Memory safety, continued

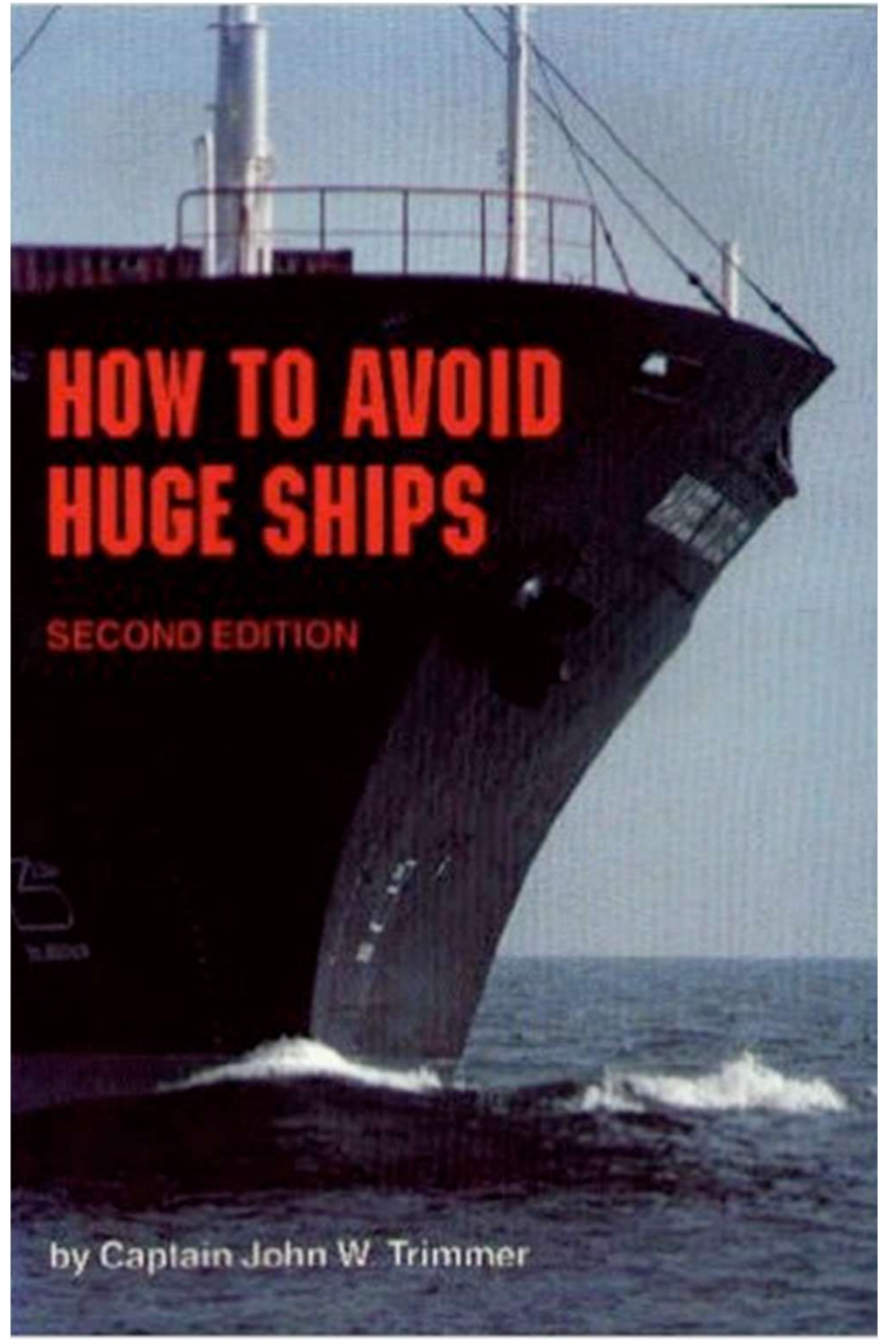
With material from Mike Hicks,
Dave Levin and Michelle Mazurek

http://images.myhardhatstickers.com/img/lg/H/Everybody-Safety-Hard-Hat-Label-HH-0115.gif

Today

- Avoiding exploitation
  - Memory violations possible but not harmful

Avoiding exploitation



https://www.amazon.com/Avoid-Huge-Ships-John-Trimmer/dp/0870334336

What can we do to protect against buffer overflow exploits?

- Make bugs **harder to exploit**
  - Crash but not code execution

- **Avoid bugs** with better programming
  - Secure coding practices, code review, testing

Better together: Try to avoid bugs, *but also* add protection if some slip through

Avoiding exploitation

Recall the steps of a stack smashing attack:

- Putting attacker code into memory
  - (No zeroes or other stoppers)
- Getting `%eip` to point to attacker code
- Finding the return address

How can we make these attack steps more difficult?

Avoiding exploitation

Recall the steps of a stack smashing attack:

- Putting attacker code into memory
  - (No zeroes or other stoppers)
- Getting `%eip` to point to attacker code
- Finding the return address

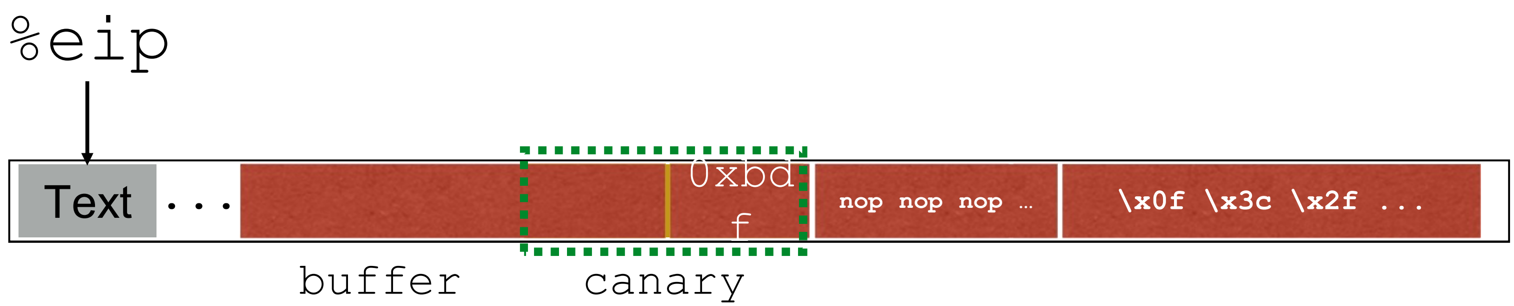
How can we make these attack steps more difficult?

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



Check canary just before every function return.

Not the expected value: abort!

What value should the canary have?

Canary values

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

Other canary tricks

- Put canaries in heap metadata
- Reorganize locals to put buffers above pointers
  - Buffers can only overwrite themselves, canary
  - [ProPolice]
- Global return stack [StackShield]
  - Copy ret address from separate stack every time

Canary weaknesses

- Overwrite function pointer
- Overwrite local variable pointer to indirectly reference eip
- Anything not stack (heap, etc.)
- Bad randomization
- Memory is not necessarily secret
  - Buffer overreads

https://yy1.staticflickr.com/6127/5953402062_a2e35ff01b_z.jpg

Overread example

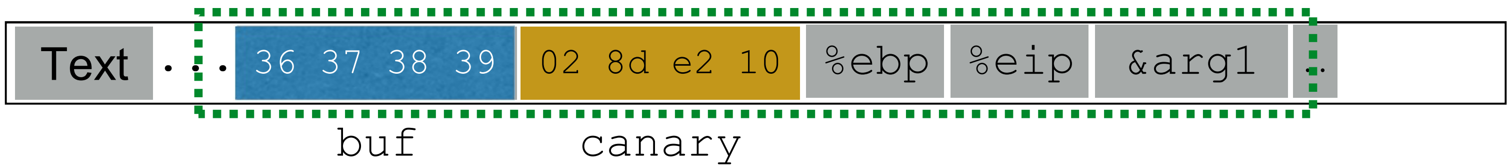
From Strackx et al.

```
void vulnerable(char *name_in)
{
    char buf[10];
    strncpy(buf, name_in, sizeof(buf));
    printf("Hello, %s\n", buf);
}
```

name_in = “0123456789ABC”

does not append NULL

prints until NULL



- Strncpy is “safe” because it won’t overwrite
  - But string not properly terminated

Avoiding exploitation

Recall the steps of a stack smashing attack:

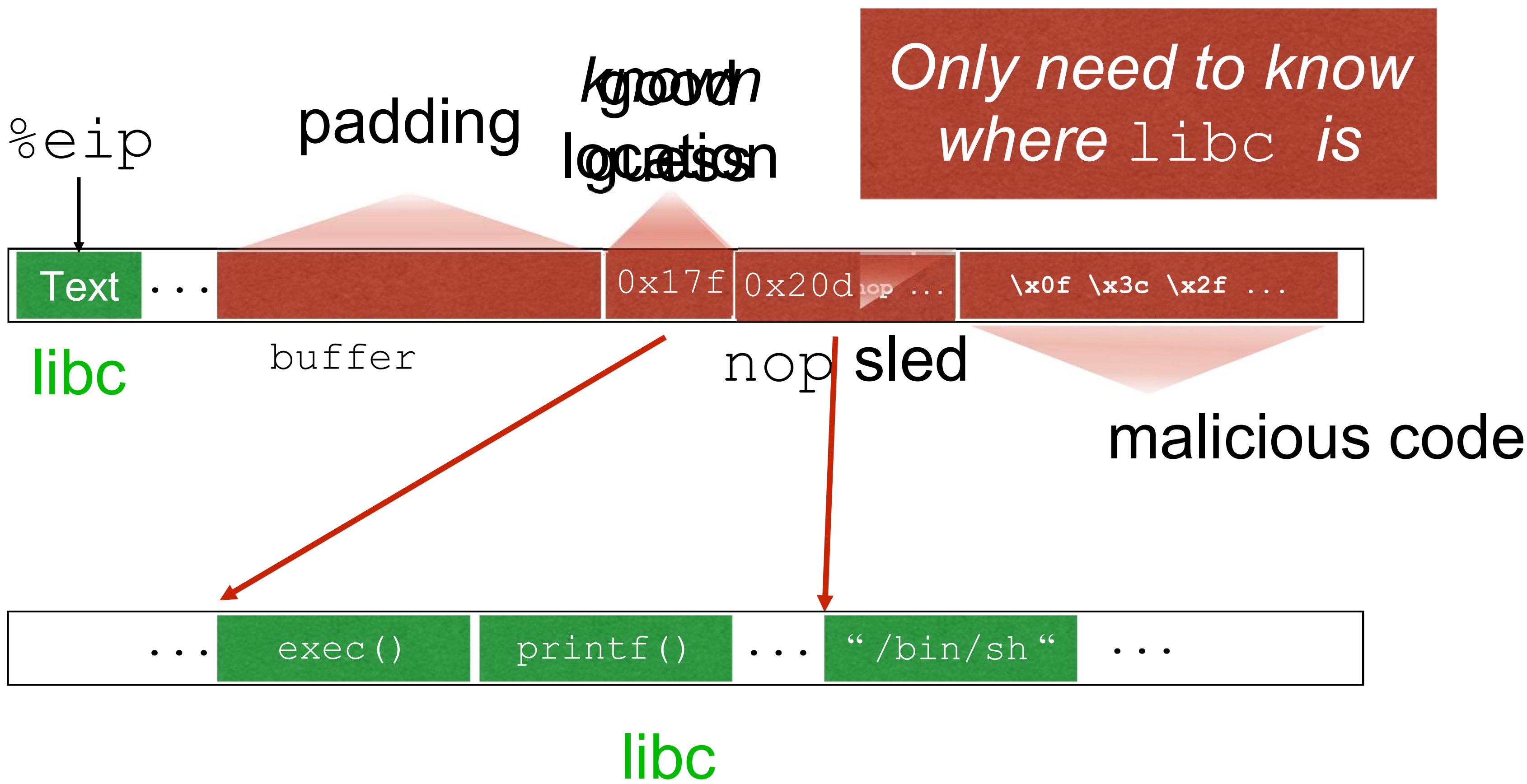
- Putting attacker code into memory
  Defense: Stack Canaries
- Getting `%eip` to point to attacker code
- Finding the return address

How can we make these attack steps more difficult?

- Goal: Don't run attacker code
- Defense: Make stack non-executable
  - Try to jump to attacker shellcode in the stack, panic instead

http://www.ipadforums.net/wallpapers/data/2/DontPanic.png

Return-to-libc



Avoiding exploitation

Recall the steps of a stack smashing attack:

- Putting attacker code into memory

  Defense: Stack Canaries

- Getting `%eip` to point to attacker code

  Defense: Non-executable stack (kind of)

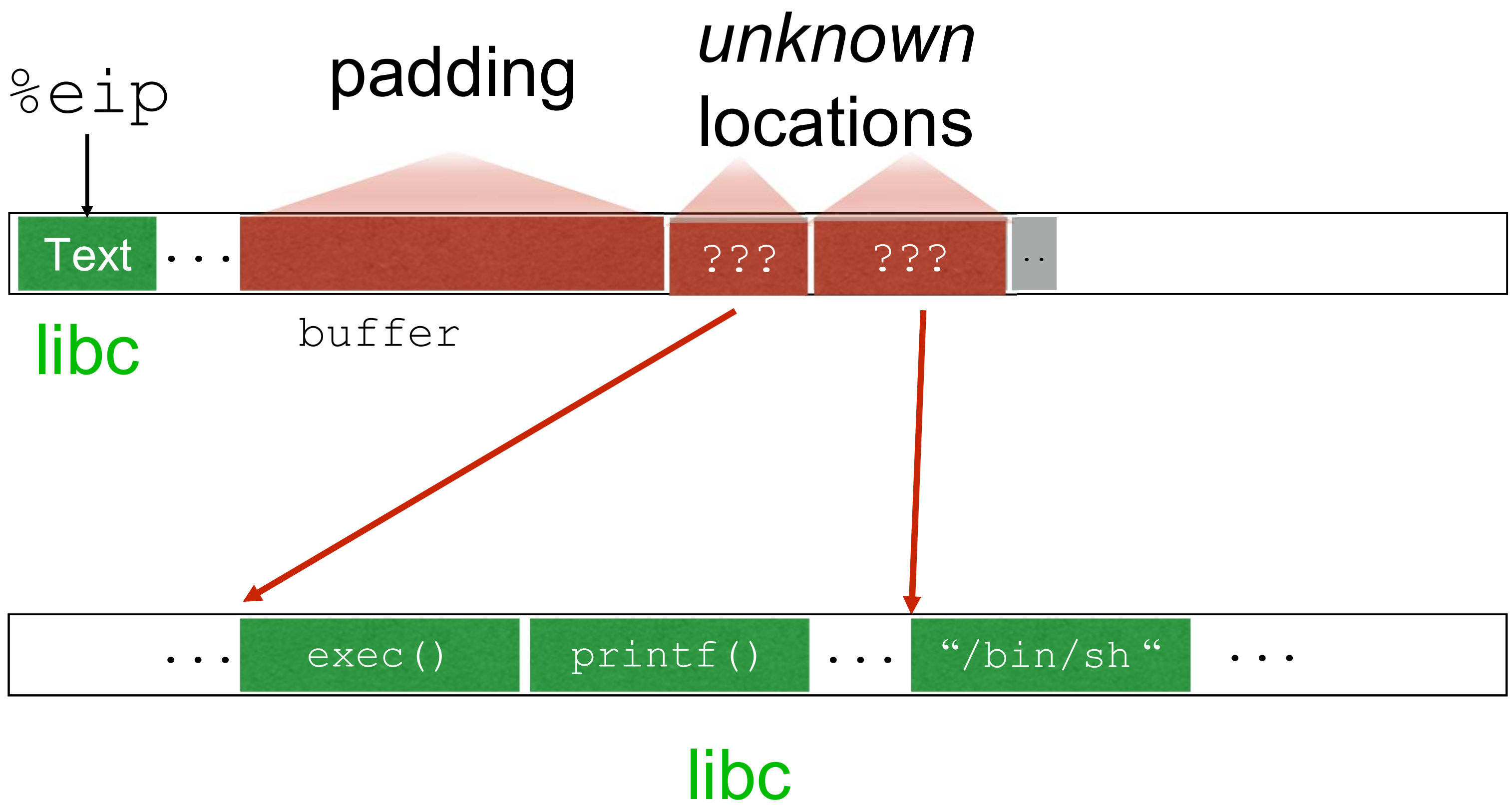
- Finding the return address

How can we make these attack steps more difficult?

Address-space layout randomization

- Randomly place some elements in memory
- Make it hard to find libC functions
- Make it hard to guess where stack (shellcode) is

Return-to-libc, thwarted



ASLR today

- Available on modern operating systems
  - Linux in 2004, other systems slowly afterwards; **most by 2011**
- Caveats:
  - **Only shifts the offset** of memory areas
    - Not locations within those areas
    - Possible to use a read exploit to find it
  - **May not apply to program code**, just libraries
  - **Need sufficient randomness**, or can brute force
    - 32-bit systems: typically16 bits = 65536 possible starting positions; sometimes 20 bits. Shacham brute force attack could defeat this in 216 seconds (2004 hardware)
    - 64-bit systems more promising, e.g., 40 bits possible

Cat and mouse

- Defense: **Make stack/heap non-executable** to prevent injection of code
  - Attack response: **Return to libc**
- Defense: **Hide the address of desired libc code** or **return address** using ASLR
  - Attack response: **Brute force search** or **information leak**
- Defense: **Avoid/limit use of libc code**
  - Attack response: Construct needed functionality using **return oriented programming (ROP)**