

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

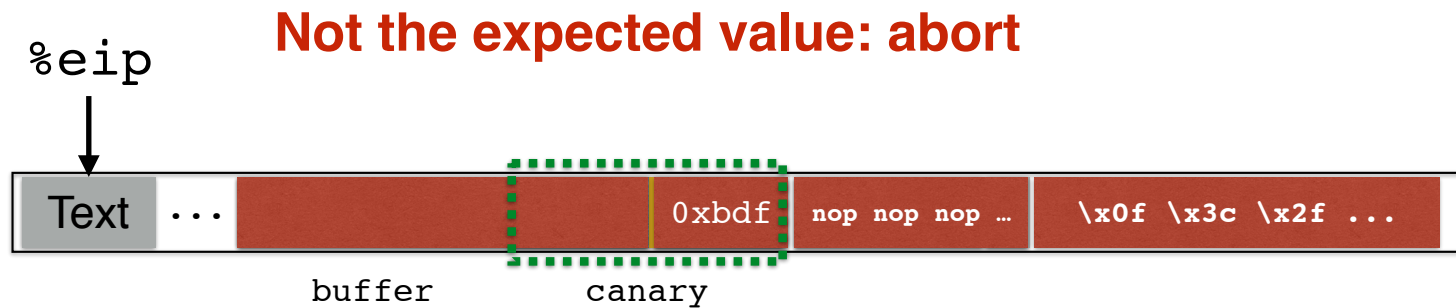
## 19th century coal mine integrity

- 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



**What value should the canary have?**

# Canary values

**From StackGuard [Wagle & Cowan]**

1. Terminator canaries (CR, LF, NULL, -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**

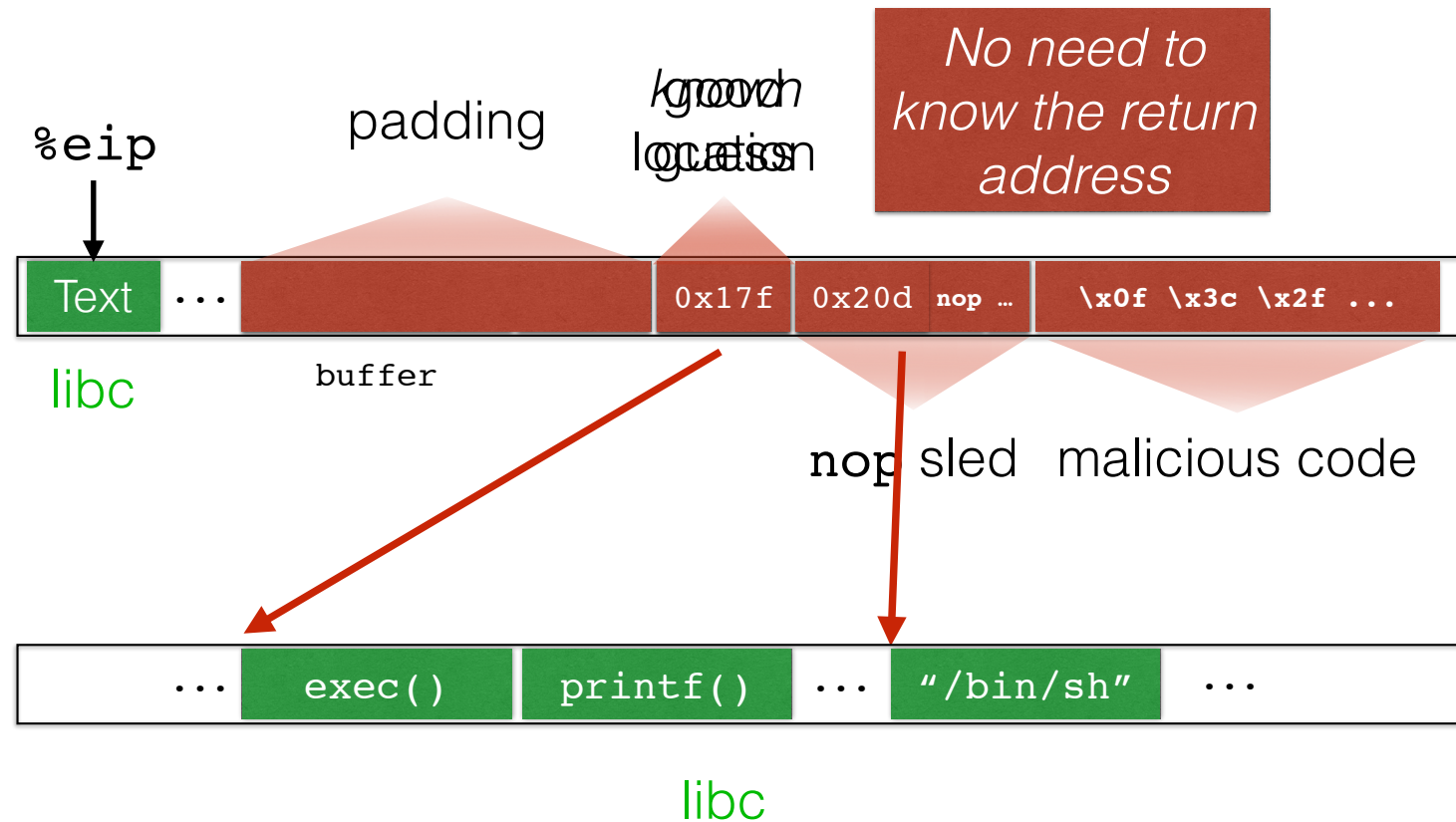
- Finding the

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

raw addr)



# Return-to-libc



# Recall our challenges

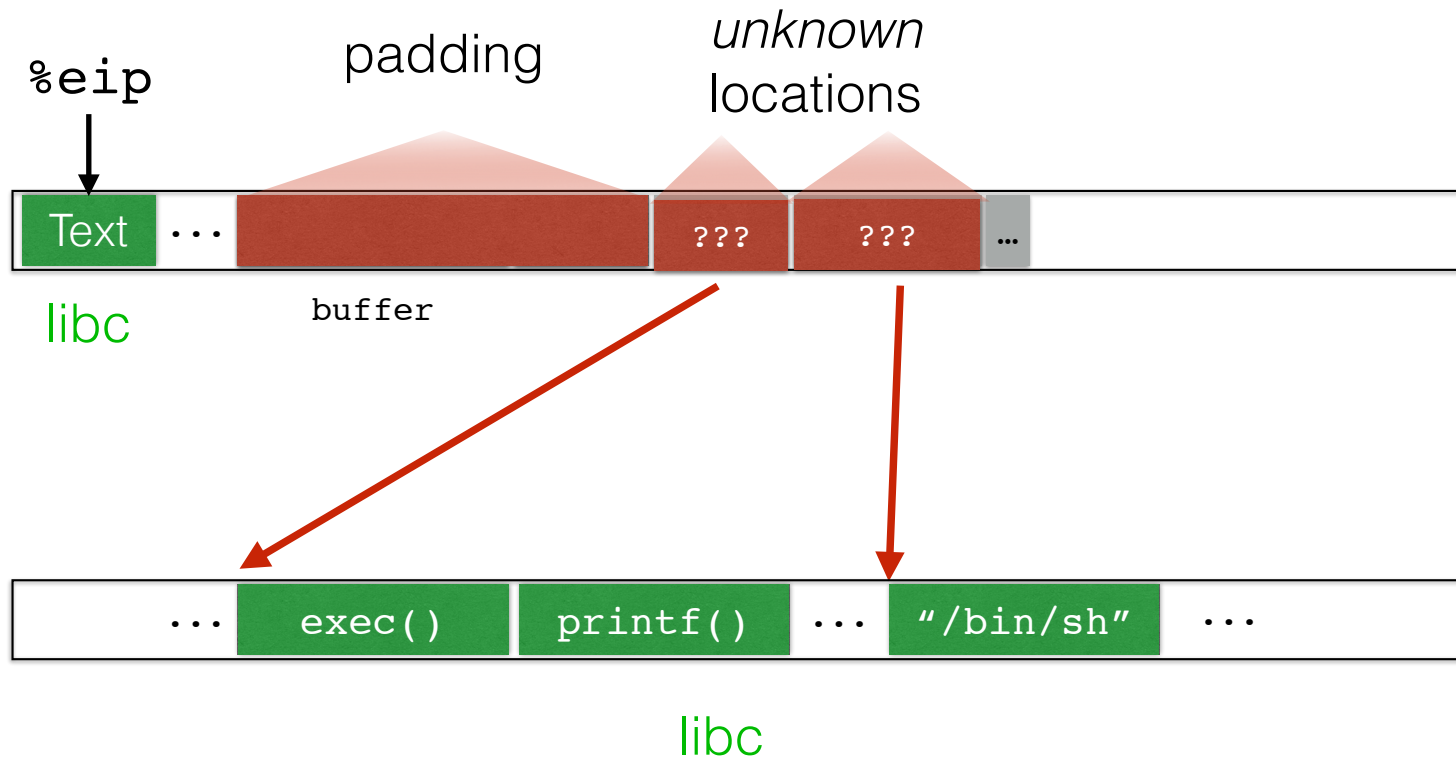
- Putting code into the memory (no zeroes)
  - Defense: Make this detectable with canaries
- Getting %eip to point to (and run) attacker code
  - Defense: Make stack (and heap) non-executable
  - **Defense: Use Address-space Layout Randomization**

- Finding the address of 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