

# IT 4823

## Information Security Administration

### Network Configuration Buffer Size Attacks



Notice: This session is  
being recorded.

Some lecture slides prepared by Dr Lawrie Brown for “*Computer Security: Principles and Practice*”, 1/e, by William Stallings and Lawrie Brown.

# Configuring a Network

- Static Configuration
  - IP address
  - Network mask
  - DNS server(s)
  - Default gateway
- Dynamic Host Configuration Protocol (DHCP)
  - A computer sends a broadcast asking for configuration information.
  - A DHCP server replies

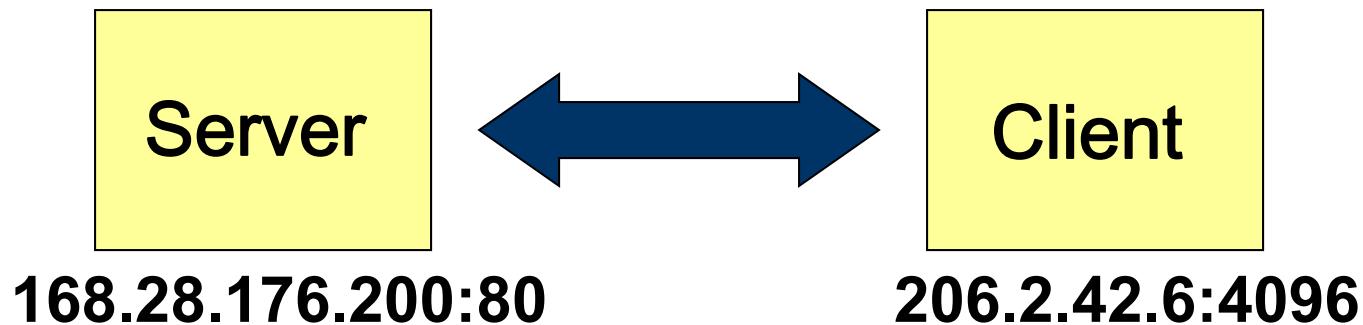
# RFC-1918 Reserved Addresses

- Every computer attached directly to the Internet must have a unique IP address.
- Not every computer that uses the IP protocol must be directly connected to the Internet
- Reserved addresses
  - 10.0.0.0 – 10.255.255.255
  - 172.16.0.0 – 172.31.255.255
  - 192.168.0.0 – 192.168.255.255
- These same address ranges may be used by many organizations; they are dropped by Internet routers; sometimes called “non-routable.”

# Network Address Translation

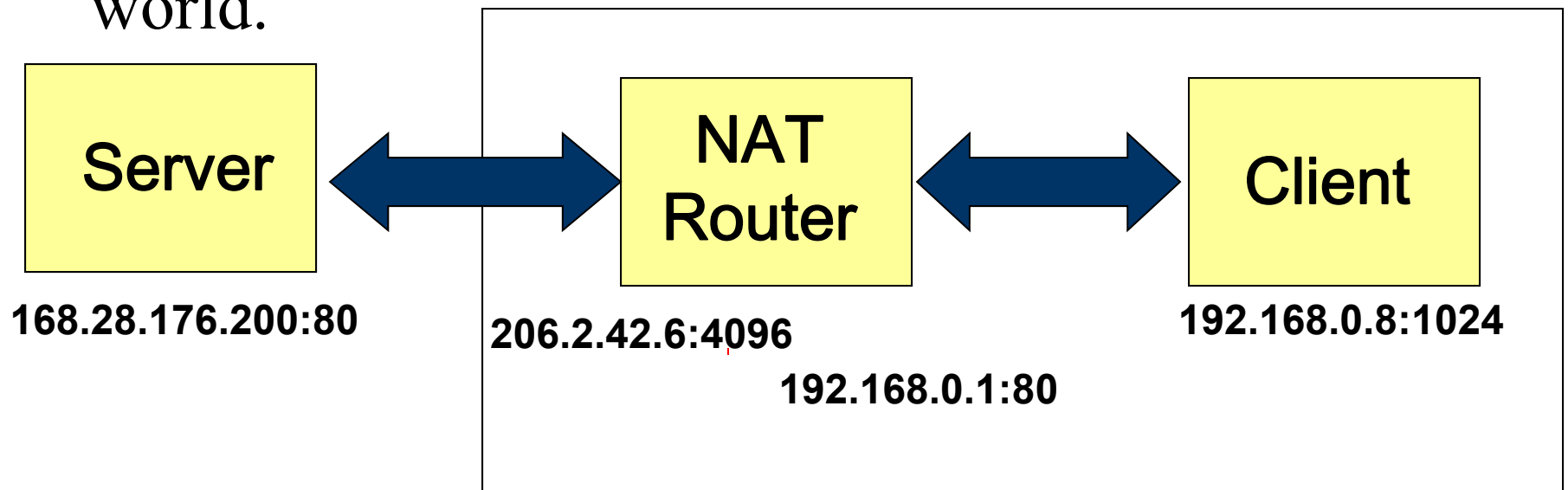
All IP connections are characterized by four numbers:

- Address of source
- Port of source
- Address of destination
- Port of destination



# Network Address Translation

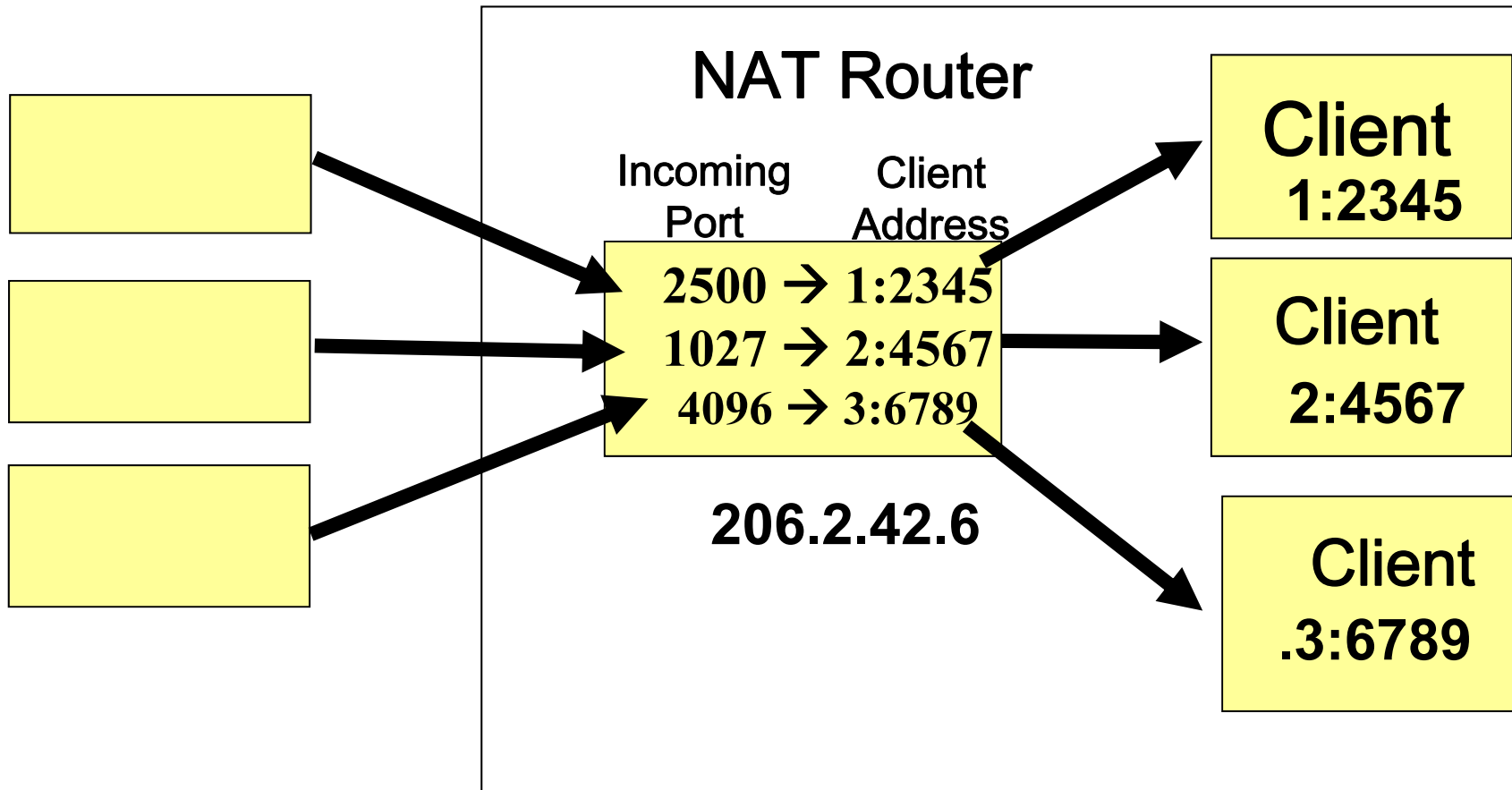
- Network address translation (NAT) interposes an “appliance” between a computer and the “outside world.”



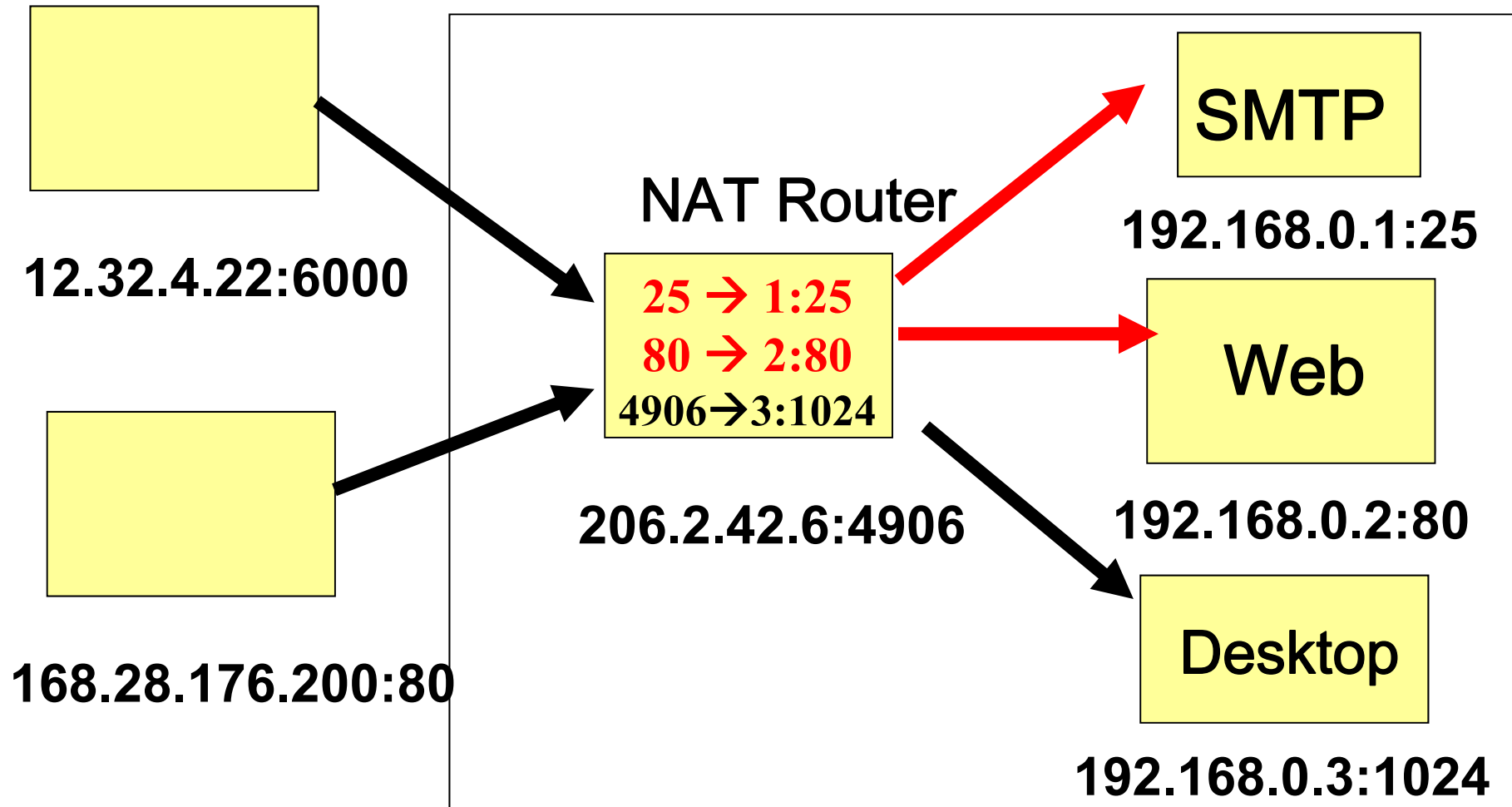
# Network Address Translation

- Advantages:
  - Internet activity using reserved addresses
  - NAT'd computers are defended from unsolicited incoming packets.
- Disadvantages
  - Running Internet servers behind a NAT router takes special configuration.
  - No protection against “solicited” malicious data, *e.g.* email, malicious Web sites.

# The NAT Table is Dynamic



# Port Forwarding





# Buffer Overflow

- A very common attack mechanism
  - from 1988 Morris Worm to Code Red, Slammer, Sasser and many others
- Prevention techniques are well known
- Still of major concern due to
  - legacy of widely deployed buggy software
  - continued careless programming techniques

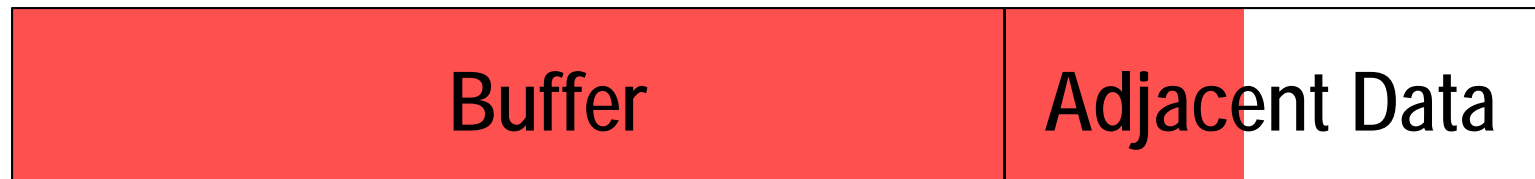
# Buffer Overflow Basics

- Enabled by programming error
- Allows more data to be stored than capacity available in a fixed sized buffer...
- Overwriting adjacent memory locations:
  - corruption of program data
  - unexpected transfer of control
  - memory access violation
  - execution of code chosen by attacker
- Buffer can be on stack, heap, global data

# Buffer Overflow Attacks

To exploit a buffer overflow an attacker must

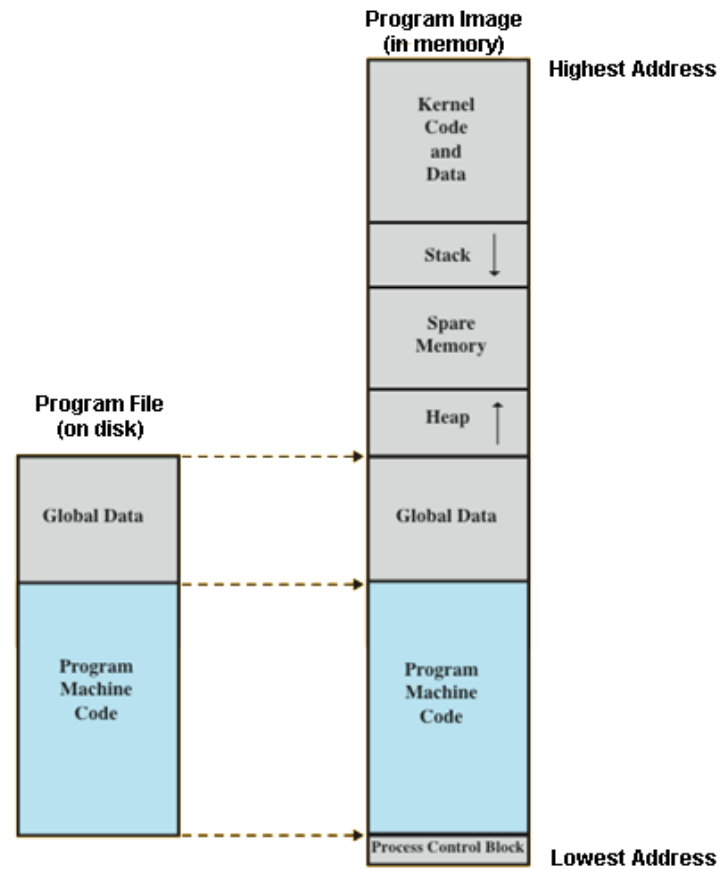
- Identify a buffer overflow vulnerability in some program
- Understand how buffer is stored in memory and determine potential for corruption
- Storing more data than will fit corrupts adjacent data or code.



# A Little Programming Language Theory

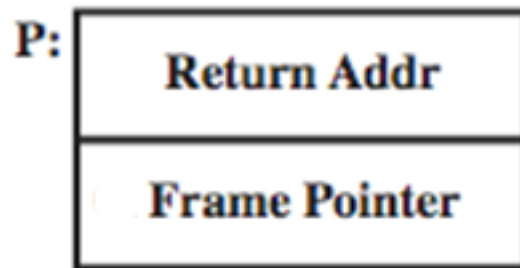
- At the machine level, everything is an array of bytes
  - Interpretation depends on instructions used, *i.e.* context
- Modern high-level languages (like Java) have a strong notion of type and valid operations
  - Not vulnerable to buffer overflows
  - But does impose overhead, some limits on use
- C and related languages have high-level control structures, but allow direct access to memory
  - Are vulnerable to buffer overflow
  - Have a large legacy of widely used, unsafe, and hence vulnerable code

# Program Loaded into Memory

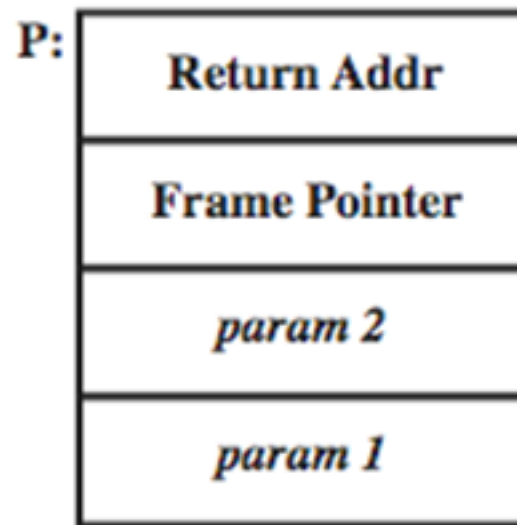


# Using the Stack: P Calls Q

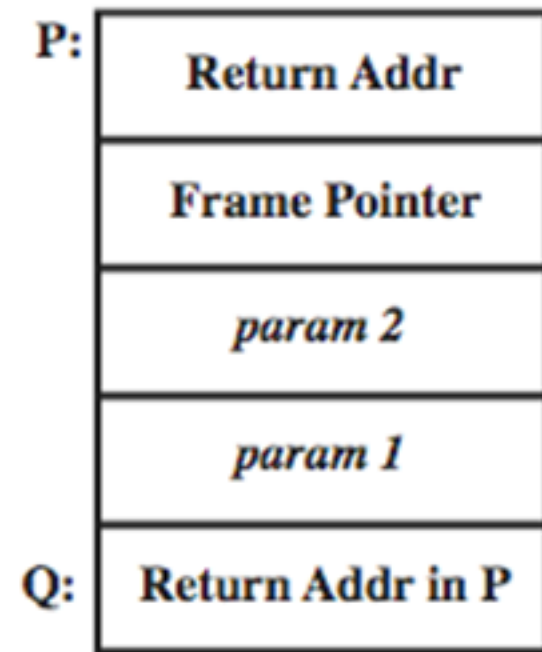
(Stack grows downward.)



Before call



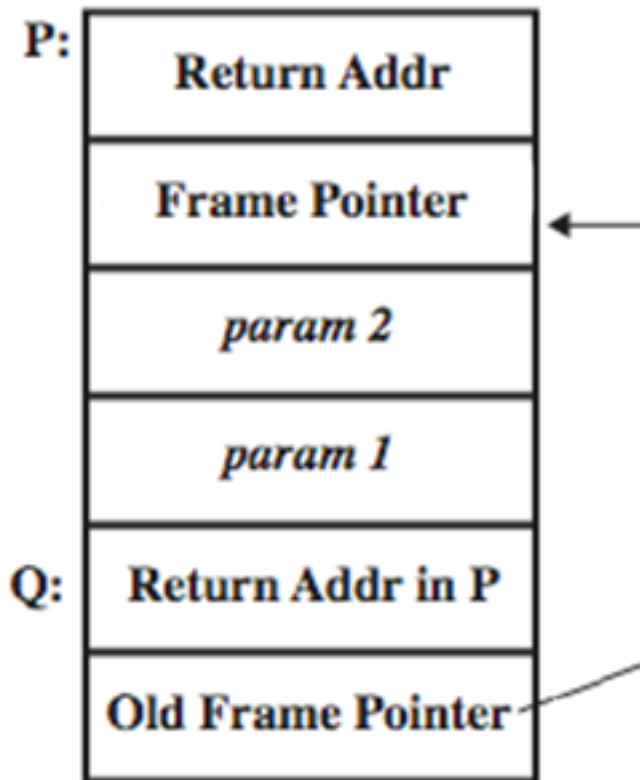
P pushes params



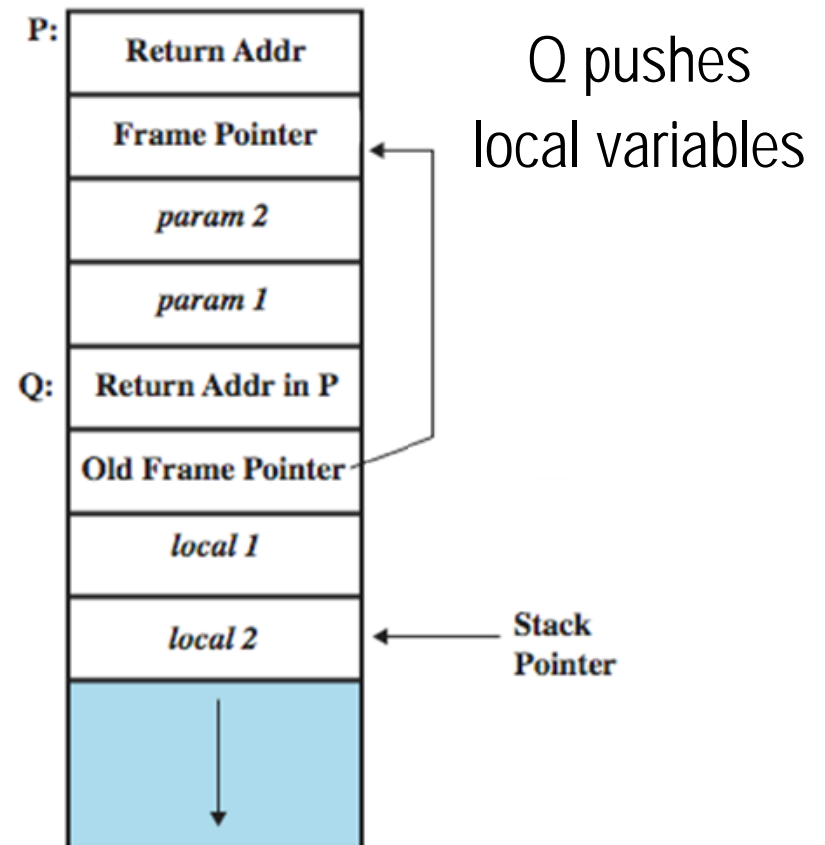
Call to Q  
OS pushes P's  
program counter

# Using the Stack: P Calls Q

(Stack grows downward.)



Q saves P's frame pointer

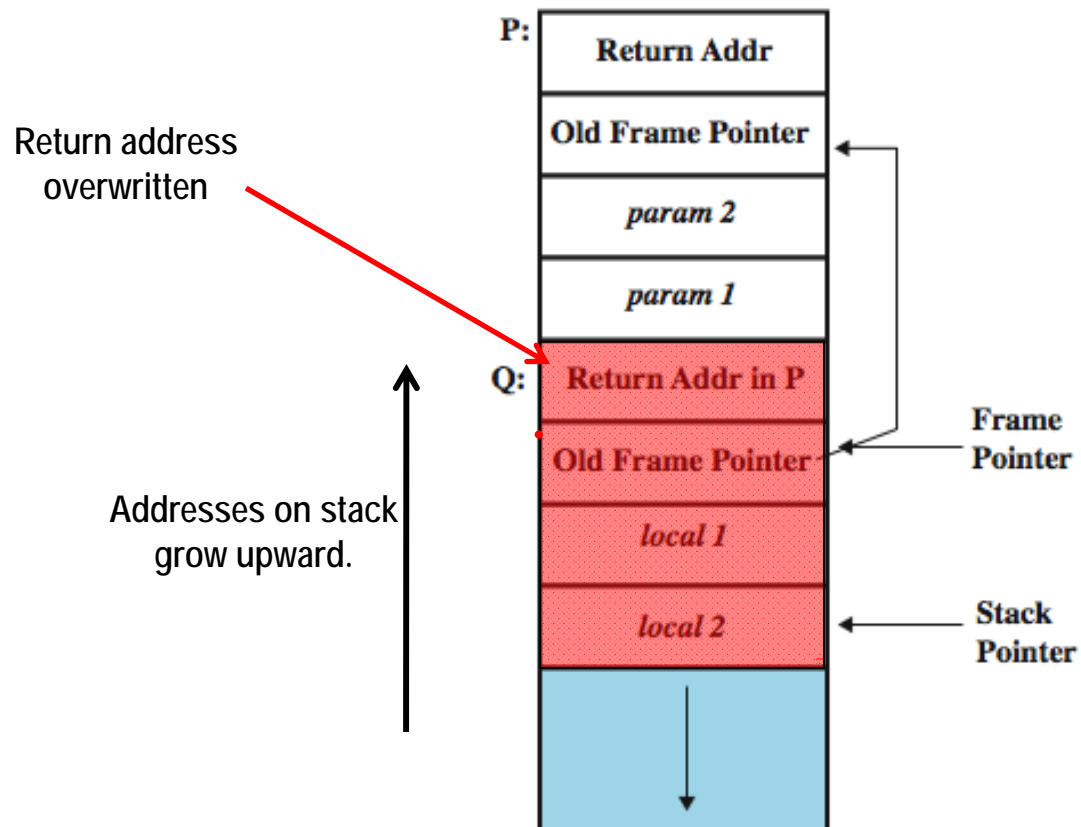


# Stack Buffer Overflow

- Can occur when buffer is located on stack
  - used by Morris Worm
  - “Smashing the Stack” paper popularized it
- And the stack has local variables below saved frame pointer and return address
  - Hence overflow of a local buffer can potentially overwrite these key control items
- Attacker overwrites return address with address of desired code
  - Can be program, system library or loaded in buffer



# Attacker Overflows Buffer in *local2*



# Shellcode

- Code supplied by attacker
  - often saved in buffer being overflowed
  - traditionally transferred control to a shell
- Shell code is **machine code**
  - specific to processor and operating system
  - traditionally needed good assembly language skills to create
  - more recently have automated sites/tools

# Shellcode Development

- Illustrated with classic Intel Linux shellcode to run Bourne shell interpreter
- Shellcode must
  - marshal argument for `execve()` and call it
  - include all code to invoke system function
  - be position-independent
  - not contain NULLs (C string terminator)

# More Stack Overflow Variants

- Target program can be:
  - a trusted system utility
  - network service daemon
  - commonly used library code
- Shellcode functions
  - spawn shell
  - create listener to launch shell on connect
  - create reverse connection to attacker
  - flush firewall rules
  - break out of chroot environment

# Buffer Overflow Defenses

- Buffer overflows are widely exploited
- There is a large amount of vulnerable code in use, despite that cause and countermeasures are known
- Two broad defense approaches
  - compile-time – harden new programs
  - run-time – handle attacks on existing programs

# Compile-Time Defenses: Programming Language

- Use a modern high-level languages with strong typing:
  - not vulnerable to buffer overflow
  - compiler enforces range checks and permissible operations on variables
- Strong typing at runtime does have cost in resource use...
- And restrictions on access to hardware
- So, we still need some code in C like languages

# Compile-Time Defenses: Safe Coding Techniques

- If using potentially unsafe languages *e.g.* C
- Programmer must explicitly write safe code
  - by design with new code
  - after code review of existing code, *c.f.* OpenBSD
- Buffer overflow safety is a subset of general safe coding techniques (Ch 12)
  - allow for graceful failure
  - checking for sufficient space in any buffer

# Compile-Time Defenses: Language Extension, Safe Libraries

- There are proposals for safety extensions to C
  - performance penalties
  - must compile programs with special compiler
- There are several safer standard library variants
  - new functions, *e.g.* **strncpy( )**
  - safer re-implementation of standard functions as a dynamic library, *e.g.* Libsafe



# Compile-Time Defenses: Stack Protection

- Add function entry and exit code to check stack for signs of corruption
- Use a random “canary” – a value that can be checked
  - *e.g.* Stackguard, Win /GS
  - check for overwrite between local variables and saved frame pointer and return address
  - abort program if change found
  - issues: recompilation, debugger support
- Or save/check safe copy of return address
  - *e.g.* Stackshield, RAD (Return Address Defender)

## Run-Time Defenses: Non Executable Address Space

- Use virtual memory support to make some regions of memory non-executable
  - *e.g.* stack, heap, global data
  - need hardware support in MMU
  - long existed on SPARC / Solaris systems
  - recent on x86 Linux/Unix/Windows systems as Data Execution Prevention (DEP)
- Issue: support for executable stack code; need special provisions in that case

# Run-Time Defenses: Address Space Randomization

- Manipulate location of key data structures
  - stack, heap, global data
  - using random shift for each process
  - Because there is a large address range on modern systems, wasting some has negligible impact
- Also randomize location of heap buffers
- And location of standard library functions

# Run-Time Defenses: Guard Pages

- Place guard pages between critical regions of memory
  - flagged in MMU as illegal addresses
  - any access aborts process
- Can even place between stack frames and heap buffers, at execution time and space cost.



# Other Overflow Attacks

- There is a range of other attack variants:
  - stack overflow variants
  - heap overflow
  - global data overflow
  - format string overflow
  - integer overflow
- More are likely to be discovered in future
- Some cannot be prevented except by coding to prevent originally (*i.e.* by not making mistakes in the first place.)

# Replacement Stack Frame

- Stack overflow variant just rewrites buffer and saved frame pointer
  - so return occurs but to dummy frame
  - return of calling function controlled by attacker
- Limitations:
  - must know exact address of buffer
  - calling function executes with dummy frame

# Return to System Call

- Stack overflow variant replaces return address with standard library function call
  - response to non-executable stack defenses
  - attacker constructs suitable parameters on stack above return address
  - function returns and library function executes
    - *e.g.* `system("shell commands")`
  - attacker may need exact buffer address
  - can even chain two library calls

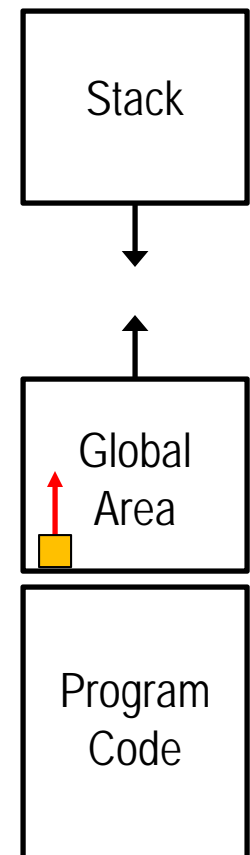
# Heap Overflow

- Attackers can also attack buffers located in heap
  - typically located above program code
  - memory requested by programs to use in dynamic data structures, *e.g.* linked lists
- No return address
  - hence no easy transfer of control
  - may have function pointers that can be exploited
  - or manipulate management data structures
- Defenses: non executable or random heap



# Global Data Overflow

- Attacker can attack buffer located in global data
  - may be located above program code
  - if has function pointer and vulnerable buffer
  - or adjacent process management tables
  - aim to overwrite function pointer later called
- Defenses: non executable or random global data region, move function pointers, guard pages



# Questions

