
- Brute force attacks or memory disclosures to map out memory on the fly
  - Disclosing a single address can reveal the location of all code within a library, depending on the ASLR implementation