Software Vulnerabilities

Jeff Foster
University of Maryland, College Park

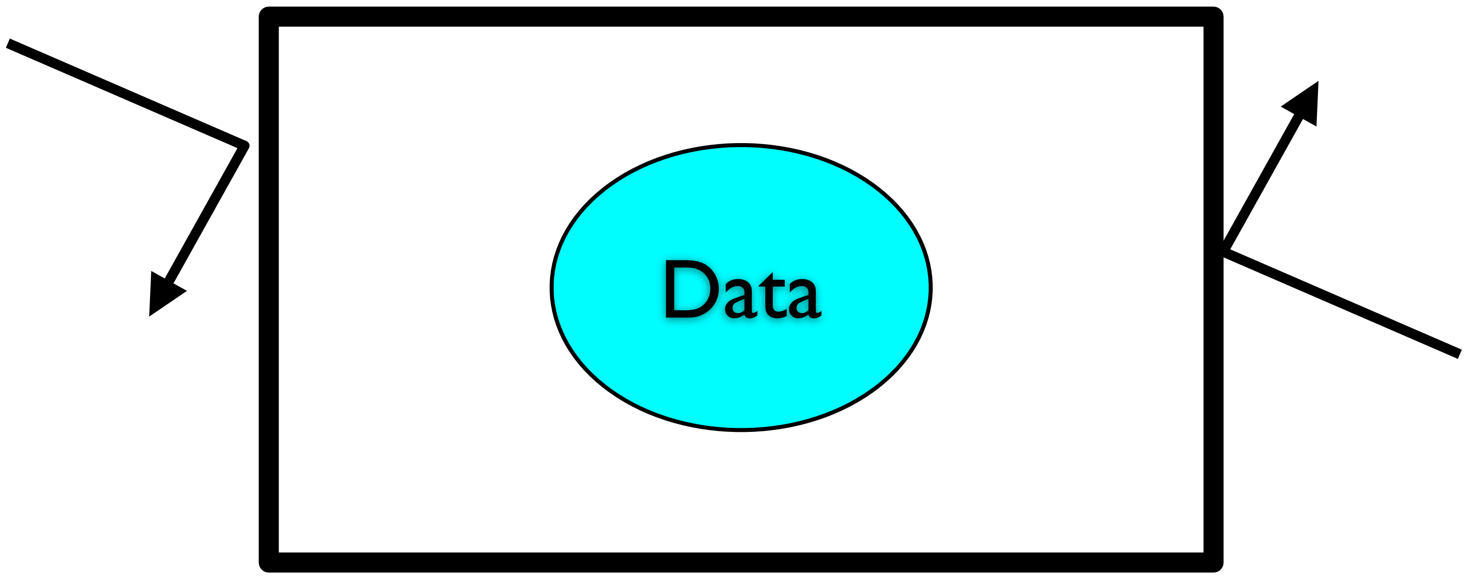
When is a Program Secure?

- When it does exactly what it should!
  - But what is it supposed to do?
    - Someone tells us (do we trust them?)
    - We decide ourselves (how do we check?)
    - We write the code ourselves (how much of the software you use have you written?)

- Perfect security does not exist
  - Must trade off performance, cost, usability, functionality
  - When is software "secure enough"?

Integrity

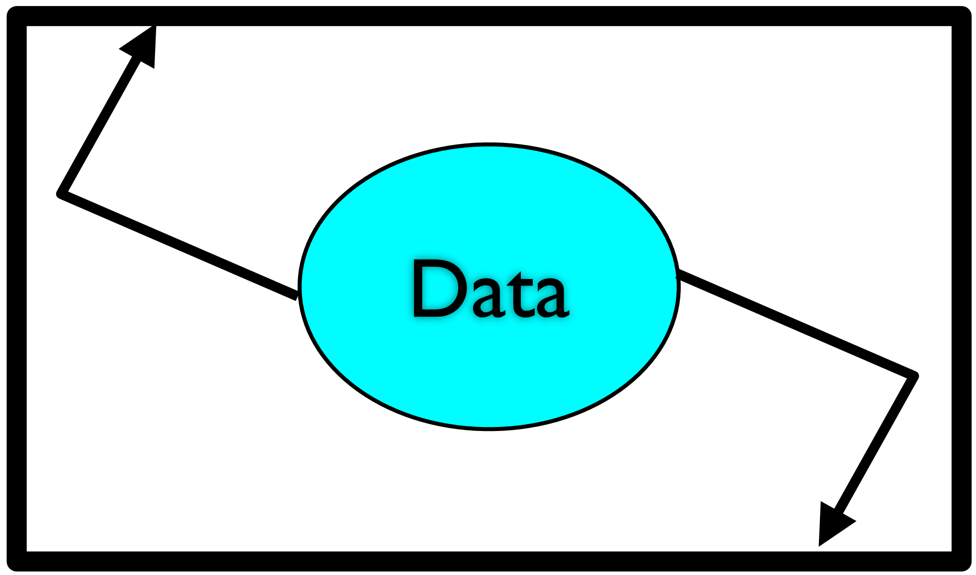
- No improper modification of data



  - E.g., account balance updated only by authorized transactions, only you can change your password
  - Integrity of security mechanism is crucial
  - Enforcement: access control, digital signatures, ...

Confidentiality

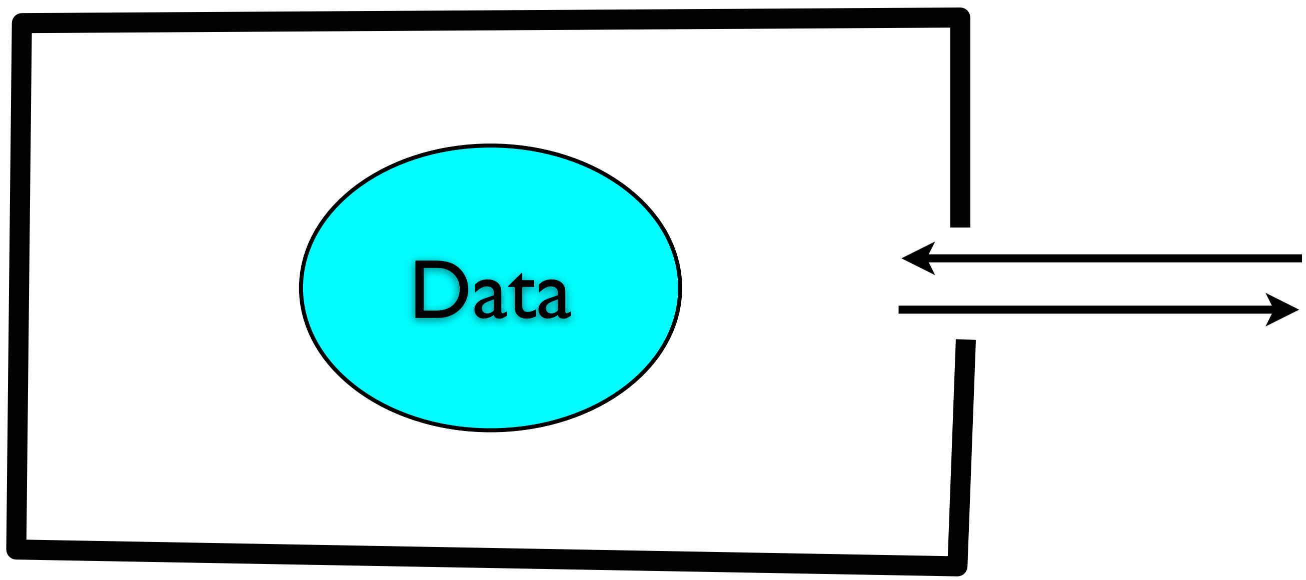
- Protect information from improper release



  - Limit knowledge of data or actions (secrecy)
    - E.g., D-Day attack date, contract bids
  - Enforcement: access control, encryption, ...
  - Hard to enforce after the fact...

Availability

- System must respond to requests



  - I.e., do not ensure confidentiality and integrity by unplugging your computer!

Vulnerabilities

- A *security vulnerability* is a bug that allows an *adversary* to compromise security
- What is the difference between a vulnerability and an ordinary bug?
  - Unclear!
  - Ordinary users try to avoid bugs
    - Adversaries seek out vulnerabilities
  - Also can change over time
    - E.g., publicly available info at courthouse vs. on Internet
    - E.g., modem access to SCADA (industrial control) system

Two classes of vulnerabilities

1. Application-specific issue
   - E.g., forget to check login credentials in one place
   - These problems can be severe
   - But finding a bug in one application doesn't necessarily give clues to other vulnerabilities
2. Language/library/system design issue
   - E.g., buffer overflows
   - These problems tend to cut across many different software systems built with same technology
   - So efforts to exploit and defend against them have a potentially large consequence

Languages and vulnerabilities

- We'll look at three vulnerabilities in C programs:
  - Null pointer errors
  - Format-string vulnerabilities
  - Buffer overflows
- All of these stem from design choices that are fundamental to the C language and its libraries
  - Vulnerabilities exist in lots of systems
  - Everyone knows about them
  - Yet, still hard to stamp out
    - (Ok, the null ones are basically fixed...)

Null pointer errors

- You all know what these are

```
x = NULL;
...
*x;
```

- Obvious problem: denial of service attack
  - Trick target program into dereferencing NULL and it will crash
    - Could take down a server this way
    - If have access as a local user to a machine and the kernel dereferences null, could crash the kernel

Null pointer errors (cont'd)

- What if this happens:

```
fp = NULL;
...
fp(x,y,z);
```

  - Looks about the same, right?

- NULL = 0, which is just another memory addr
  - By convention, NULL is not a valid addr, so results in a failure when we try to look up that logical address
  - But we can change that with mmap!

mmap

```
NAME
      mmap, munmap - map or unmap files or devices into memory

SYNOPSIS
      #include <sys/mman.h>

      void *mmap(void *start, size_t length, int prot, int flags,
                int fd, off_t offset);

      int munmap(void *start, size_t length);

DESCRIPTION
      The  mmap()  function  asks to map length bytes starting at offset offset from
the file (or other object) specified by the file descriptor fd into memory, preferably
at address start.  This latter address is a hint only, and is usually specified as 0.
The actual place where the object is mapped is returned by mmap().
```

Mapping the 0 page

- If we look more closely at the options, we can force the lowest page to be mapped

```
mmap(0, 4096, PROT_READ|PROT_WRITE,
        MAP_PRIVATE|MAP_ANONYMOUS|MAP_FIXED, -1, 0);
```

  - PROT_READ|PROT_WRITE = can read and write page
  - MAP_FIXED = allocate at starting addr or fail
    - "Use of this option is discouraged"
  - MAP_PRIVATE = for my process only
  - MAP_ANONYMOUS = don't map to any file

Kernel memory model

- In Linux, jumping into kernel does not remap pages
  - This would add a lot of expense to syscalls
- So, if we mmap page 0 and then jump into the kernel, that page will still be in memory
  - Uh oh...

Exploiting a NULL pointer error

- Recipe for exploit:
  - Find a function call, null pointer error in the kernel
    - There are probably lots of them
  - Get access to a local user account
  - mmap the 0 page
  - load whatever code you want executed there
  - trigger the null pointer error

  - ⇒ kernel jumps to code you control

  - Privilege escalation

Impractical solution

- Eliminate all NULL pointer errors
  - Tony Hoare, "Null References: The Billion Dollar Mistake"
  - Could try to remove use of NULL in C
    - Pervasive use in system, library, and many coding idioms makes this unrealistic
  - Can try to give programmers tools to detect null pointer errors
    - But practical tools will always miss some errors

Actual solution

- Don't allow 0 page to be remapped
  - Modern systems have a check in the kernel's mmap function that requires the mapped page to be some minimal address:

```
$ cat /proc/sys/vm/mmap_min_addr
4096
```

Lesson learned?

- Vulnerability results from interaction between
  - Language feature (NULL)
  - OS feature (mmap)

- Lesson: it's hard to anticipate all possible interactions

Printing strings in C

- How to print a string in C:

```
printf("%s", some_string);
```

- That's too much typing...why don't we just write:

```
printf(some_string);
```

  - We saved 5 characters and some whitespace!

Format-string vulnerabilities

- What happens if an attacker can control that string?

```
printf(some_string_from_network);
```

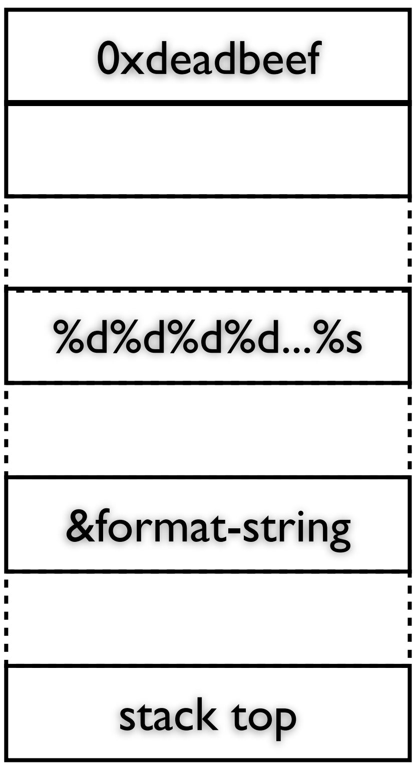
  - Could pass in “%s%s%s”
    - For each %s, printf looks for corresponding arg on stack, treats it as pointer to char, derefs pointers, and prints until hitting NULL
    - Args not there, so printf will chase garbage pointers
    - ⇒ very likely to produce segfault

What can we do with this?

- Crash program, as above
- Print out data in memory
  - Display sensitive information (e.g., plaintext passwords or keys)
  - Find pointer values, offsets, return addresses, etc
    - Help make further exploitation easier, defeat ASLR
- Can even print arbitrary memory addresses

Reading arbitrary memory

- x86 stack grows downward
- Suppose format string itself lives on stack
  - E.g., perhaps it goes in a stack-allocated buffer



printf("\xde\xad\xbe\xef%d%d...%d%s")

- Each format specifier will cause printf to look for the next arg on the stack
- Args pushed right-to-left
- So, with the right # of %d's, the %s will match the 0xdeadbeef value
  - printf will start printing memory from that addr

Writing to memory

```
NAME
    printf, fprintf, sprintf, snprintf, asprintf, dprintf, vprintf, vfprintf,
    vsprintf, vsnprintf, vasprintf, vdprintf -- formatted output conversion

...

    The conversion specifiers and their meanings are:
...
    n      The number of characters written so far is stored into the integer
           indicated by the int * (or variant) pointer argument.  No
           argument is converted.
```

- Oops!

Writing arbitrary memory

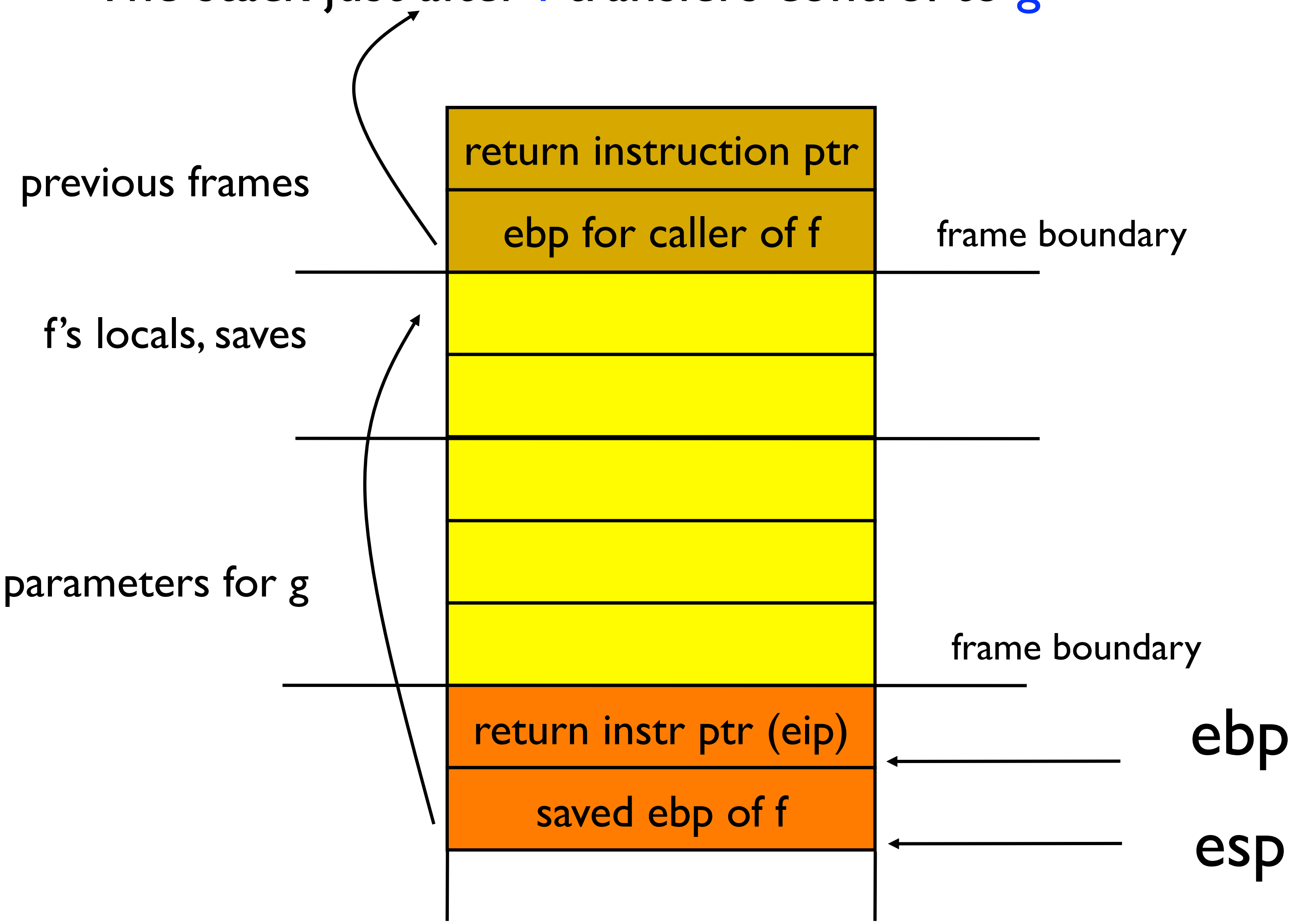
- Same trick as with reading memory
  - printf("\xde\xad\xbe\xef%d%d...%d%n")
  - Add enough %d's so %n corresponds to 0xdeadbeef
- Control the value by controlling # bytes written
  - To write large value, write one byte at a time
    - printf("\xde\xad\xbe\xef...%n...\xde\xad\xbe\xf0...%n...\xde\xad\xbe\xf1...%n...\xde\xad\xbe\xf2...%n")
    - Note must write increasing sequence of values
  - Fortunately, x86 has *lots* of instructions!

Solution and lessons

- Only use trusted strings in format positions
  - Harder than it seems, e.g., internationalization
  - Input validation problem, similar to SQL injection
- *Potential* problems known for a long time
  - Only became well known as vulnerability in 2000
- Vulnerability results from interaction between
  - Language feature (stack)
  - Library feature (printf)
- Lesson?

The x86 Stack Frame

- The stack just after f transfers control to g



Based on Fig 6-1 in Intel ia-32 manual

x86 Calling Convention

- To call a function
  - Push parameters for function onto stack
  - Invoke CALL instruction to
    - Push current value of eip onto stack
      - I.e., save the program counter
    - Start executing code for called function
  - Callee pushes ebp onto stack to save it
- When a function returns
  - Put return value in eax
  - Invoke LEAVE to pop stack frame
    - Set esp to ebp
    - Restore ebp that was saved on stack and pop it off the stack
  - Invoke RET instruction to load return address into eip
    - I.e., start executing code where we left off at call

Example

```c
int f(int a, int b) {
  return a + b;
}

int main(void) {
  int x;

  x = f(3, 4);
}
```

```
f:
        pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        leave
        ret
main:
        ...
        subl    $8, %esp
        pushl   $4
        pushl   $3
        call    f
     l: addl    $16, %esp
        movl    %eax, -4(%ebp)
        leave
        ret
```

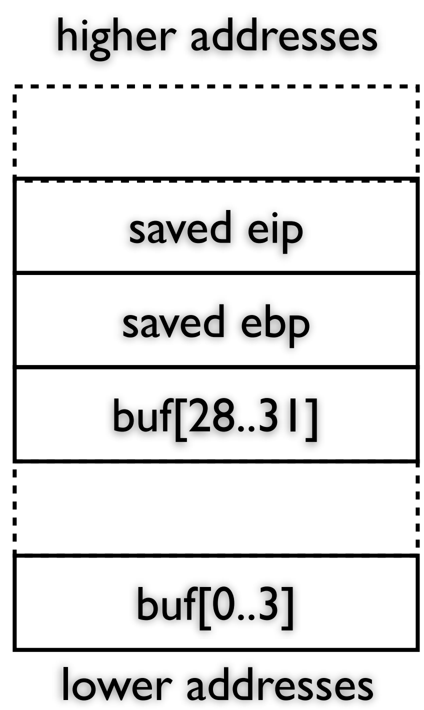
gcc -S a.c

Buffer overflows

```
void f() {
  char buf[32];

  buf = gets();
}
```

- gets() may return more than 32 characters
  - Might write past the end of buf
- What's going to happen on the stack?
  - Could overwrite eip
  - When f() returns, jump to attacker-specified address



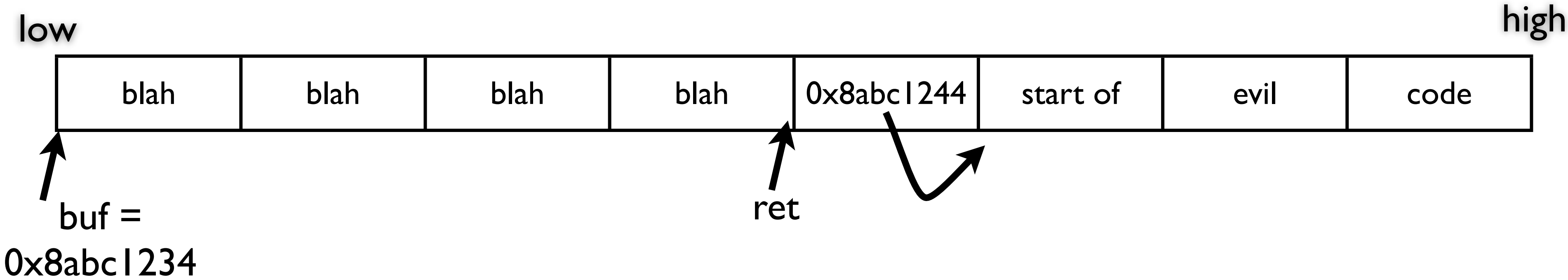
Problematic functions

- Lots of C functions do not check buffer sizes
  - strcpy(dst, src) — copy null-terminated src to dst
    - Does not check size of dst
  - strcat(s1, s2) — concatenate null-terminated strings
    - Adds to end of s1, but doesn't check size
  - sprintf(str, fmt, ...) — print to string
    - Does not check size of str
- Also, many large systems have custom wrappers around these functions, or similar custom fns
  - Hard to avoid something that's so widely used!

Lesson learned?

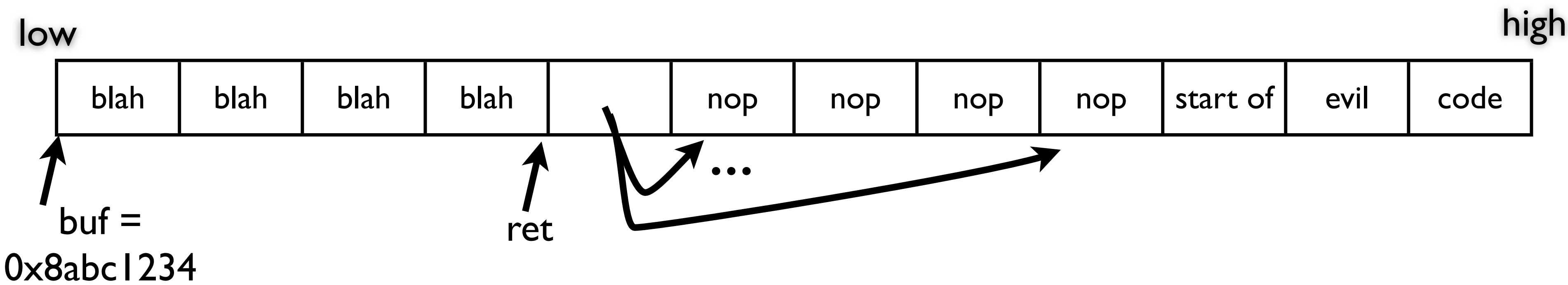
- Buffer overflows caused by interaction of
  - No array bounds checks in C
    - It's faster that way!
  - Stack-based calling convention
  - Arrays are stack-allocated
    - Can exploit buffer overflows on heap, but harder, less systematic
- $\Rightarrow$ type and memory safety are good properties

Exploiting buffer overflows



- Idea #1: Overwrite ret with address in overflowed buffer
  - So whatever code we put in the buffer will execute
- Problem: need to know address on stack
  - Might vary some from run to run, e.g., if environment variables change, system configuration slightly different, etc

NOP sled



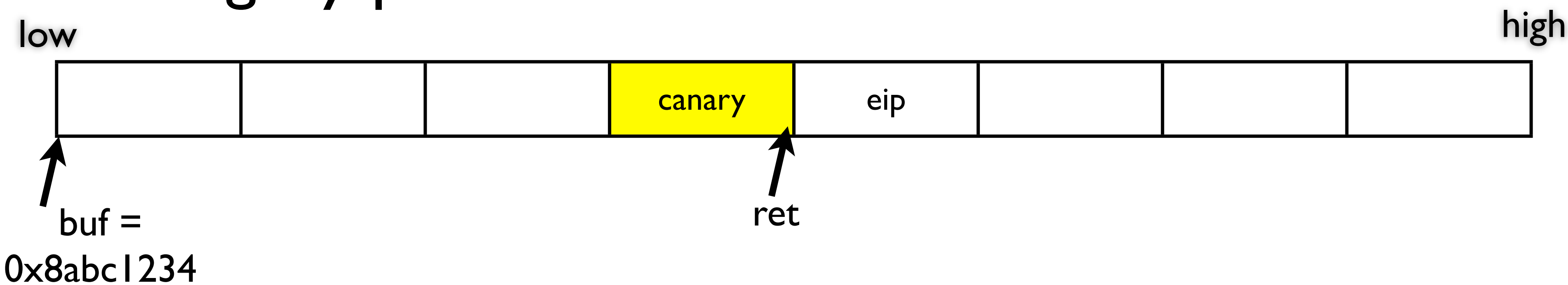
- Idea #2: Add a bunch of no-ops at the front of the code, so we don't need to get the exact address

Preventing buffer overflows

- Eliminate them in the source code
  - Use C safely
    - This is much harder than it seems...
  - Don't use C!
  - Use static analysis to detect and prevent vulnerabilities
    - Can never be perfect
- Modify the system to deter exploits
  - Stack canaries
  - Address space layout randomization (ASLR)
  - Non-executable stack (NX)

StackGuard (canaries)

- Embed random "canaries" in stack and ensure integrity prior to function return



- Limitations
  - Adds overhead
  - Only works for stack overflows
  - Canaries should be random to avoid guessing
  - Canaries could still be discovered via other means (e.g., format string vulnerabilities)

ASLR

- Randomize starting location of the stack
  - Makes it hard for overwritten eip to go to attacker-controlled location
- Commonly in use today
- Limitations
  - Some implementations don't have enough randomness
    - Can be brute-forced
  - Other vulnerabilities could reveal stack addresses

NX

- Mark memory page containing stack data as "non-executable" in the page tables
  - Processor will fault if try to set eip to point anywhere on the stack
- Commonly in use today
- Limitations
  - Only protects stack
  - Can be defeated with fancier exploit techniques...

Return-oriented programming

- See next slide deck