Proactive Computer Security

Shellcode

Ken Friis Larsen
`<kflarsen@diku.dk>`

Department of Computer Science
University of Copenhagen

May 2, 2016

Original slides by Morten Brøns-Pedersen `<f@ntast.dk>`
and Morten Shearman Kirkegaard `<moki@fabletech.com>`

Disclaimer

These slides do not replace the articles, and you should read those as well. The slides are primarily an aid for me to get my points across during the lecture. Don’t hesitate to ask questions during or after the lecture, or on the forum.

Today’s Program

- Shellcode vs. programs
- x86 assembler code
- Considerations when writing shellcode

The goal of an attacker is to take control of some privileged process, so he can act with the privileges of that process.

- Setuid binary (e.g., backup software) – other user / root
- Local daemon (e.g., cron) – other user / root
- Remote server (e.g., web server) – other host
- Remote client (e.g., web browser) – other host

The ultimate control of a target process is known as *arbitrary code execution*

In that case, an attacker can choose a sequence of instructions to be executed in the target process. The instructions are injected into the process as part of the exploitation. Traditionally the code has spawned a shell.

Question

How does a "sequence of instructions" (i.e., shellcode) differ from an executable file (e.g., `C:\windows\system32\cmd.exe` or `/bin/sh`)?

A program is not just a sequence of instructions. Apart from code most programs also need static and dynamic data, some way to call library functions, and so on.

Both Windows and UNIX have several executable formats, but the most used ones are:

UNIX
: ELF (Executable and Linkable Format)

Windows
: PE (Portable Executable)

Shellcode can be more advanced than to spawn a shell, but it is still referred to as "shellcode."

Common shellcode categories:

- Execve(/bin/sh) – only local
- Listen Shell
- Connect-back Shell
- Downloader
- Egg Hunter
- Socket Hunter

Typical shellcode

- Is tiny — often less than 100 bytes
- Contains no `NUL` bytes
- Is position independent
- Does not rely on a specific address space layout

To write good shellcode you need to know your processor and your operating system. Source code is great. Reverse engineering is good enough.

We will look at Intel 80386 shellcode with focus on Linux. The same basic principles apply to other processors and operating systems.

80386 Crash Course

Intel introduced the 32 bit 80386 processor in 1985. Nothing has changed since then.
This is – technically – not accurate. Advanced Micro Devices introduced the 64 bit AMD64 processor series in 2003.

The architecture of the Intel 80386 processor is officially called *IA32*, but is often referred to as *x86* or *i386*.

The *AMD64* architecture is often referred to as *x86-64*, or – by Intel – as *IA-32e* or *EM64T* and – by Microsoft – as *x64*.

Do **not** confuse it with *IA64* (Itanium is an IA64 processor).

Intel x86 assembler.

- Before you do anything else, get a copy of “Intel Architecture Software Developer’s Manual”.
- Two syntaxes: AT&T and Intel (we use Intel).
- Little endian.

Question

What does it mean that the processor is “little endian”?

Question

Actually while we're talking about bits, what is "two's complement representation"? (And why is it smart?)

Extra credit: What 8-bit number is this:

`11110100` = -12 or 244

There are 8 (Almost) general purpose 32bit registers.
See also “The Art of Picking Intel Registers”[1].

EAX	Accumulator. Results.
ECX	Loop counter.
EDX	Data register, I/O pointer. 64-bit extension of EAX.
EBX	Base register (data pointer). General purpose.
ESP	Stack pointer.
EBP	Base pointer.
ESI	Source register.
EDI	Destination register.

[1] http://www.swansontec.com/sregisters.html.

For each register there are one or more extra “registers” which are a subset of the full register. They are:

Register	Lower 16 bits	Lower 8 bits	bits 8–15
EAX	AX	AL	AH
ECX	CX	CL	CH
EDX	DX	DL	DH
EBX	BX	BL	BH
ESP	SP	–	–
EBP	BP	–	–
ESI	SI	–	–
EDI	DI	–	–

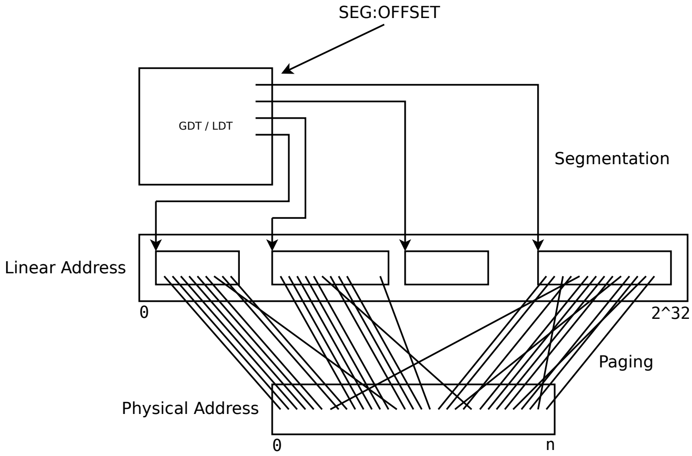
Other registers:

- **EIP** Instruction pointer. Can't be controlled directly.
- **Eflags** Indicate events. For example ZF (zero flag), SF (sign flag) and CF (carry flag). See "Intel Architecture Software Developer's Manual, Volume 1".

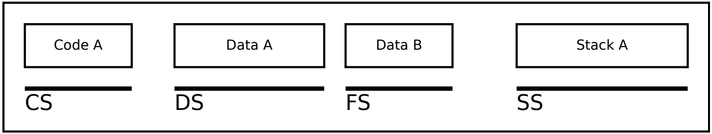
When the 80386 is running in *Protected Mode* all memory references are 48 bit; a 16 bit segment selector, and a 32 bit offset.

The 16 bit segment selector is loaded into a segment register.

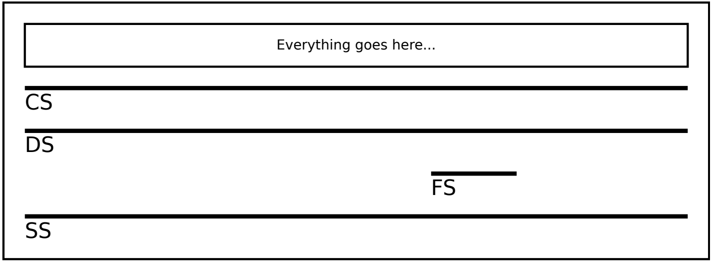
- Instructions are read from the segment in **CS**
- The stack is accessed via the segment in **SS**
- Data is, by default, accessed via the segment in **DS**
- Data can be accessed via the segments in **ES**, **FS** and **GS** by using an instruction prefix.


SEG:OFFSET
GDT / LDT
Segmentation
Linear Address
0
2^32
Paging
Physical Address
0
n

What Intel Imagined:



What Operating System Designers Did:



Segmentation is practically disabled, CS:12345678 is DS:12345678. Code is data and data is code. Pointers are 32 bits.

One exception is a segment register used for TLS. **FS:0000**1234 might be **DS:AAAA**1234 in thread A and **DS:BBBB**1234 in thread B.

Execution:

1. The instruction at CS:EIP is fetched from memory
2. EIP is incremented by the length of the instruction
3. The instruction is executed
4. Rinse, repeat

Some instructions – `call` and `jmp` – change EIP (either by offset or by absolute address). This is done *after* adding the length of the instruction to EIP.

- The instruction at CS:EIP is fetched from memory
- EIP is incremented by the length of the instruction
- `call` pushes EIP (the *return address*) onto the stack
- The offset is added to EIP

Question

What does these instructions do?

`EB 00`	`jmp short 0`	is a no-op.
`EB FE`	`jmp short -2`	is an infinite loop.

Some OP codes

PUSH src Pushes it's operand onto the stack. Decreases ESP according to the operand. If ESP itself is pushed, it is the value of ESP *before* the operation that gets pushed[2].

POP dst Pops a value from the stack and stores it in the operand. Increases ESP accordingly.

MOV dst, src Moves the value of *src* to *dst*. The source operand can be a value, a register or a memory location (using the $[reg1 + reg2 \cdot imm1 + imm2]$ notation). The destination can be a register or a memory location.

ADD dst, src Adds src and dst and stores the result in dst.

[2] GOTCHA: `PUSH BYTE` *imm* pushes 4 bytes (with the first byte sign-extended into the upper three), but `PUSH WORD` *imm* pushes only 2 bytes.

Stack?

What is “the stack” and what is it used for? (We saw one example already.)

More OP codes

SUB dst, src You guessed it.

CMP op1, op2 Compares $op1$ to $op2$ by subtracting $op2$ from $op1$, and sets CF, OF, SF, ZF, AF and PF in Eflags accordingly. Normally used in conjunction with jump instructions.

JZ, JNZ, JL, JLE, JG, JGE Jump according to Eflags. If Eflags was set by CMP op1 op2, then the OP codes mean jump if op1 is equal to, not equal to, less than, less than or equal to, greater than or greater than or equal to op2 respectively. There are many other OP codes in the Jcc family.

JMP dst Jump to dst (register, memory, offset or absolute address).

Even more OP codes

CALL dst — Push the address of the following OP code on the stack and jump to *dst*.

RET — Pop the return address and jump to it.

Look these up yourself:

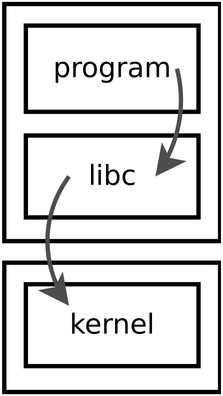
> LEA, PUSHA, POPA, PUSHF, POPF, CDQ, AND, OR, NOT, XOR, BSF, BSR, BSWAP, XCHG, MUL, DIV, NEG, NOP, INC, DEC, ENTER, LEAVE, LOOP, STOS, ROL, ROR, SHL, SHR, XADD, XLAT, and many more...

As we said before, get a copy of “Intel Architecture Software Developer’s Manual”, specifically “Volume 2: Instruction Set Reference”.

System Calls

The shellcode must do something useful.

We need some help from the operating system.


program
libc
kernel

```
AA000000 push byte 13           ; length
AA000002 push dword 0xAA110000  ; buffer
AA000007 push byte 1            ; fd
AA000009 call 0xAA220000        ; write()

AA110000 "hello, world\n"

AA220000 jmp [0xAA330000]       ; Procedure Linkage Table
AA330000 0xBB000000             ; Global Offset Table

BB000000 mov edx, [esp+12]      ; write() in libc
BB000004 mov ecx, [esp+8]
BB000008 mov ebx, [esp+4]
BB00000C mov eax, 4
BB000011 int 0x80
BB000013 ret
```

On Linux IA32 the system call number is passed to the kernel in EAX and the arguments in EBX, ECX, EDX, ESI, EDI and EBP. The result is returned in EAX.

The kernel system call handler is invoked via INT 0x80, SYSCALL or SYSENTER.

System call numbers don't change.

A list of the system calls can be found in `/usr/include/asm/unistd.h` (IA32) or `/usr/include/asm/unistd_{32,64}.h` (AMD64). (Paths might be system dependent)

```
...
#define __NR_exit                1
#define __NR_fork                2
#define __NR_read                3
#define __NR_write               4
#define __NR_open                5
#define __NR_close               6
...
```

write(STDOUT_FILENO, "hello, world\n", 13);

```
        mov edx, 13           ; length
        mov ecx, 0x11223344   ; buffer
        mov ebx, 1            ; fd
        mov eax, 4            ; SYS_WRITE
        int 0x80
        ...
0x11223344:
        db 'hello, world', 10
```

Use the `man` pages in section 2. They describe system calls.

```
man 2 exit
man 2 fork
man 2 read
man 2 write
...
man 2 execve
```

The following is true for Darwin, OpenBSD, FreeBSD, NetBSD, . . .

On BSD IA32 the system call number is passed to the kernel in EAX and the arguments are passed on the stack, as if the kernel system call handler had been called via a regular call instruction; [ESP] is ignored, first argument is in [ESP+4], second in [ESP+8], and so on. The kernel system call handler is invoked via INT 0x80.

Return Value

The result of a system call is returned in EAX, unless the call was to SYS_FORK.

The fork() library call returns

- 0 in the child process
- The PID of the child in the parent process

Coffee Break

Writing Shellcode

Writing Position Independent Code.

Shellcode is often written to a buffer at some unknown address, and the dynamic linker won't help us locate anything.
The shellcode must either:

- Find its own absolute address
- Not rely on knowing the absolute addresses of itself

Finding the Absolute Address.

It would be great to be able to do

EIP →	`...`
EIP →	`mov eax, eip`
EAX EIP →	`...`
EIP →	`add eax, 123`
EIP →	`...`
	`...`
EAX →	`DATA`

... but the 80386 has no such instruction.

Using a trampoline

EIP $\rightarrow$	`...`			
EIP $\rightarrow$	`jmp +123`			
EIP $\rightarrow$	`...`			
EIP $\rightarrow$	`pop eax`		ESP $\rightarrow$	`...`
EIP $\rightarrow$	`...`		ESP $\rightarrow$	`7A DA D5 AD`
	`...`			
EIP $\rightarrow$	`call -118`			
EAX $\rightarrow$	`DATA`			

Call without jump

EIP $\rightarrow$	`...`			
EIP $\rightarrow$	`call +0`			
EAX EIP $\rightarrow$	`...`		ESP $\rightarrow$	`...`
EIP $\rightarrow$	`pop eax`		ESP $\rightarrow$	`7A DA D5 AD`
EIP $\rightarrow$	`...`			
	`...`			

Question

What are the pros and cons of a trampoline (forward jump to backwards call), as opposed to just a call?

Not Relying on the Absolute Address

If we need a string, and we don't know where the shellcode is loaded, we could build this string by moving byte values into a writable location in memory.

But where can we write?

The stack is a very good option. It is writable, and we know its location (ESP).

EIP →	`push dword 0x000A2165`
EIP →	`push dword 0x646F636C`
EIP →	`push dword 0x6C656853`
EIP →	`mov eax, esp`
EIP →	`push dword ...`
EIP →	`push dword ...`
EIP →	`nop`

ESP →	`...`
ESP →	`65 21 0A 00`
ESP →	`6C 63 6F 64`
EAX ESP →	`53 68 65 6C`
ESP →	`...`
ESP →	`...`

EAX points to the string `"Shellcode!\n"` somewhere in memory.

Filters.

Every exploit has its own constraints on the shellcode. A common one is a set of prohibited byte values (`\0`, `\n`, `\r`, ...). This is called a "filter."

CISC is your friend.

Setting a register to zero

```
B8 00 00 00 00  mov eax, 0

31 C0           xor eax, eax

29 C0           sub eax, eax

B8 FF FF FF FF  mov eax, -1
40              inc eax

89 D8           mov eax, ebx
31 D8           xor eax, ebx
```

Setting a register to some arbitrary value

```
B8 E4 58 41 00  mov eax, 0x004158E4

B8 E5 59 40 01  mov eax, 0x014059E5
35 01 01 01 01  xor eax, 0x01010101

B8 40 8E 15 04  mov eax, 0x04158E40
C1 C8 04        ror eax, 4

B8 1B A7 BE FF  mov eax, 0xFFBEA71B
F7 D0           not eax
```

Setting a register to some value less than 2^8

```
31 C0  xor eax, eax
B0 FF  mov al, 0xFF
```

Setting a register to some value less than 2^7

```
6A 41  push byte 0x41
58     pop eax
```

Adding an arbitrary value to a register

```
05 22 11 00 00  add eax, 0x1122

2D DE EE FF FF  sub eax, -0x1122
```

Pro Tips.

Setting a register to zero

Is it even?

```
C1 E0 1F  shl eax, 31
```

Is it positive?

```
C1 E8 1F  shr eax, 31
```

or

```
C1 F8 1F  sar eax, 31
```

Setting EDX to zero when EAX is zero

```
89 C2  mov edx, eax

99     cdq
```

Setting EAX, ECX and EDX to zero

```
31 C9  xor ecx, ecx
F7 E1  mul ecx
```

ECX = 0

EDX:EAX = EAX ∗ ECX

Call to unaligned self...

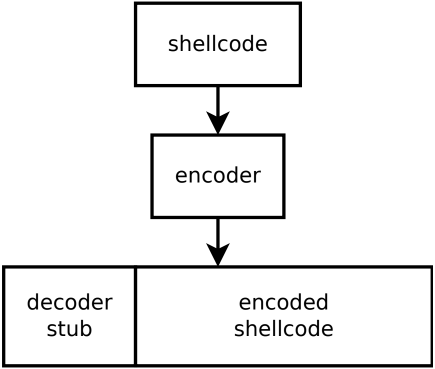
EIP $\rightarrow$	...		
EIP $\rightarrow$	E8		
	FF		
	FF	ESP $\rightarrow$	...
	FF	ESP $\rightarrow$	7A DA D5 AD
EIP $\rightarrow$	FF		
EAX $\rightarrow$	C0		
EIP $\rightarrow$	58		
EIP $\rightarrow$	...		

```
E8 FF FF FF FF          call -1
            FF C0       inc eax (almost nop)
                  58    pop eax
```

Encoders.

If it is not feasible to write the shellcode within the given constraints, an encoded shellcode might be necessary.

The decoder must still be written within those constraints, but that is easier than writing all of the shellcode like that.


shellcode
encoder
decoder stub
encoded shellcode

Alphanumeric Encoded Shellcode

```
LLLLYhb0pLX5b0pLHSSPPWQPPaPWSUTBRDJfh5tDSRajYX0Dka0Tka
fhN9fYf1Lkb0TkdjfY0Lkf0Tkgfh6rfYf1Lki0tkkh95h8Y1LkmjpY
0Lkq0tkrh2wnuX1Dks0tkwjfX0Dkx0tkx0tkyCjnY0LkzC0TkzCCjt
X0DkzC0tkzCj3X0Dkz0TkzC0tkzChjG3IY1LkzCCCC0tkzChpfcMX1
DkzCCCC0tkzCh4pCnY1Lkz1TkzCCCCfhJGfXf1Dkzf1tkzCCjHX0Dk
zCCCCjvY0LkzCCCjdX0DkzC0TkzCjWX0Dkz0TkzCjdX0DkzCjXY0Lk
z0tkzMdgvvn9F1r8F55h8pG9wnuvjrNfrVx2LGkG3IDpfcM2KgmnJG
gbinYshdvD9d
```

Debugging

INT3 (`0xcc`) is your friend. It's a software breakpoint.

shellcode.asm

```
[bits 32]
        int3
        mov eax, 0x11223344
        nop
```

$ **nasm shellcode.asm**	*assemble your code*
$ **ndisasm -u shellcode**	*check for NUL bytes, etc.*
$ **gdb ./demo**	*launch debugger*
$ **gdb ––args ./demo shellcode**	*...with argument*

Sane gdb settings.

Put this in `~/.gdbinit`:

```
set disassembly-flavor intel
set disable-randomization off
set pagination off
set history filename ~/.gdbhistory
set history save
set history expansion
```

```
(gdb) run ./shellcode
Program received signal SIGTRAP, Trace/breakpoint trap.
0xf7fdd001 in ?? ()
 (gdb) x/3i $eip-1

        0xf7fdd000:   int3
  =>    0xf7fdd001:   mov eax,0x11223344
        0xf7fdd006:   nop
 (gdb) info registers eax

eax 0x26718662 644974178
```

```
(gdb) stepi
0xf7fdd006 in ?? ()
(gdb) x/3i $eip-6
        0xf7fdd000:   int3
        0xf7fdd001:   mov eax,0x11223344
   =>   0xf7fdd006:   nop
(gdb) info registers eax
eax 0x11223344 287454020
```

Failed System Calls

shellcode.asm

```
        ; execve syscall
        push byte 11
        pop eax
        int 0x80

        ; breakpoint
        int3
```

(gdb) **run ./shellcode**
Program received signal SIGTRAP, Trace/breakpoint trap.
0xf7fdd010 in ?? ()

(gdb) **info registers eax**

eax 0xfffffffe -2

If EAX is negative, an error occurred. You can find a table of error codes in `/usr/include/asm-generic/errno-base.h`.

```
...
#define EPERM     1   /* Operation not permitted */
#define ENOENT    2   /* No such file or directory */
#define ESRCH     3   /* No such process */
#define EINTR     4   /* Interrupted system call */
...
```

$ **strace ./demo ./shellcode**

...

execve("|\215\25", [0x10], [/* 8 vars */])

= -1 ENOENT (No such file or directory)

Note that the execve *library call* returns -1 on error, while the execve *system call* returns an error code.

The wrapper in the C library saves the error code in `errno` and returns -1. The strace output reflects this, even if you are not using the C library.

Determining if the shellcode is running

If you can’t attach a debugger (e.g., because you are trying to exploit a program on a machine you don’t have access to), it can be very valuable to know whether your code is running or not. Usually it is possible to differentiate an infinite loop from a crash.

Question

What is the shortest byte sequence that results in an infinite loop on x86?

```
eb fe    jmp short -2
```

Summary

- Shellcode is a sequence of instructions to be executed in the target process.
- Shellcode must sometimes adhere to filters (size, allowed bytes, ...).
- Use `int 0x80` to make system calls.
- Student presentations: There will be an assignment, where you hand-in your preferred topic(s) and preferences for dates. Priority will be given to uniqueness of topic and flexibility (and extra credit for May 13).