Information Security Principles; Software Exploits and Defenses

COS 518: *Advanced Computer Systems*
Lecture 8

Kyle Jamieson

[Credits: Content adapted from M. Freedman, B. Karp, N. Zeldovich]

Today

1. **Introduction to the design of secure systems**

2. Stack smashing buffer overflows and countermeasures

3. Heap smashing buffer overflows and countermeasures

Building secure systems

- Secure = achieves some property despite attacks by adversaries

- High-level plan for thinking about secure systems:

1. ***Policy:*** The goal you want to achieve
   - *e.g.*, only Alice should read file *F*
   - Common goals: confidentiality, integrity, availability
2. ***Threat model:*** What the attacker can do
   - *e.g.* can guess passwords, cannot physically steal our server
3. ***Mechanism:*** Software/hardware you system uses to enforce security
   - *e.g.* user accounts, passwords, trusted hardware

Building secure systems is hard

- *e.g.* grades.txt stored on a shared file server
  - Policy: Only course staff may read and write the grades file

- Easy to implement the positive aspect of the policy

- But **security is a negative goal**
  - Want no tricky way for non-staff to get at file
  - *e.g.* exploit a bug in file server's code
    - Guess TA's password
    - Steal instructor's laptop, maybe it has a local copy of the grades file.
    - Intercept grades when they are sent over the network to the university registrar
    - Get a job in the university registrar's office

Building secure systems is hard (2)

- **Cost asymmetry**
  - "Fortune favors the attacker"
  - Secure system designer must protect everything
  - Attacker need only find one "hole"

- Can't get policies/threats/mechanisms right on first try

- What to do? Usually iterate: design, watch attacks, update

- What's the point if we can't achieve perfect security?
  - It's rarely required
  - Make the cost of the attack > value of the information
  - Today we'll look at ways to cut off entire classes of attacks
  - Success: Popular attacks *ca.* 10 years ago no longer fruitful

What can go wrong? (1/3)

- **Problems with the policy itself**
  - System enforces policy, but the policy is inadequate

- e.g. Wired editor Mat Honan's Amazon, Apple, Gmail accounts
  - Someone wanted to break into Gmail
  - Gmail password reset: Send verification link to backup email address
    - Mat's was his Apple @me.com account
  - Apple password reset: Need billing address, last four of cc
  - Amazon's password reset e-mail includes last 4 digits of all your registered credit cards
  - Call Amazon tech support; you can persuade them to add a new e-mail to any account

- Now attacker can reset Apple password → read Gmail reset e-mail, **reset Gmail password**

What can go wrong? (2/3)

- **Problems with the threat model**
  - Designer assumed an attack wasn't feasible or didn't anticipate the attack

- *e.g.* Browser trusts all SSL certificate authorities
  - 2011: Two CAs compromised
  - 2012: CA inadvertently issued a root certificate valid for any domain

- *e.g.* Assuming your hardware is trustworthy
  - Firmware malware
  - NSA hardware interdiction

What can go wrong? (3/3)

- **Problems with the mechanism:** bugs

- Example: Missing access control checks in Citigroup's credit card web site
  - Login page asks for username and password.
  - The URL of the account info page included some numbers. *e.g.* `x.citi.com/id=1234`
    - The numbers were (related to) the user's account number.
  - Adversary tried different numbers, got different people's account info
  - The server didn't check that you were logged into that account!

- Lesson: programmers tend to think only of intended operation.

Today

1. Introduction to the design of secure systems

2. **Stack smashing buffer overflows and countermeasures**

3. Heap smashing buffer overflows and countermeasures

Imperfect software

- Fundamentally hard to prevent all programmer error

- C and C++ particularly dangerous
  - Allow arbitrary manipulation of pointers
  - Require programmer-directed allocation and freeing of memory
  - Offer high performance, so extremely prevalent, especially in network servers and OSes

- Java offers memory safety, but not a panacea
  - JRE written in (many thousands of lines of) C!

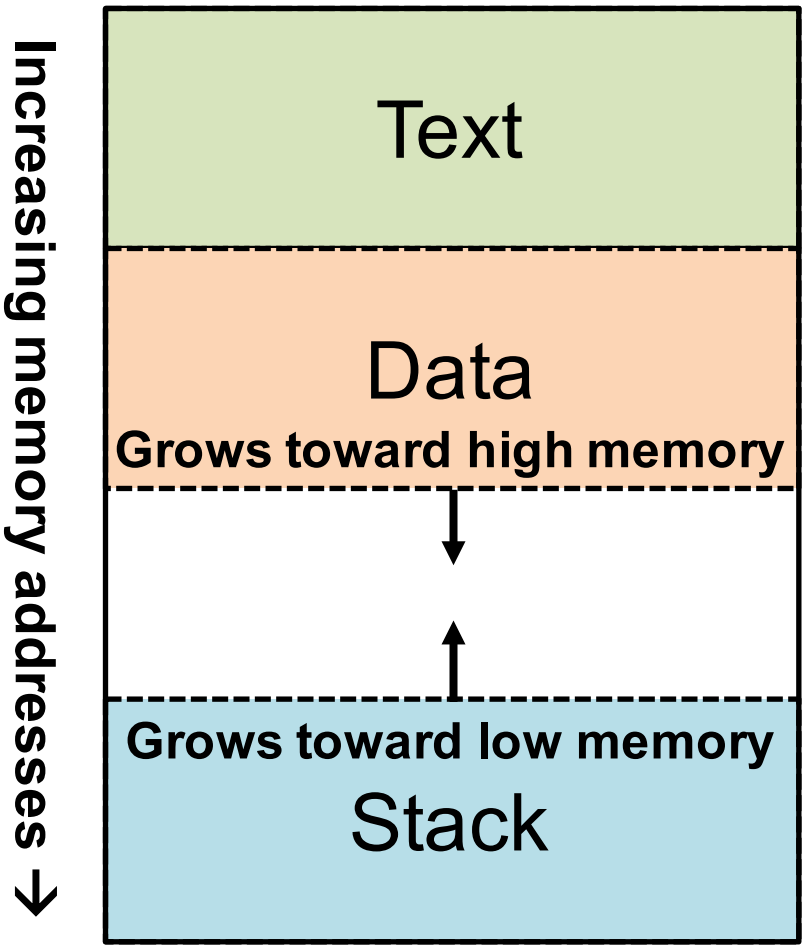
Buffer overflows in C: General idea

- Buffers (arrays) in C manipulated using pointers
- C allows arbitrary arithmetic on pointers
  - Compiler has no notion of size of object pointed to
  - Programmers must explicitly check in code that pointer remains within intended object
  - But programmers often do not do so; **vulnerability!**

- Buffer overflows used in many exploits:
  - Input long data that runs past end of programmer's buffer, over memory that guides program control flow
  - Enclose code you want executed within data
  - Overwrite control flow info with address of your code!

Memory map of a UNIX process

- **Text:** executable instructions, read-only data; size fixed at compile time
- **Data:** initialized and uninitialized; grows towards higher addresses
- **Stack:** Holds function arguments and local variables; grows toward lower addresses

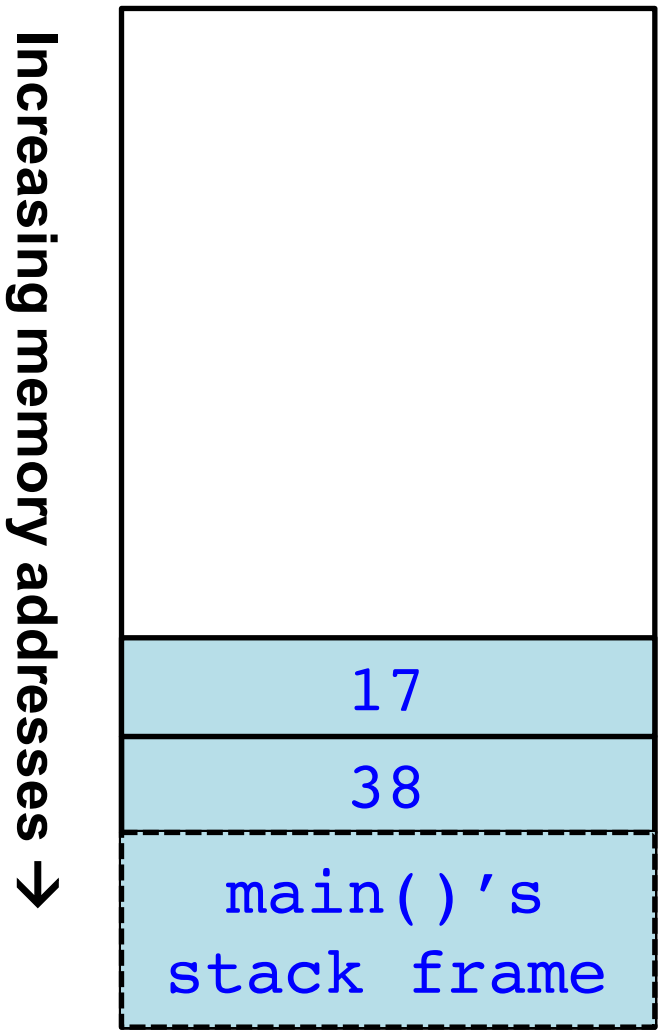


Intel X86 Stack: Stack Frames

- Region of stack used within C function: ***stack frame***
- Within function, local variables allocated on stack
- ***SP*** register: stack pointer, points to top of stack
- ***BP*** register: frame pointer (aka base pointer), points to bottom of stack frame of currently executing function

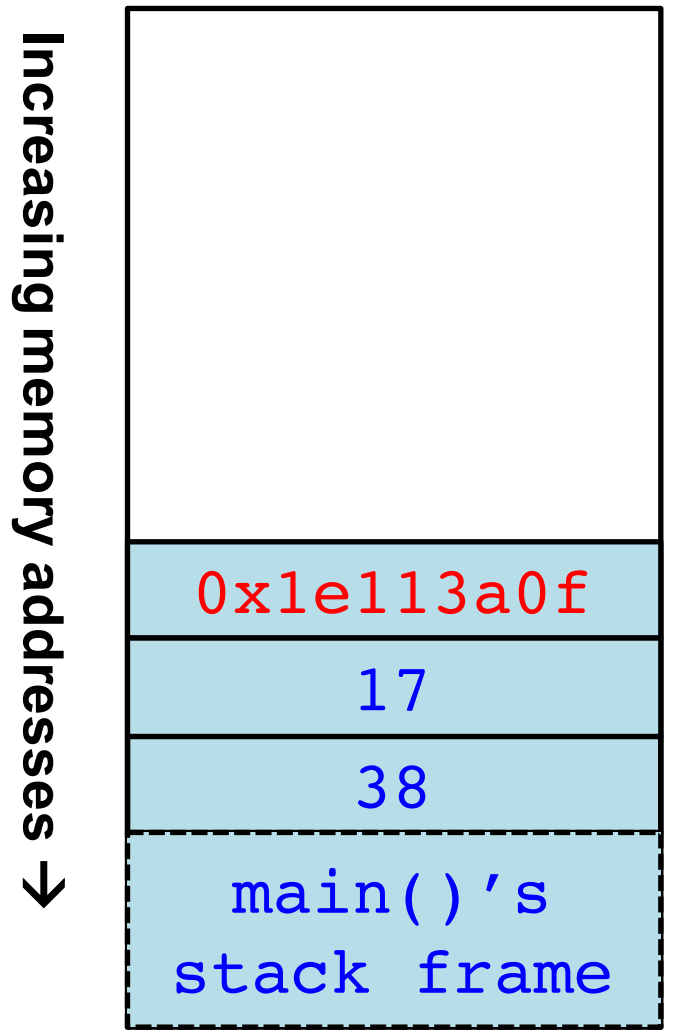
Intel x86 stack: Function call

```
void f(int a, int b) {
   char request[256];
   scanf("%s", request);
   /* process request.. */
   return;
}

int main(int, char **) {
   while (1) {
      f(17, 38);
      fprintf (log, "done!\n");
   }
}
```

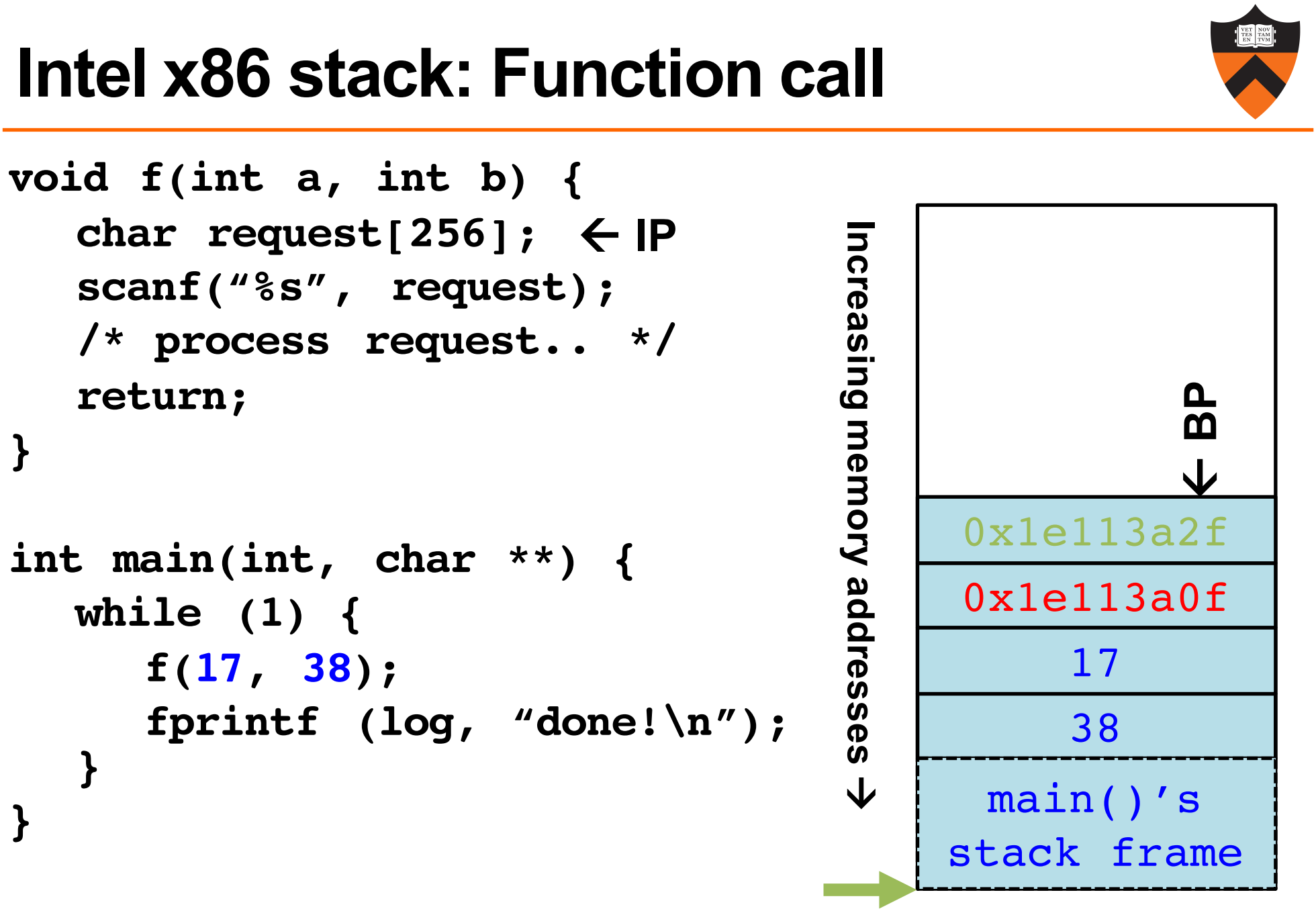


Intel x86 stack: Function call

```
void f(int a, int b) {
   char request[256];
   scanf("%s", request);
   /* process request.. */
   return;
}

int main(int, char **) {
   while (1) {
      f(17, 38);
      fprintf (log, "done!\n");
   }
}
```

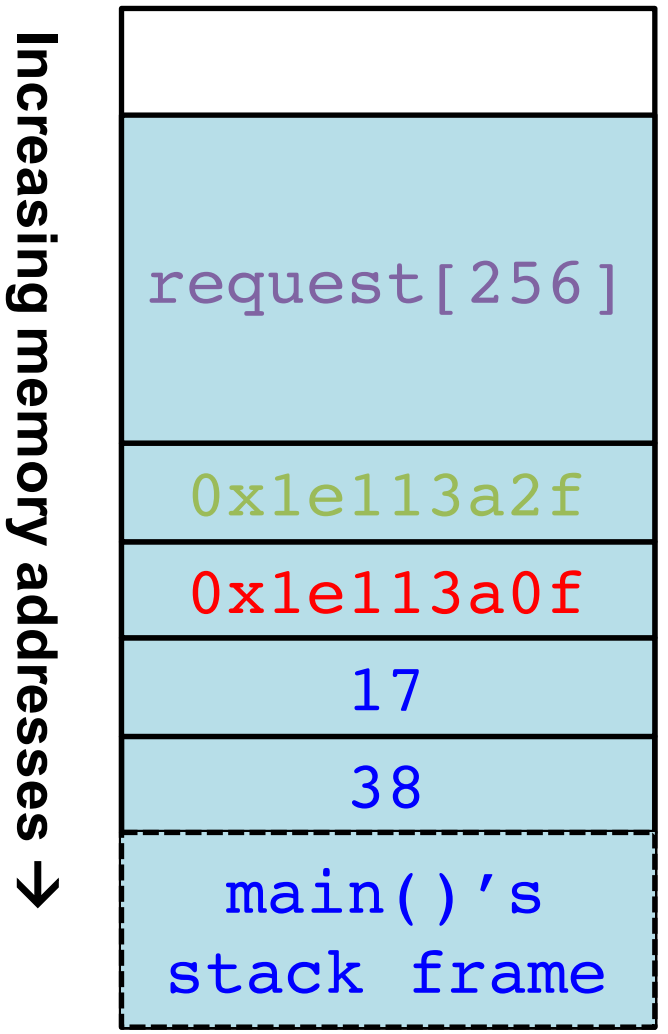


Intel x86 stack: Function call



Intel x86 stack: Function call

```
void f(int a, int b) {
   char request[256];
   scanf("%s", request);
   /* process request.. */
   return;
}

int main(int, char **) {
   while (1) {
      f(17, 38);
      fprintf (log, "done!\n");
   }
}
```

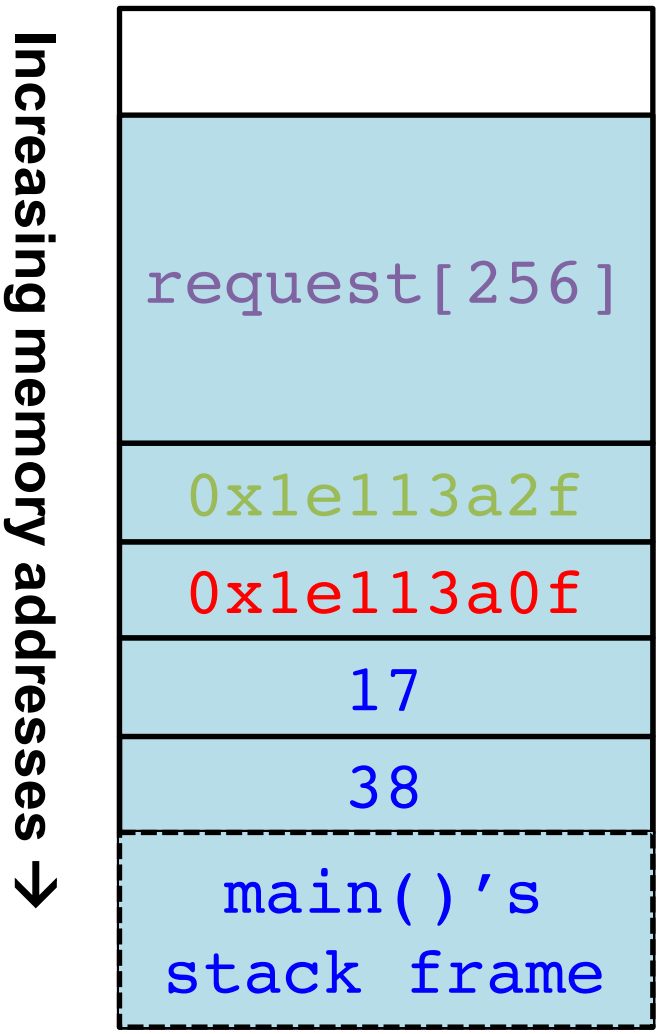


Intel x86 stack: Function return

- Upon return from **`f(int,int)`**

- Deallocate stack frame:
  - SP ← SP + sizeof(locals)
    - deallocates local vars
  - BP ← saved frame pointer from stack
    - change to caller's stack frame

- Return to next instruction in caller
  - Set IP = saved return address from stack



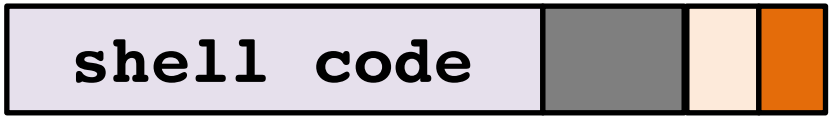
Stack Smashing Exploits: Basic Idea

- Return address stored on stack directly influences program control flow
- Stack frame layout: local variables allocated just before return address
- If programmer allocates buffer as local on stack, reads input, and writes it into buffer without checking input fits in buffer:
  - Send input containing shellcode you wish to run
  - Write past end of buffer, and overwrite return address with address of your code within stack buffer
  - When function returns, your code executes!

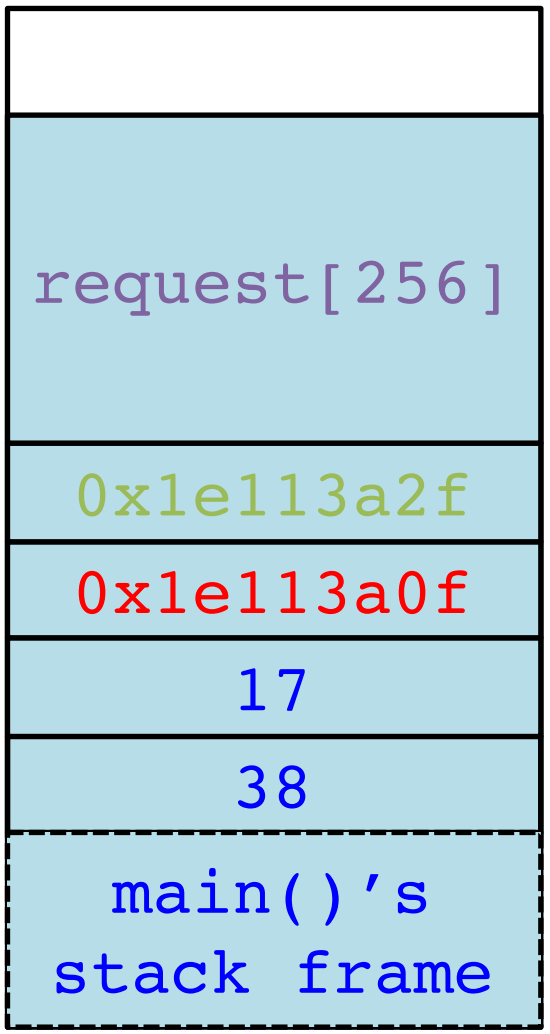
Example: Stack smashing

```
void f(int a, int b) {
   char request[256];
   scanf("%s", request);
   /* process request.. */
   return;
}

int main(int, char **) {
   while (1) {
      f(17, 38);
      fprintf (log, "done!\n");
   }
}
```

Malicious input:



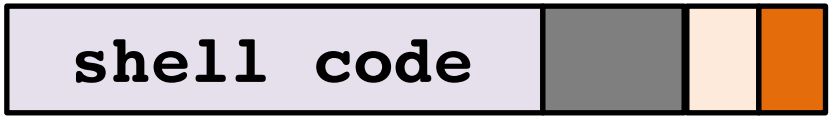
Increasing memory addresses →



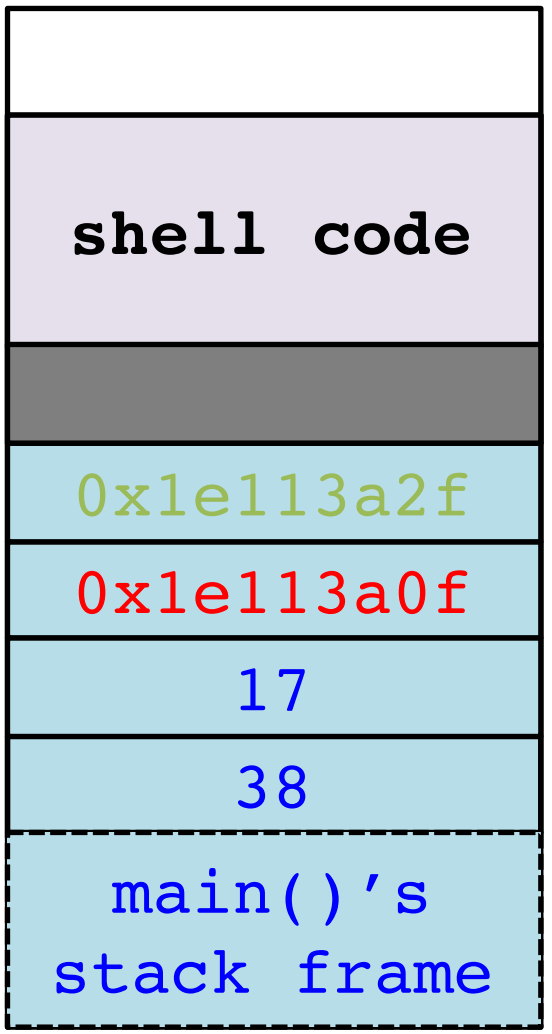
Example: Stack smashing

```
void f(int a, int b) {
   char request[256];
   scanf("%s", request);
   /* process request.. */
   return;
}

int main(int, char **) {
   while (1) {
      f(17, 38);
      fprintf (log, "done!\n");
   }
}
```

Malicious input:



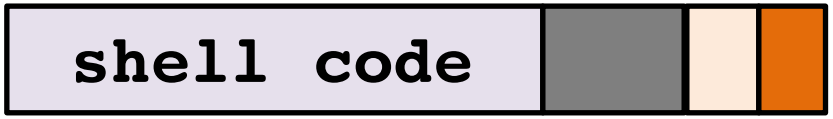
Increasing memory addresses →



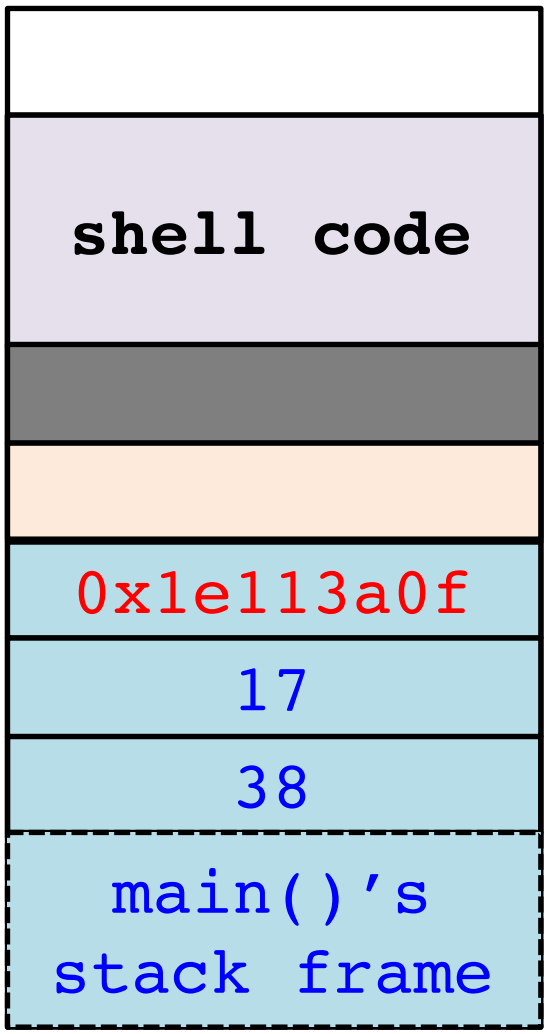
Example: Stack smashing

```
void f(int a, int b) {
   char request[256];
   scanf("%s", request);
   /* process request.. */
   return;
}

int main(int, char **) {
   while (1) {
      f(17, 38);
      fprintf (log, "done!\n");
   }
}
```

Malicious input:



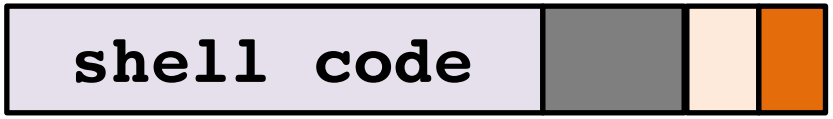
Increasing memory addresses →

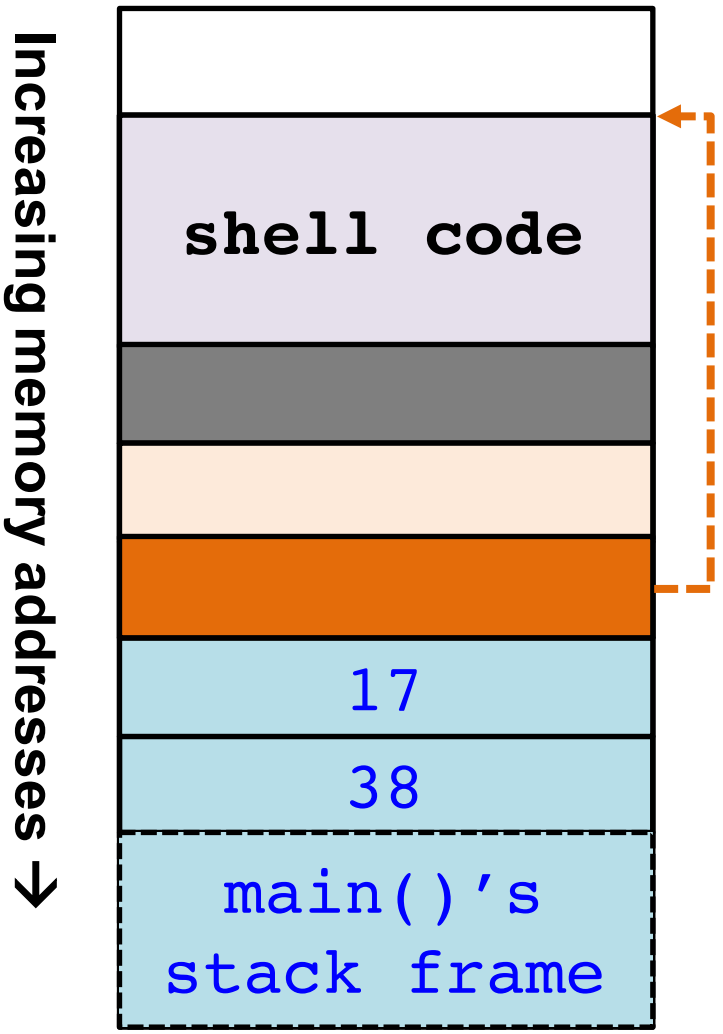


Example: Stack smashing

```
void f(int a, int b) {
   char request[256];
   scanf("%s", request);
   /* process request.. */
   return;
}

int main(int, char **) {
   while (1) {
      f(17, 38);
      fprintf (log, "done!\n");
   }
}
```

Malicious input:





Designing a stack smashing exploit

- In our example, attacker had to know
  - Existence of stack-allocated buffer without bounds check in program
  - **exact address** for start of stack-allocated buffer
  - **exact offset** of return address beyond buffer start

- Hard to predict these exact values
  - Stack size before call to function containing vulnerability may vary, changing exact buffer address
  - attacker may not know exact buffer size

- **Don't need to know either exact value, though!**

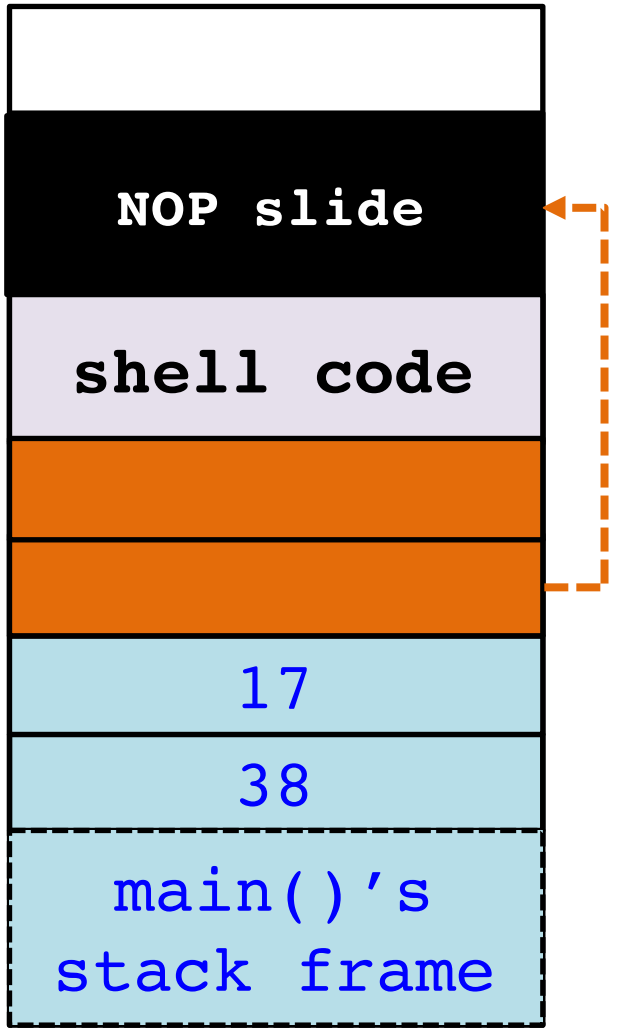
Designing a stack smashing exploit

- **No need** to know **exact return address**
  - Precede shellcode with NOP slide: long sequence of NOPs (or equivalent instructions)
  - So long as jump into NOP slide, shellcode executes
  - Effect: range of return addresses works

- **No need** to know **exact offset of return address** beyond buffer start:
  - Repeat shellcode's address many times in input
    - If first repetition occurs before return address's location on stack, and enough reps, will overwrite it

Example: Stack smashing "v2.0"

```
void f(int a, int b) {
   char request[256];
   scanf("%s", request);
   /* process request.. */
   return;
}

int main(int, char **) {
   while (1) {
      f(17, 38);
      fprintf (log, "done!\n");
   }
}
```



Malicious input:

Defensive coding to avoid buffer overflows

- Always explicitly check input length against target buffer size
- Avoid C library calls that don't do length checking:
  - *e.g.*, **`sprintf(buf, ...), scanf("%s", buf), strcpy(buf, input)`**

- Better:
  - **`snprintf(buf, buflen, ...), scanf("%256s", buf), strncpy(buf, input, 256)`**

Recap

- The perfect programmer would check bounds 100% of the time, but it turns out no programmers are perfect.

- So we need **defenses** that make make buffer overflows harder to exploit, for big buggy programs

W ⊕ X page protection

- Recall from operating systems class: CPU implements page protection in hardware
  - For each 4K memory page, permission bits specified in page table entry in kernel: read, write

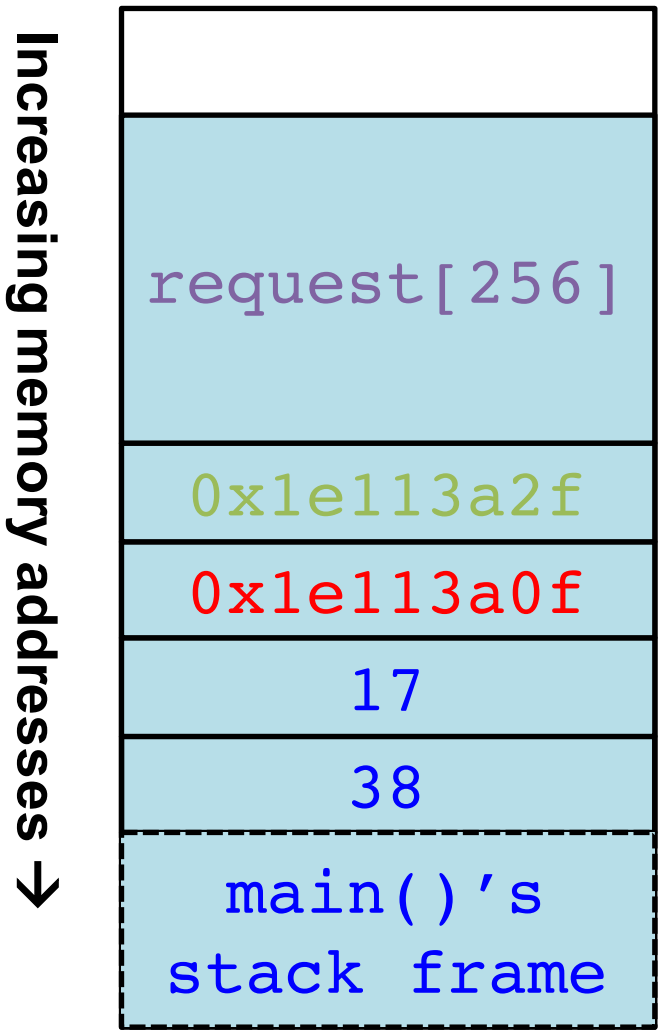
- Central problem in many exploits:
  - Code supplied by user in input data
  - Execution transferred to user's input data

- **Idea:** don't let CPU execute instructions in data pages
  - *i.e.,* each page should **either be writable or executable, but not both**
  - Text pages: X, not W; stack and heap pages: W, not X

W ⊕ X hole: "Return-to-libc" attack

- Instead of putting shellcode on the stack:
  - Just put arguments there
    - Data, so okay
  - Overwrite ret address to a well-known library function

- *e.g.*, `system("/bin/sh")`

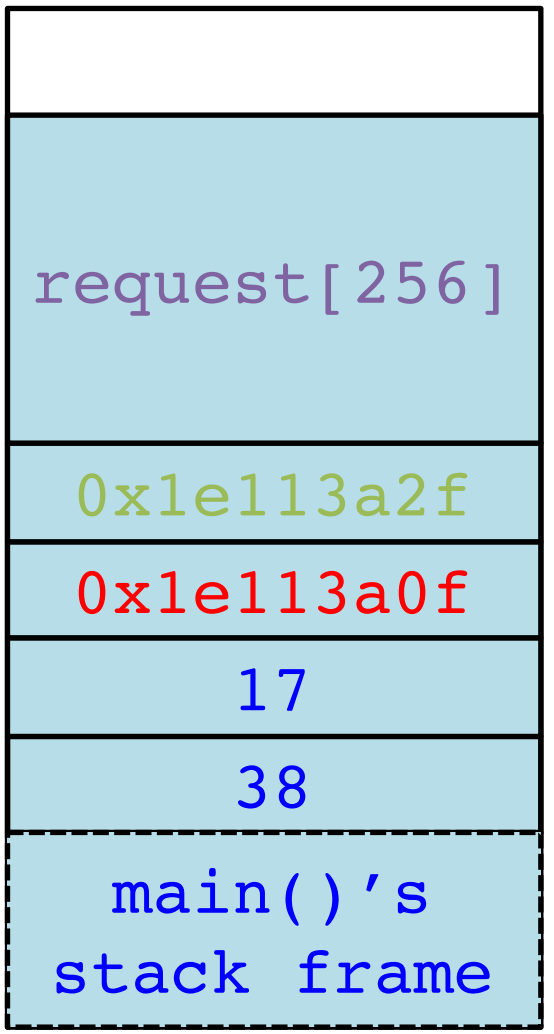


W ⊕ X hole: "Return-to-libc" attack

```
void f(int a, int b) {
   char request[256];
   scanf("%s", request);
   /* process request.. */
   return;
}

int main(int, char **) {
   while (1) {
      f(17, 38);
      fprintf (log, "done!\n");
   }
}
```

Malicious input:

Increasing memory addresses →

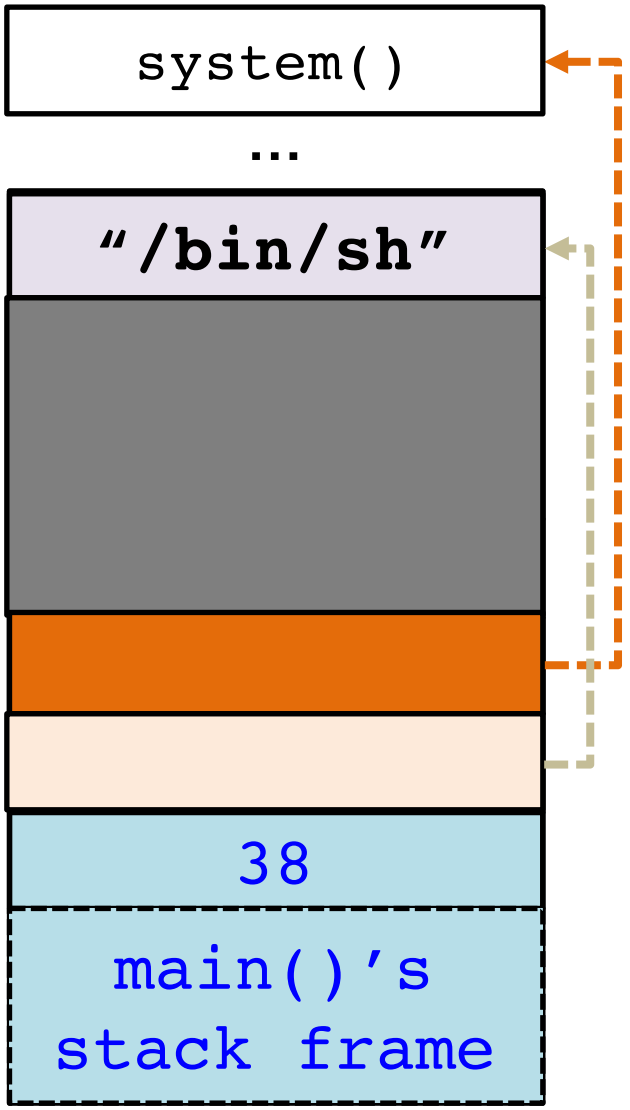


W ⊕ X hole: "Return-to-libc" attack

```
void f(int a, int b) {
   char request[256];
   scanf("%s", request);
   /* process request.. */
   return;
}

int main(int, char **) {
   while (1) {
      f(17, 38);
      fprintf (log, "done!\n");
   }
}
```

Malicious input:

Increasing memory addresses →



Address Space Layout Randomization (ASLR)

- Central observation: attacker must predict addresses
  - *e.g.*, shellcode buffer address, libc function address, string argument address

- **Idea:** randomize addresses in process
  - With high probability, attacker will guess wrong
  - Jump to unmapped memory: crash
  - Jump to invalid instruction stream: crash

- Useful as efficient exploit detector
  - Memory faults or illegal instructions suggest exploit

ASLR Implementation: PaX for Linux

- Linux process contains three memory regions:
  - Executable: text, init data, uninit data
  - Mapped: heap, dynamic (shared) libraries, thread stacks, shared memory
  - Stack: user stack

- ASLR adds random offset to each area when process created
  - Efficient; easily supported by virtual memory hardware
  - 16, 16, 24 bits randomness, respectively

- Mapped offset limited to 16 bits
  - bits 28-31 cannot be changed; would interfere with big mmap()s
  - bits 0-11 cannot be randomized; would make mmap()ed pages not be page-aligned

Derandomization Attack on ASLR [Shacham, Boneh et al.]

- 16 bits not that big; try to guess random offset added to mapped area

- Once know random offset, can predict addresses of shared libraries
  - thus libc function addresses
  - ...so can mount return-to-libc attack

- Two phases:
  - brute-force random offset to mapped area
  - compute “derandomized” address of syscall(), use in return-to-libc attack

Today

1. Introduction to the design of secure systems

2. Stack smashing buffer overflows and countermeasures

3. **Heap smashing buffer overflows and countermeasures**

Heap attacks

- Are overflows of heap-allocated buffers exploitable?
  - Modern code tends to use the heap a lot

```
foo() {
   char *p = malloc(16);
   gets(p);
}
```

- Can attacker predict what's after `p` in memory?

`malloc`: A rough primer

- Some `malloc()`s lay out blocks of free/used memory as:

```
prev
next
data
----
prev
next
data
----
prev
next
data
```

- malloc keeps free blocks on a doubly-linked list
- When a free block is allocated, here's part of what `malloc()` does:

```
b = choose a free block
b->next->prev = b->prev;
b->prev->next = b->next;
```

Heap smashing

- If the attacker overflows a `malloc()`ed block, the attacker can modify the next and prev pointers in the next block:

- Suppose attacker writes `x y` to start of next block
  - Call next block `b`
  - then `b->prev` = `x`, `b->next` = `y`
  - Suppose `b` free and happens to be chosen by next `malloc()`
    - `malloc()` will effectively execute `*y = x`
  - Thus writing an attacker-chosen value to any memory location!

Heap smashing: Limitations

- But, note the pieces the attacker must assemble:

1. Find a buffer overflow bug in the application or library

2. Find a way to get the program to execute the buggy code in a way that causes attacker's bytes to overflow the buffer

3. Understand `malloc()` implementation.

4. Find a code pointer and guess its address.

5. Guess the address of the buffer, *i.e.* attacker's injected instructions.

Other countermeasures

- **Stack Canaries** (*e.g.,* StackGuard, gcc stack protection)
  - Detects modification of return PC on stack before used
  - Compiler generates code that pushes "canary" value on stack at function entry, pops and checks value before ret
    - Canary sits between variables and ret address
  - But what if attacker can read/write canary?

- **Taint Checking**
  - Many exploits use data supplied by user to subvert control flow of program
  - Mark all data from user (received from network, or from input files) as tainted and propagate taint during execution
  - Drawback: 25× slowdown

Coming up

- **Happy Thanksgiving!**
- **Mon 11/30** Guest Lecture (Tor developer Philipp Winter)
- Then: Traffic analysis and censorship resistance in Tor
  - Untrusted Cloud Infrastructure
  - Deniable/Stealthy Communication

		Project Presentations
Fri 12/11 2:00 PM– 4:00PM	**Project presentations** **Room: J201 (tentative; TBC)**	Paraskevopoulou/Giannarakis; Nagree/Leef; Apthorpe/Grover; Sotirios/Mohajeri
Mon 12/14	**Project presentations**	**Deadline** for project presentation and version 0 demo. Miltner/Yang/L. Zhang; Li/Jackson/Turner; Garg/Singh/Schvartzman; Fuchs/Shahrad/Zhou
Tue 12/15 10:00 AM– 12:00 AM	**Project presentations** **Non-standard room: CS 302**	Bullins/Agarwal/Arashloo; Zung/Kathpalia/Ouyang; Suo/Zeng/Y. Zhang/Simmons-Edler
Wed 12/16	**No class.**	
Tue 1/12/2016 (Dean's Date)	No class.	**Deadline** for five-page project writeup.