

OSP2P

131.179.80.139:11111 -- the *normal tracker*. Students' peers can connect to this tracker and serve files to each other. This tracker is "seeded" with a mixture of good and slow peers running on our servers.

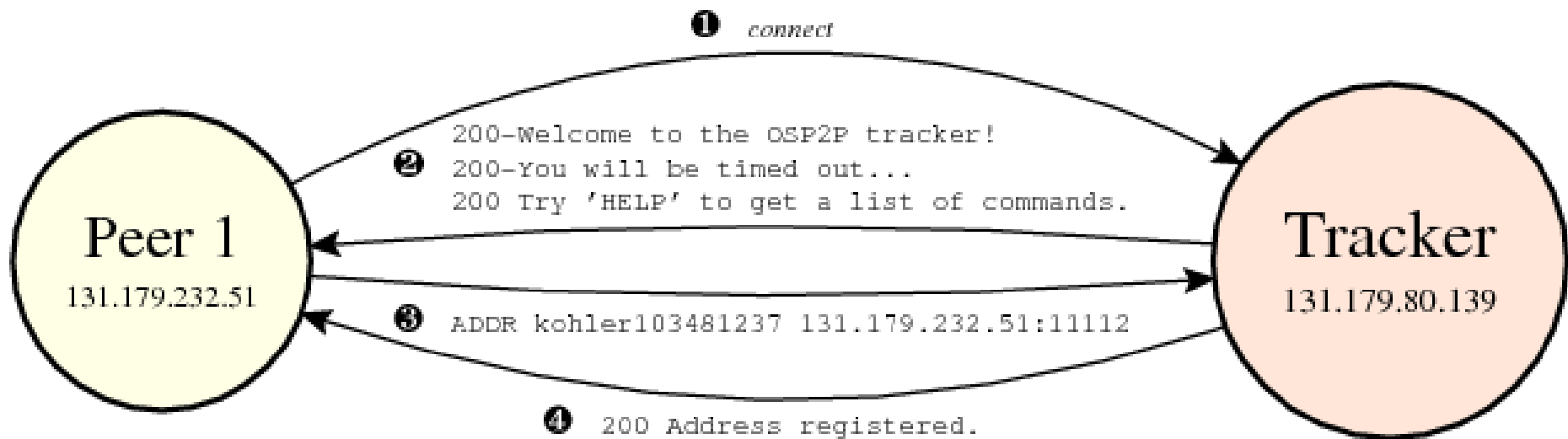
131.179.80.139:11112 -- the *good tracker*. Our own peers are connected to this tracker.

131.179.80.139:11113 -- the *slow tracker*. Slow peers, which serve files slowly, are running on this tracker. You can use this tracker to verify that your peer downloads files in parallel.

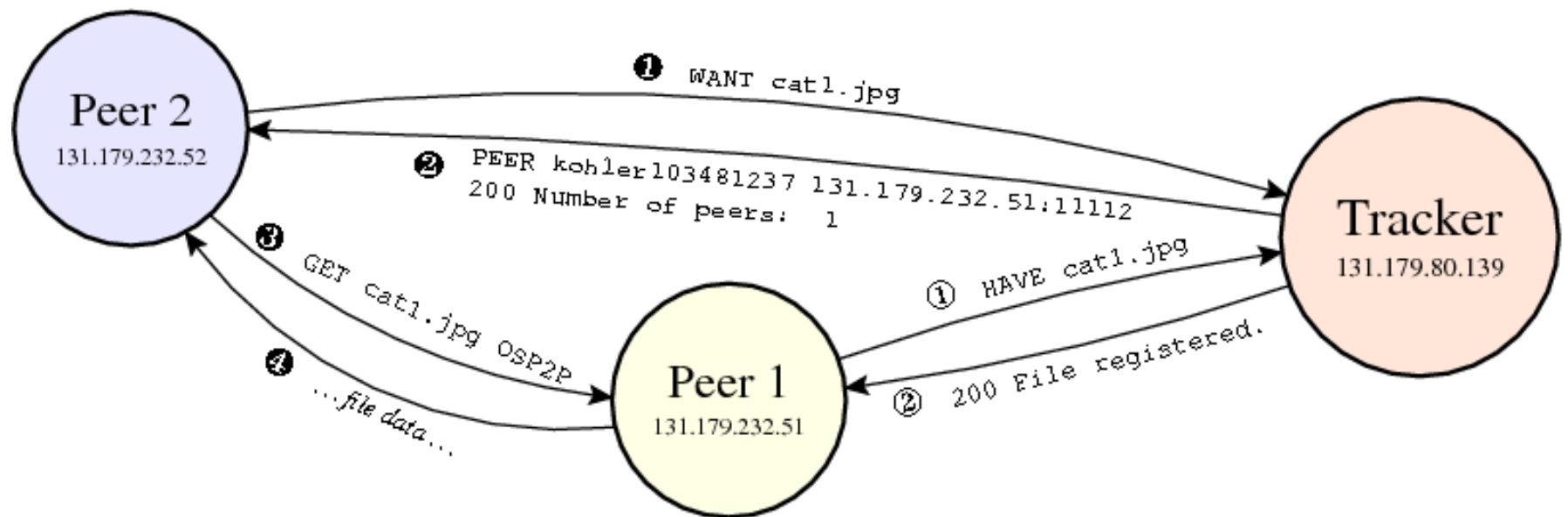
131.179.80.139:11114 -- the *bad tracker*. Bad peers, which attack other peers that attempt to connect, are running on this tracker.

131.179.80.139:11115 -- the *popular tracker*. Our own peers are connected to this tracker, plus a bunch of fake peers, making the tracker look very popular.

OSP2P



OSP2P



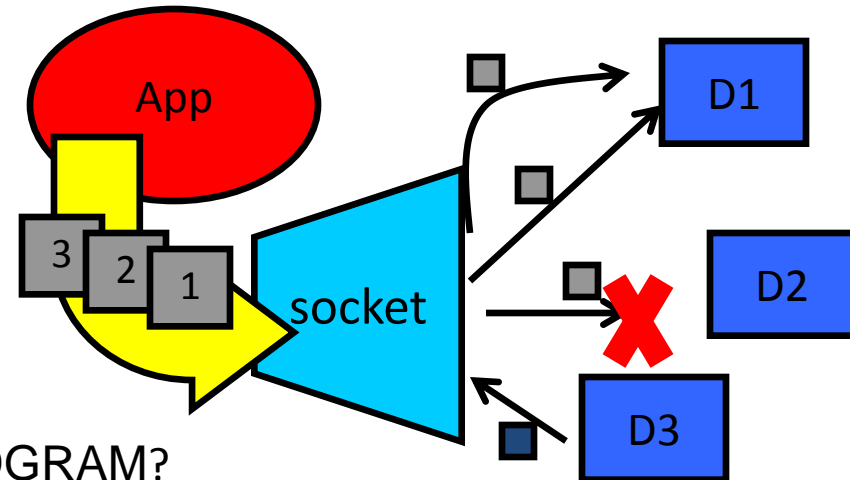
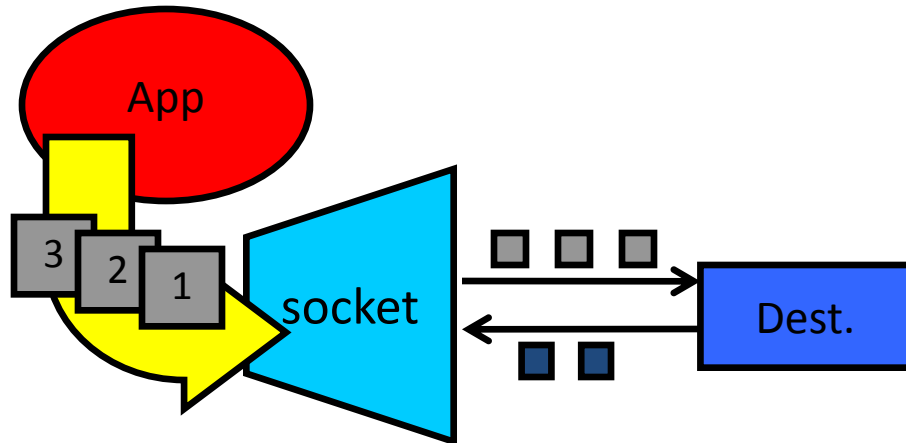
Socket Programming

What is a socket?

- An interface between application and network
 - The application creates a socket
 - The socket *type* dictates the style of communication
 - reliable vs. best effort
 - connection-oriented vs. connectionless
- Once configured the application can
 - pass data to the socket for network transmission
 - receive data from the socket (transmitted through the network by some other host)

Two essential types of sockets

- SOCK_STREAM
 - a.k.a. TCP
 - reliable delivery
 - in-order guaranteed
 - connection-oriented
 - bidirectional
- SOCK_DGRAM
 - a.k.a. UDP
 - unreliable delivery
 - no order guarantees
 - no notion of “connection” – app indicates dest. for each packet
 - can send or receive



Q: why have type SOCK_DGRAM?

Socket Creation in C: `socket`

- `int s = socket(domain, type, protocol);`
 - `s`: socket descriptor, an integer (like a file-handle)
 - `domain`: integer, communication domain
 - e.g., `PF_INET` (IPv4 protocol) – typically used
 - `type`: communication type
 - `SOCK_STREAM`: reliable, 2-way, connection-based service
 - `SOCK_DGRAM`: unreliable, connectionless,
 - `protocol`: specifies protocol (see file `/etc/protocols` for a list of options) - usually set to 0
- NOTE: `socket` call does not specify where data will be coming from, nor where it will be going to – it just creates the interface!

A Socket-eye view of the Internet



vahab.cs.ucla.edu

(128.59.21.14)



newworld.cs.umass.edu

(128.119.245.93)



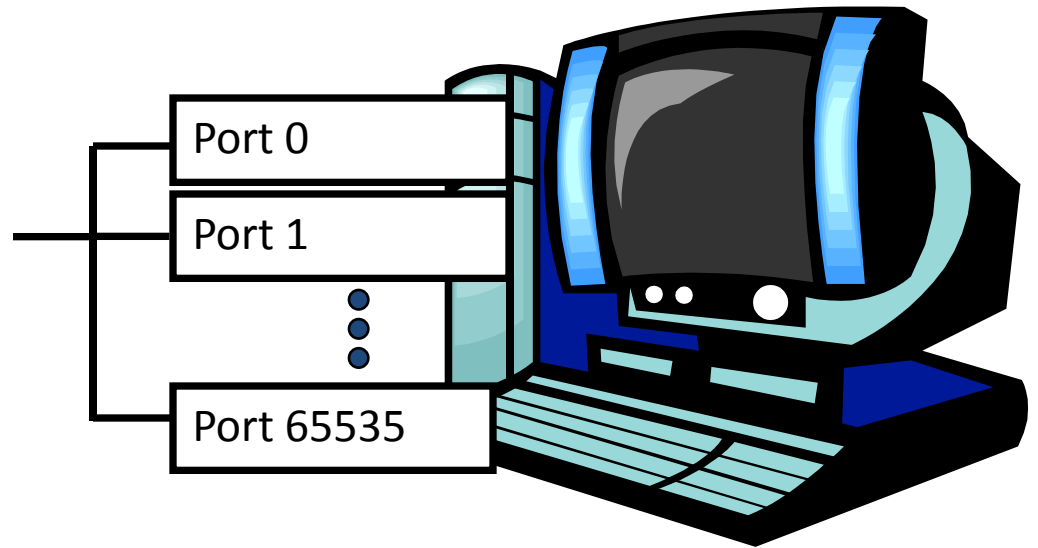
cluster.cs.columbia.edu

(128.59.21.14, 128.59.16.7,
128.59.16.5, 128.59.16.4)

- Each host machine has an IP address
- When a packet arrives at a host

Ports

- Each host has 65,536 ports
- Some ports are *reserved for specific apps*
 - 20,21: FTP
 - 23: Telnet
 - 80: HTTP
 - see RFC 1700 (about 2000 ports are reserved)



□ A socket provides an interface to send data to/from the network through a port

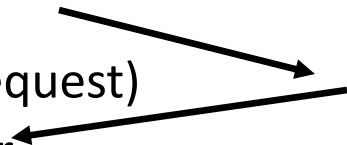
The bind function

- associates and (can exclusively) reserves a port for use by the socket
- `int status = bind(sockid, &addrport, size);`
 - `status`: error status, = -1 if bind failed
 - `sockid`: integer, socket descriptor
 - `addrport`: struct `sockaddr`, the (IP) address and port of the machine (address usually set to `INADDR_ANY` – chooses a local address)
 - `size`: the size (in bytes) of the `addrport` structure

Connection Setup (SOCK_STREAM)

- Recall: no connection setup for SOCK_DGRAM
- A connection occurs between two kinds of participants
 - passive: waits for an active participant to request connection
 - active: initiates connection request to passive side
- Once connection is established, passive and active participants are “similar”
 - both can send & receive data
 - either can terminate the connection

Connection setup cont'd

- Passive participant
 - step 1: **listen** (for incoming requests)
 - step 3: **accept** (a request)
 - step 4: data transfer
 - Active participant
 - step 2: request & establish **connection**
 - step 4: data transfer
- 
- The diagram consists of two arrows. The first arrow originates from the text 'step 1: listen' under the 'Passive participant' and points to the text 'step 2: request & establish connection' under the 'Active participant'. The second arrow originates from the text 'step 2: request & establish connection' and points to the text 'step 3: accept' under the 'Passive participant'.

Connection setup: listen & accept

- Called by passive participant
- `int status = listen(sock, queuelen);`
 - `status`: 0 if listening, -1 if error
 - `sock`: integer, socket descriptor
 - `queuelen`: integer, # of active participants that can “wait” for a connection
 - `listen` is **non-blocking**: returns immediately
- `int s = accept(sock, &name, &namelen);`
 - `s`: integer, the new socket (used for data-transfer)
 - `sock`: integer, the orig. socket (being listened on)
 - `name`: struct `sockaddr`, address of the active participant
 - `namelen`: `sizeof(name)`: value/result parameter
 - must be set appropriately before call
 - adjusted by OS upon return
 - `accept` is **blocking**: waits for connection before returning

connect call

- `int status = connect(sock, &name, namelen);`
 - `status`: 0 if successful connect, -1 otherwise
 - `sock`: integer, socket to be used in connection
 - `name`: struct `sockaddr`: address of passive participant
 - `namelen`: integer, `sizeof(name)`
- connect is **blocking**

Sending / Receiving Data

- With a connection (SOCK_STREAM):
 - `int count = send(sock, &buf, len, flags);`
 - `count`: # bytes transmitted (-1 if error)
 - `buf`: char[], buffer to be transmitted
 - `len`: integer, length of buffer (in bytes) to transmit
 - `flags`: integer, special options, usually just 0
 - `int count = recv(sock, &buf, len, flags);`
 - `count`: # bytes received (-1 if error)
 - `buf`: void[], stores received bytes
 - `len`: # bytes received
 - `flags`: integer, special options, usually just 0
 - Calls are **blocking** [returns only after data is sent (to socket buf) / received]

Sending / Receiving Data (cont'd)

- Without a connection (SOCK_DGRAM):
 - `int count = sendto(sock, &buf, len, flags, &addr, addrlen);`
 - `count, sock, buf, len, flags`: same as `send`
 - `addr`: struct `sockaddr`, address of the destination
 - `addrlen`: `sizeof(addr)`
 - `int count = recvfrom(sock, &buf, len, flags, &addr, &addrlen);`
 - `count, sock, buf, len, flags`: same as `recv`
 - `name`: struct `sockaddr`, address of the source
 - `namelen`: `sizeof(name)`: value/result parameter
- Calls are **blocking** [returns only after data is sent (to socket buf) / received]

close

- When finished using a socket, the socket should be closed:
- `status = close(s);`
 - status: 0 if successful, -1 if error
 - s: the file descriptor (socket being closed)
- Closing a socket
 - closes a connection (for `SOCK_STREAM`)
 - frees up the port used by the socket

The struct sockaddr

- The generic:

```
struct sockaddr {  
    u_short sa_family;  
    char sa_data[14];  
};
```

- **sa_family**

- specifies which address family is being used
- determines how the remaining 14 bytes are used

- The Internet-specific:

```
struct sockaddr_in {  
    short sin_family;  
    u_short sin_port;  
    struct in_addr sin_addr;  
    char sin_zero[8];  
};
```

- **sin_family** = AF_INET
- **sin_port**: port # (0-65535)
- **sin_addr**: IP-address
- **sin_zero**: unused

Address and port byte-ordering

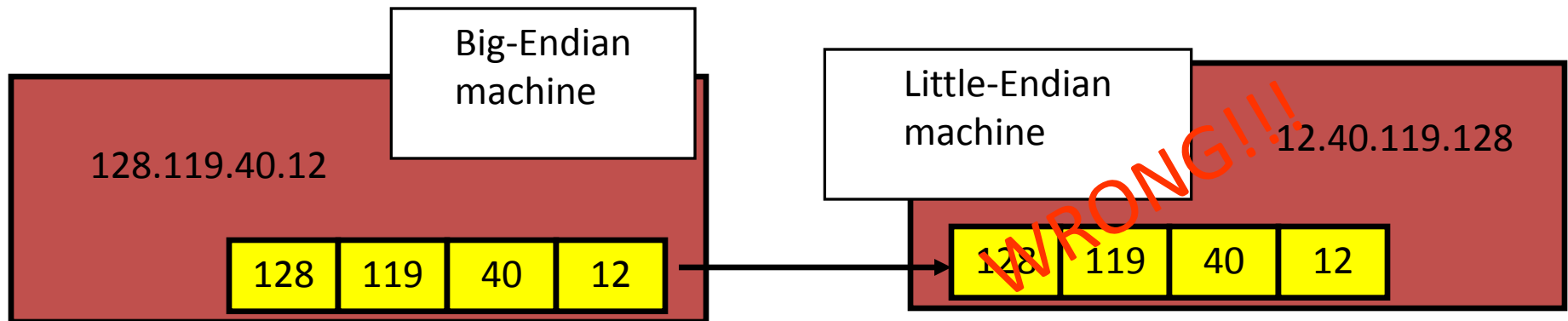
- Address and port are stored as integers

- u_short sin_port; (16 bit)
- in_addr sin_addr; (32 bit)

```
struct in_addr {  
    u_long s_addr;  
};
```

❑ Problem:

- different machines / OS's use different word orderings
 - little-endian: lower bytes first
 - big-endian: higher bytes first
- these machines may communicate with one another over the network



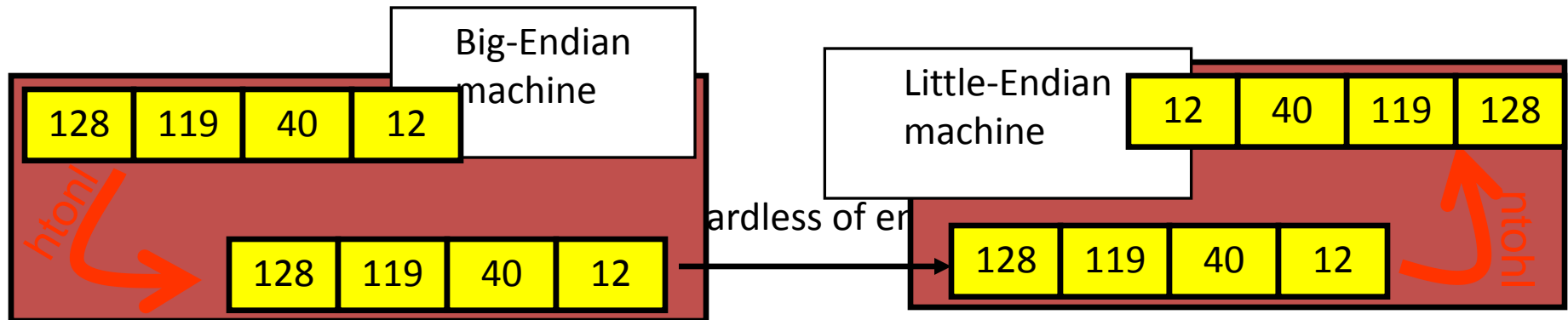
Solution: Network Byte-Ordering

- Defs:
 - Host Byte-Ordering: the byte ordering used by a host (big or little)
 - Network Byte-Ordering: the byte ordering used by the network – always big-endian
- Any words sent through the network should be converted to Network Byte-Order prior to transmission (and back to Host Byte-Order once received)

UNIX's byte-ordering funcs

- `u_long htonl(u_long x);`
- `u_long ntohl(u_long x);`
- `u_short htons(u_short x);`
- `u_short ntohs(u_short x);`

- ❑ On big-endian machines, these routines do nothing
- ❑ On little-endian machines, they reverse the byte order



Dealing with blocking calls

- Many of the functions we saw block until a certain event
 - accept: until a connection comes in
 - connect: until the connection is established
 - recv, recvfrom: until a packet (of data) is received
 - send, sendto: until data is pushed into socket's buffer
 - Q: why not until received?
- For simple programs, blocking is convenient
- What about more complex programs?
 - multiple connections
 - simultaneous sends and receives
 - simultaneously doing non-networking processing

Dealing w/ blocking (cont'd)

- Options:
 - create multi-process or multi-threaded code
 - turn off the blocking feature (e.g., using the `fcntl` file-descriptor control function)
 - use the `select` function call.
- What does `select` do?
 - can be permanent blocking, time-limited blocking or non-blocking
 - input: a set of file-descriptors
 - output: info on the file-descriptors' status
 - i.e., can identify sockets that are “ready for use”: calls involving that socket will return immediately

Buffer Overflow

General Overview

- Can be done on the stack or on the heap.
- Can be used to overwrite the return address (transferring control when returning) or function pointers (transferring control when calling the function)

“Smashing the Stack” (overflowing buffers on the stack to overwrite the return address) is the easiest vulnerability to exploit and the most common type in practice

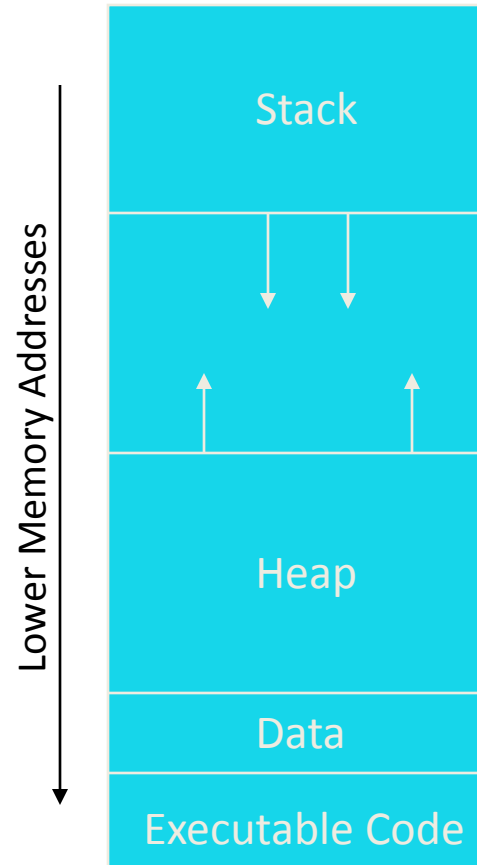
Are Buffer Overflows Really A Problem?

- A large percentage of CERT advisories are about buffer overflow vulnerabilities.
- They dominate the area of remote penetration attacks, since they give the attacker exactly what they want - the ability to inject and execute attack code.

Anatomy of the Stack

Assumptions

- Stack grows down (Intel, Motorola, SPARC, MIPS)
- Stack pointer points to the last address on the stack

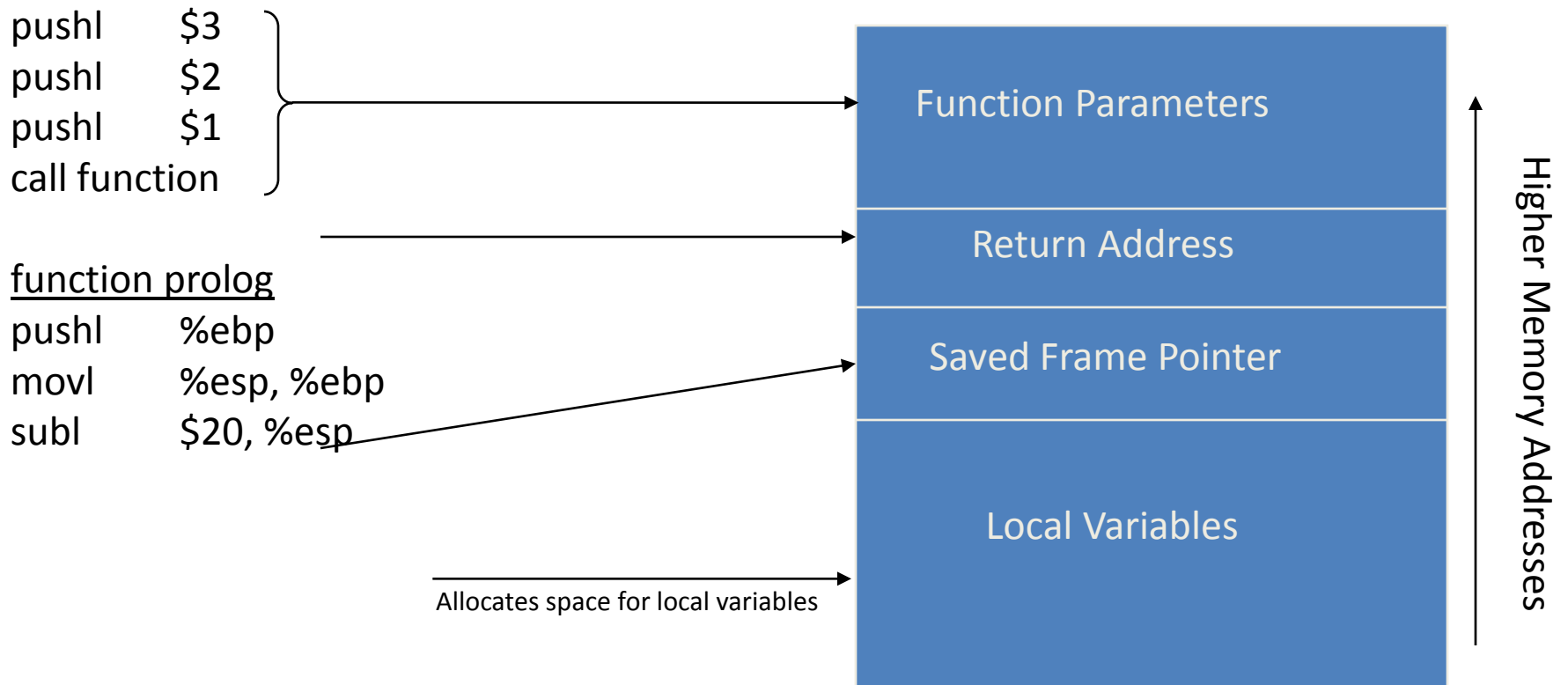


Example Program

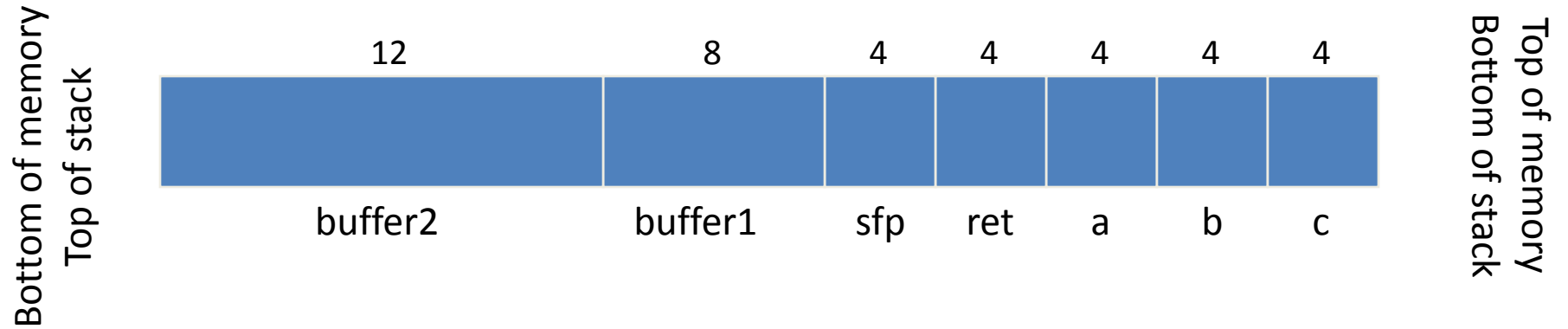
Let us consider how the stack of this program would look:

```
void function(int a, int b, int c){  
    char buffer1[5];  
    char buffer2[10];  
}  
  
int main(){  
    function(1,2,3);  
}
```

Stack Frame



Linear View Of Frame/Stack



Example Program 2

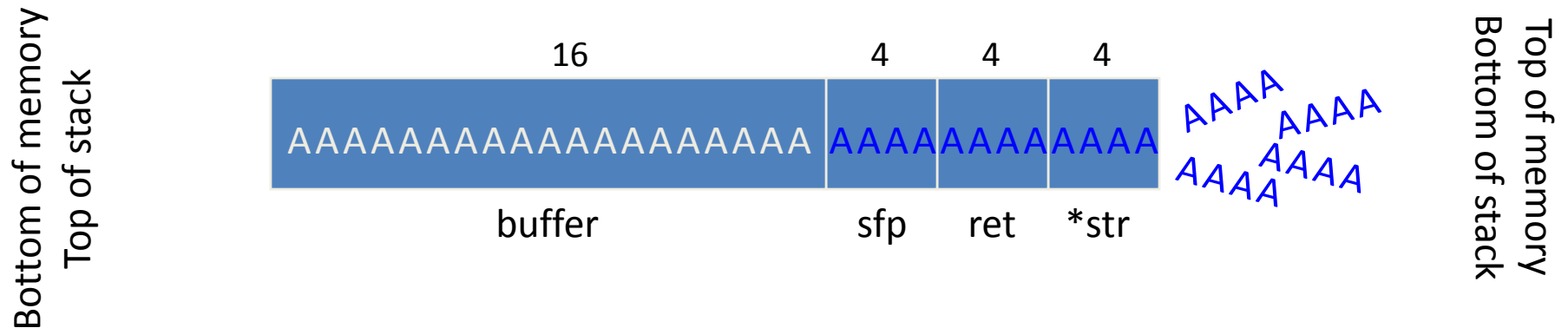
Buffer overflows take advantage of the fact that bounds checking is not performed

```
void function(char *str){
    char buffer[16];
    strcpy(buffer, str);
}

int main(){
    char large_string[256];
    int i;
    for (i = 0; i < 255; i++){
        large_string[i] = 'A';
    }
    function(large_string);
}
```

Example Program 2

When this program is run, it results in a segmentation violation



The return address is overwritten with 'AAAA' (0x41414141)

Function exits and goes to execute instruction at 0x41414141.....

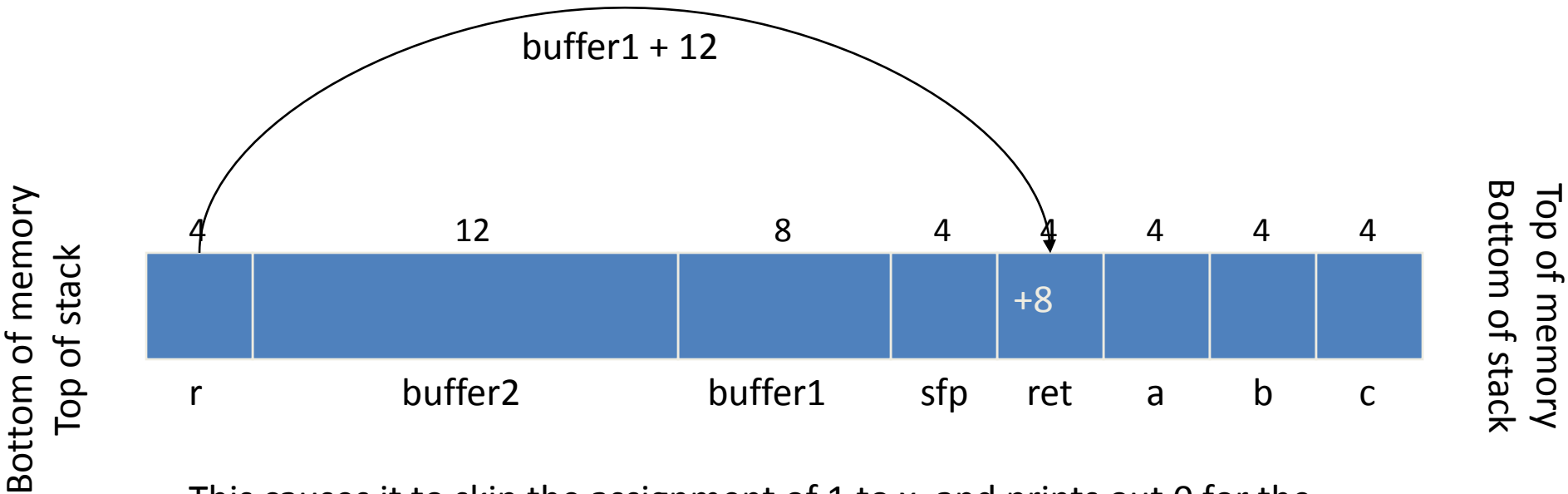
Example Program 3

Can we take advantage of this to execute code, instead of crashing?

```
void function(int a, int b, int c){  
    char buffer1[5];  
    char buffer2[10];  
    int *r;  
    r = buffer1 + 12;  
    (*r) += 8;  
}
```

```
int main(){  
    int x = 0;  
    function(1,2,3);  
    x = 1;  
    printf("%d\n", x);  
}
```

Example Program 3



This causes it to skip the assignment of 1 to x, and prints out 0 for the value of x

Note: modern implementations have extra info in the stack between the local variables and sfp. This would slightly impact the value added to the address of buffer1.

So What?

- We have seen how we can overwrite the return address of our own program to crash it or skip a few instructions.
- How can these principles be used by an attacker to hijack the execution of a program?

How To Find Vulnerabilities

UNIX - search through source code for vulnerable library calls (strcpy, gets, etc.) and buffer operations that don't check bounds. (grep is your friend)