CSE 484 / CSE M 584: Computer Security and Privacy

Autumn 2019

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

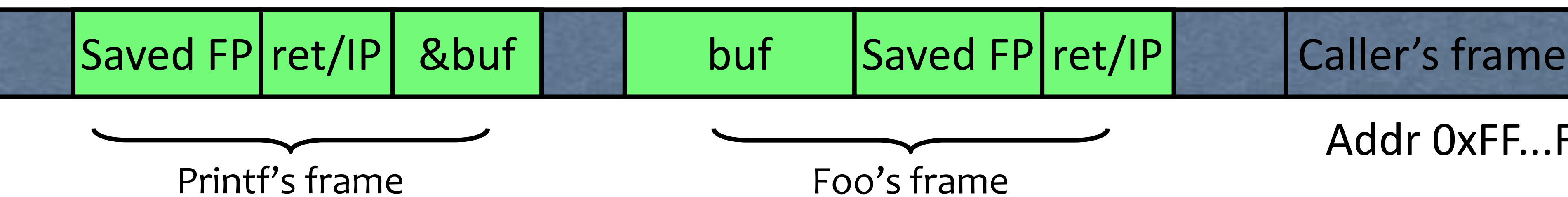
Thanks to Dan Boneh, Dieter Gollmann, Dan Halperin, John Manferdelli, John Mitchell, Franzi Roesner, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials ...

Announcements

- Day Before Thanksgiving: Alternate Video Lesson (e.g., use to support your final project)
- Final Project: Online, marked as draft but dates *should* be set
  - Linked off of Assignments page
  - 12-15 minute video on security-related topic of your choice
  - Note requirements, e.g., include references, discuss ethics/legal issues, length
- Lab 1: Try to make sure your sploit0 works by end of today (recommendation)

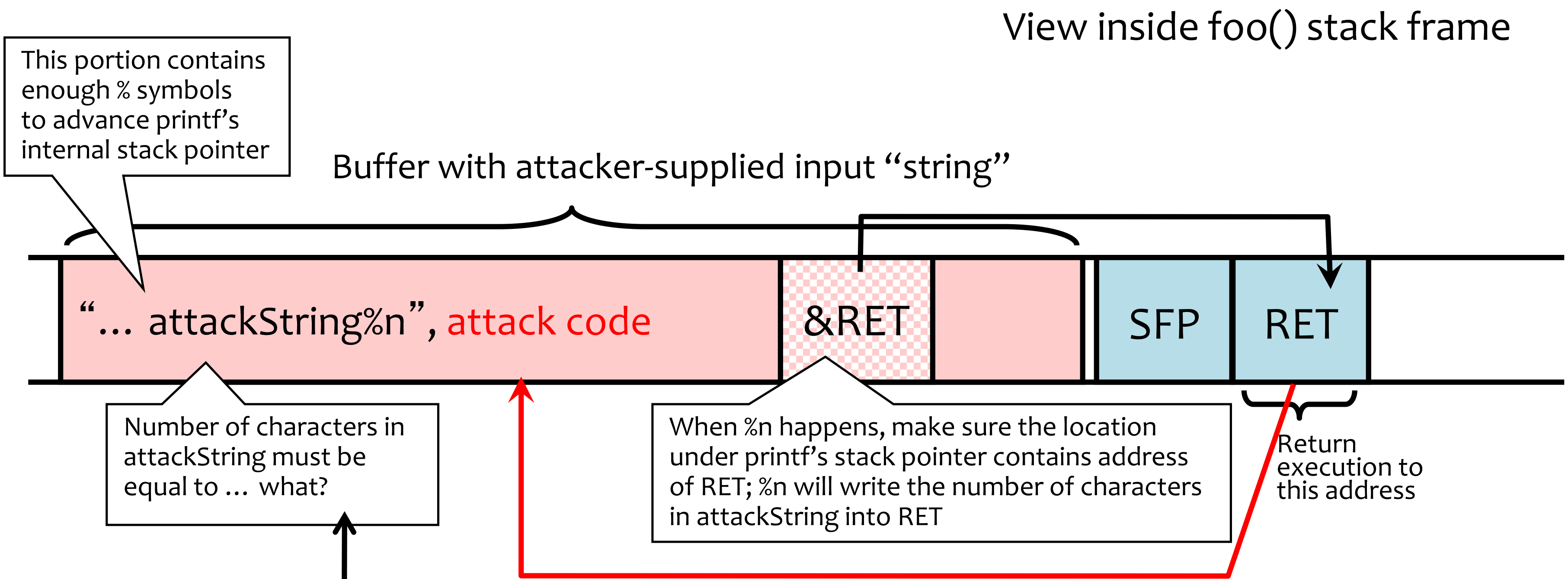
How Can We Attack This?

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



What should “attackString” be??

Using %n to Overwrite Return 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)

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

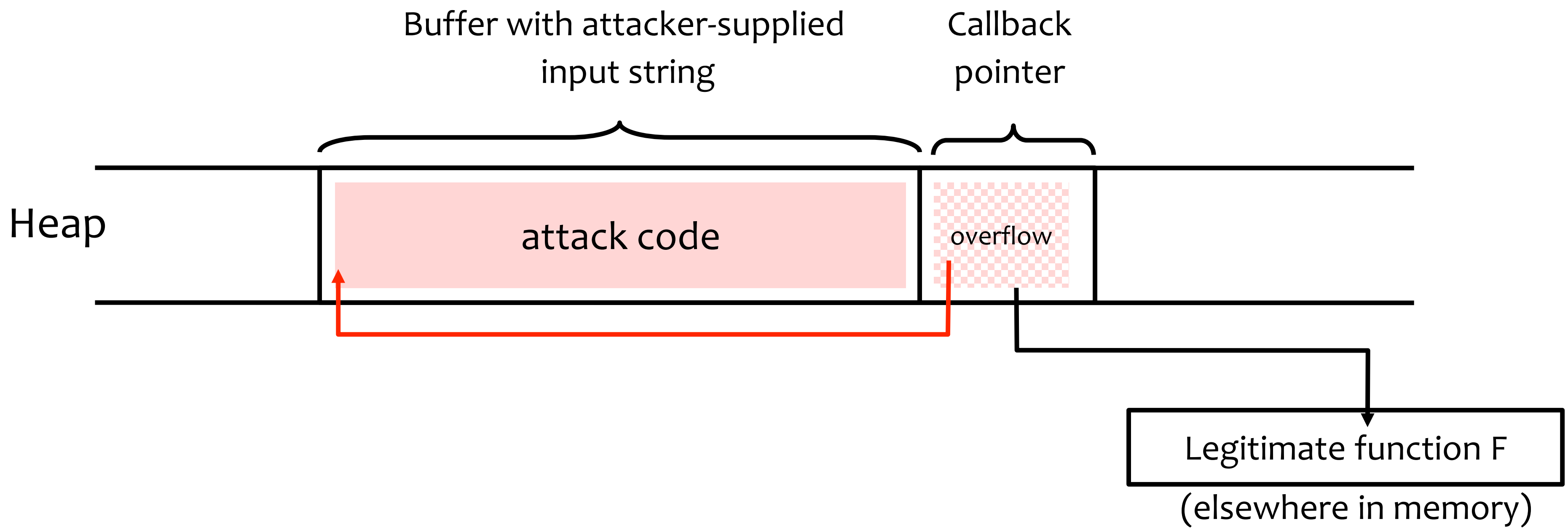
```c
#include <stdio.h>
void someFunction(int arg)
{
    printf("This is someFunction being called and arg is: %d\n", arg);
    printf("Whoops leaving the function now!\n");
}

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 Target

- Heap management structures used by malloc()
  - More details in section

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 in the lab description

Stepping Back

- This class: Broad tour of key concepts in security
  - Key principles
  - Foundations / historical perspective
  - Threat modeling, context, ethics, …
  - 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

Buffer Overflow: Causes and Cures

- Classic memory exploit involves code injection
  - Approach: 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. …

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+

What Does "Executable Space Protection" Not Prevent?

- Write on back of worksheet (we'll call this worksheet 6)

What Does "Executable Space Protection" Not Prevent?

- Can still corrupt stack …
  - … or function pointers or critical data on the heap
- **As long as "saved EIP" points into existing code, executable space protection will not block control transfer**
- This is the basis of return-to-libc exploits
  - Overwrite saved EIP with address of any library routine, arrange stack to look like arguments
- Does not look like a huge threat
  - Attacker cannot execute arbitrary code
  - But … ?

return-to-libc

- Can still call critical functions, like exec
- See lab 1, sploit 8

return-to-libc on Steroids

- 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 (ESP)
    - Guess what? Its value is under attacker's control!
  - Use it as the new value for EIP
    - Now control is transferred to an address of attacker's choice!
  - Increment ESP to point to the next word on the stack

Chaining RETs for Fun and Profit

- 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

Return-Oriented Programming