Stepping back

What do these attacks have in common?

1. The **attacker** is able to **control some data** that is used by the program

2. The use of that data **permits unintentional access to some memory area** in the program
    - past a buffer
    - to arbitrary positions on the stack

Outline

- **Memory safety** and **type safety**
  - Properties that, if satisfied, ensure an application is immune to memory attacks
- Automatic defenses
  - **Stack canaries**
  - Address space layout randomization (**ASLR**)
- Return-oriented programming (**ROP**) attack
  - How Control Flow Integrity (**CFI**) can defeat it
- **Secure coding**

Memory Safety

Low-level attacks enabled by a lack of **Memory Safety**

A memory safe program execution:

1. only **creates pointers** through **standard means**
   - **`p = malloc(…)`**, or **`p = &x`**, or **`p = &buf[5]`**, etc.
2. only uses a pointer to **access memory** that **"belongs" to that pointer**

Combines two ideas:

temporal safety and **spatial safety**

Spatial safety

- View pointers as triples (**p**,***b***,***e***)
  - **p** is the actual pointer
  - ***b*** is the base of the memory region it may access
  - ***e*** is the extent (bounds) of that region
- **Access allowed** *iff* ***b*** ≤ **p** ≤ **e**-`sizeof`(`typeof`(**p**))
- Operations:
  - Pointer arithmetic increments **p**, leaves ***b*** and ***e*** alone
  - Using `&`: ***e*** determined by size of original type

Examples

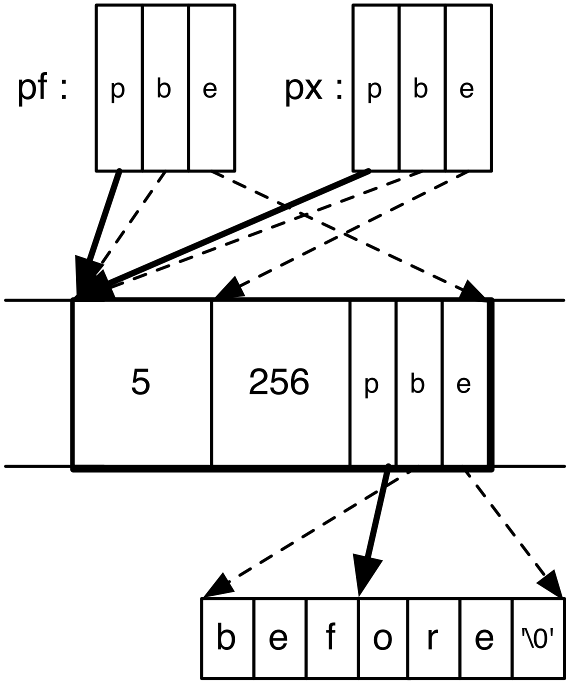
```
int x;           // assume sizeof(int)=4
int *y = &x;     // p = &x, b = &x, e = &x+4
int *z = y+1;    // p = &x+4, b = &x, e = &x+4
*y = 3;          // OK: &x ≤ &x ≤ (&x+4)-4
*z = 3;          // Bad: &x ≤ &x+4 ≰ (&x+4)-4
```

```
struct foo {
 char buf[4];
 int x;
};
```

```
struct foo f = { “cat”, 5 };
char *y = &f.buf; // p = b = &f.buf, e = &f.buf+4
y[3] = ‘s’;      // OK: p = &f.buf+3 ≤ (&f.buf+4)-1
y[4] = ‘y’;      // Bad: p = &f.buf+4 ≰ (&f.buf+4)-1
```

Visualized example

```
struct foo {
  int x;
  int y;
  char *pc;
};
struct foo *pf = malloc(...);
pf->x = 5;
pf->y = 256;
pf->pc = "before";
pf->pc += 3;
int *px = &pf->x;
```



No buffer overflows

- A buffer overflow violates spatial safety

```
void copy(char *src, char *dst, int len)
{
  int i;
  for (i=0;i<len;i++) {
    *dst = *src;
    src++;
    dst++;
  }
}
```

  - Overrunning the bounds of the source and/or destination buffers implies either **`src`** or **`dst`** is illegal

Temporal safety

- A **temporal safety violation** occurs when trying to **access undefined memory**
  - Spatial safety assures it was to a legal region
  - Temporal safety assures that region is still in play
- Memory regions either **defined** or **undefined**
  - Defined means allocated (and active)
  - Undefined means unallocated, uninitialized, or deallocated
- Pretend memory is infinitely large (we never reuse it)

No dangling pointers

- Accessing a freed pointer violates temporal safety

```
int *p = malloc(sizeof(int));
*p = 5;
free(p);
printf(“%d\n”,*p); // violation
```

The memory dereferenced no longer belongs to p.

- Accessing uninitialized pointers is similarly not OK:

```
int *p;
*p = 5; // violation
```

Most languages memory safe

- The easiest way to avoid all of these vulnerabilities is to **use a memory safe language**

- Modern languages are memory safe
  - Java, Python, C#, Ruby
  - Haskell, Scala, Go, Objective Caml, Rust
- In fact, these **languages are type safe,** which is even **better** (more on this shortly)

Memory safety for C

- **C/C++ here to stay**. While not memory safe, you can write memory safe programs with them
  - The problem is that there is no guarantee
- Compilers could add **code to check for violations**
  - An out-of-bounds access would result in an immediate failure, like an *ArrayBoundsException* in Java
- This idea has been around for more than 20 years. **Performance has been the limiting factor**
  - Work by Jones and Kelly in 1997 adds 12x overhead
  - Valgrind memcheck adds 17x overhead

Progress

Research has been **closing the gap**

- **CCured** (2004), 1.5x slowdown
  - But no checking in libraries
  - Compiler rejects many safe programs

ccured

- **Softbound/CETS** (2010): 2.16x slowdown
  - Complete checking
  - Highly flexible

SoftBound
CETS

- Coming soon: **Intel MPX** hardware
  - Hardware support to make checking faster

https://software.intel.com/en-us/blogs/2013/07/22/intel-memory-protection-extensions-intel-mpx-support-in-the-gnu-toolchain

intel

Type Safety

Type safety

- Each object is ascribed a **type** (`int`, pointer to `int`, pointer to function), and
- Operations on the object are always *compatible* with the object's type
  - Type safe programs do not "go wrong" at run-time
- **Type safety** is **stronger** than memory safety

```
int (*cmp)(char*,char*);
int *p = (int*)malloc(sizeof(int));
*p = 1;
cmp = (int (*)(char*,char*))p;
cmp(“hello”,”bye”); // crash!
```

Memory safe,
but not type safe

Dynamically Typed Languages

- **Dynamically typed languages**, like Ruby and Python, which do not require declarations that identify types, can be viewed as **type safe** as well
- Each **object** has **one type: Dynamic**
  - Each operation on a Dynamic object is permitted, but *may be unimplemented*
  - In this case, it *throws an exception*

Well-defined (but unfortunate)

Enforce invariants

- Types really show their strength by **enforcing invariants** in the program
- Notable here is the enforcement of **abstract types**, which characterize modules that keep their **representation hidden** from clients
  - As such, we can reason more confidently about their **isolation** from the rest of the program

For **more on type safety**, see http://www.pl-enthusiast.net/2014/08/05/type-safety/

Types for Security

- **Type-enforced invariants can relate directly to security properties**
  - By expressing stronger invariants about data's privacy and integrity, which the type checker then enforces
- **Example**: **Java with Information Flow (JIF)**

```
int{Alice→Bob} x;
int{Alice→Bob, Chuck} y;
x = y; //OK: policy on x is stronger
y = x; //BAD: policy on y is not
       //as strong as x
```

Types have *security labels*

Labels define what information flows allowed

http://www.cs.cornell.edu/jif

Why not type safety?

- **C/C++** often chosen **for performance** reasons
  - Manual memory management
  - Tight control over object layouts
  - Interaction with low-level hardware
- **Typical enforcement** of type safety is **expensive**
  - **Garbage collection** avoids temporal violations
    - Can be as fast as malloc/free, but often uses much more memory
  - **Bounds** and **null-pointer checks** avoid spatial violations
  - **Hiding representation** may **inhibit optimization**
    - Many C-style casts, pointer arithmetic, & operator, not allowed

Not the end of the story

- **New languages** aiming to **provide similar features** to C/C++ while **remaining type safe**
  - Google's **Go**
  - Mozilla's **Rust**
  - Apple's **Swift**
- **Most applications do not need C/C++**
  - Or the risks that come with it

These languages may be the future of low-level programming

Avoiding exploitation

Other defensive strategies

Until C is memory safe, what can we do?

Make the bug harder to exploit

- Examine necessary steps for exploitation, make one or more of them difficult, or impossible

Avoid the bug entirely

- Secure coding practices
- Advanced code review and testing
    - E.g., program analysis, penetrating testing (fuzzing)

Strategies are **complementary**: Try to **avoid bugs**, *but* **add protection** if some slip through the cracks

Avoiding exploitation

Recall the steps of a stack smashing attack:

- Putting attacker code into the memory (no zeroes)
- Getting `%eip` to point to (and run) attacker code
- Finding the return address (guess the raw addr)

How can we make these attack steps more difficult?

- **Best case**: Complicate exploitation by changing the the **libraries**, **compiler** and/or **operating system**
  - Then we don't have to change the application code
  - *Fix is in the architectural design, not the code*

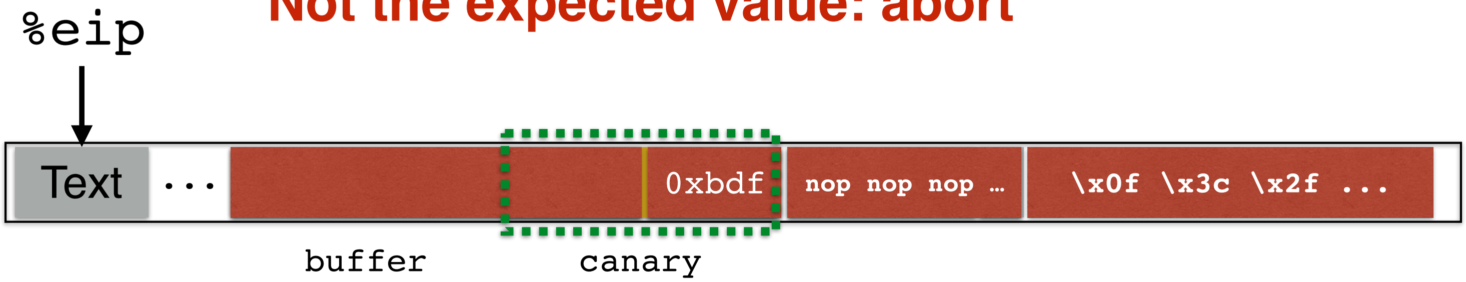
Detecting overflows with canaries

19th century coal mine integrity

- Is the mine safe?
- Dunno; bring in a canary
- If it dies, abort!

We can do the same for stack integrity

Detecting overflows with canaries



Not the expected value: abort

What value should the canary have?

Canary values

From StackGuard [Wagle & Cowan]

1. Terminator canaries (CR, LF, NUL (i.e., 0), -1)
   - Leverages the fact that scanf etc. don't allow these
2. Random canaries
   - Write a new random value @ each process start
   - Save the real value somewhere in memory
   - Must write-protect the stored value
3. Random XOR canaries
   - Same as random canaries
   - But store canary XOR some control info, instead

Recall our challenges

- Putting code into the memory (no zeroes)
  - **Defense**: Make this detectable with **canaries**
- Getting %eip to point to (and run) attacker code
- Finding the return address (guess the raw addr)

Recall our challenges

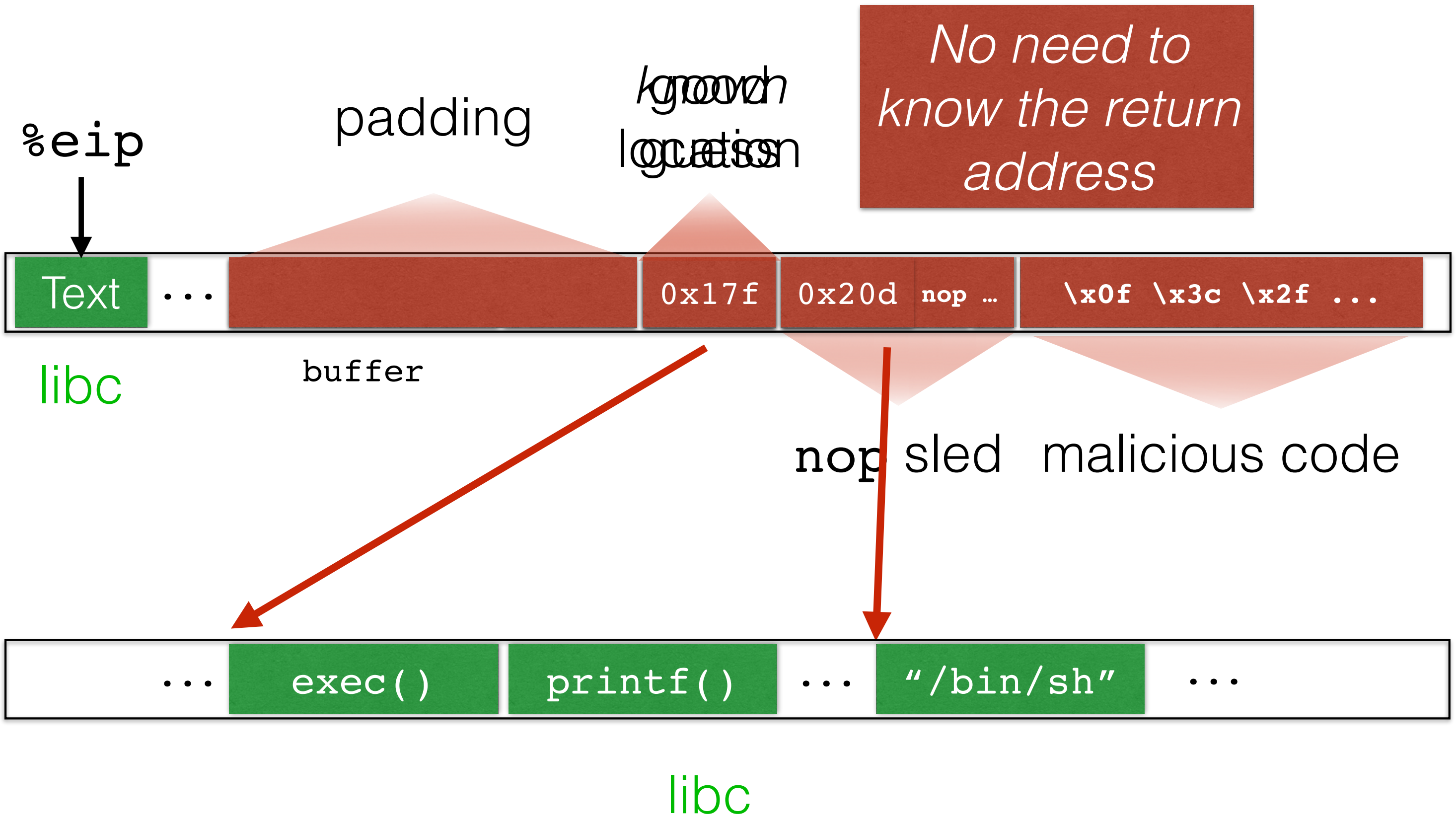
- Putting code into the memory (no zeroes)
  - **Defense**: Make this detectable with **canaries**

- Getting %eip to point to (and run) attacker code
  - **Defense**: **Make stack (and heap) non-executable**

- Finding the raw addr)

So: even if canaries could be bypassed, no code loaded by the attacker can be executed (will panic)

Return-to-libc


No need to know the return address

Recall our challenges

- Putting code into the memory (no zeroes)
  - Defense: Make this detectable with canaries
- Getting %eip to point to (and run) attacker code
  - Defense: Make stack (and heap) non-executable
  - **Defense: Use Address-space Layout Randomization**
- Finding t addr)

Randomly place standard libraries and other elements in memory, making them harder to guess

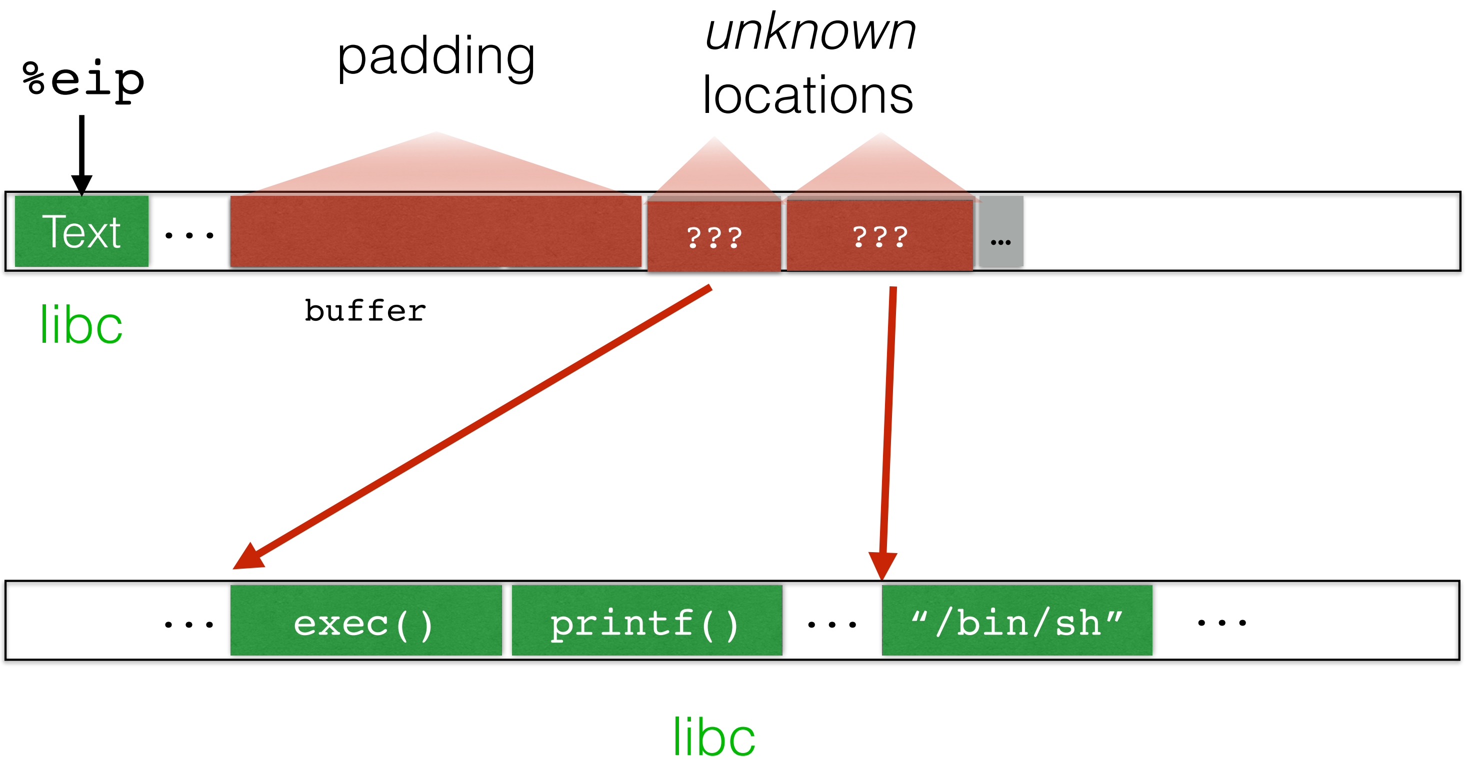
Recall our challenges

- Putting code into the memory (no zeroes)
  - Defense: Make this detectable with canaries

- Getting %eip to point to (and run) attacker code
  - Defense: Make stack (and heap) non-executable
  - Defense: Use Address Space Layout Randomization

- Finding the return address (guess the raw addr)
  - **Defense: Use Address-space Layout Randomization**

Return-to-libc, thwarted



ASLR today

- **Available on modern operating systems**
  - Available on Linux in 2004, and adoption on other systems came slowly afterwards; **most by 2011**
- Caveats:
  - **Only shifts the offset** of memory areas
    - Not locations within those areas
  - **May not apply to program code**, just libraries
  - **Need sufficient randomness**, or can brute force
    - 32-bit systems typically offer 16 bits = 65536 possible starting positions; sometimes 20 bits. Shacham demonstrated a brute force attack could defeat such randomness in 216 seconds (on 2004 hardware)
    - **64-bit systems more promising**, e.g., 40 bits possible