Exploiting Introduction

Andrea Mambretti (m4mbr3@gmail.com)

Politecnico di Milano

April 18, 2013

Crash course on Assembly Language

- Overview on the common 32-bit Intel Architecture (IA)
- Overview on different syntaxes
- Basic Instructions
- x86_64

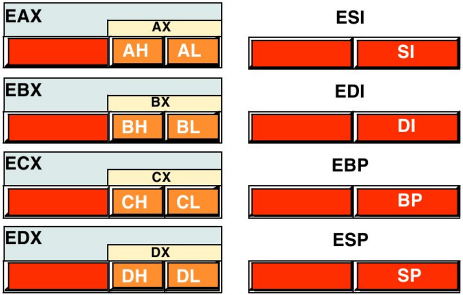
Something more

- Program layout in memory
- Function and call convention
- GDB tutorial
- Conclusion

(1) Does AI look like?

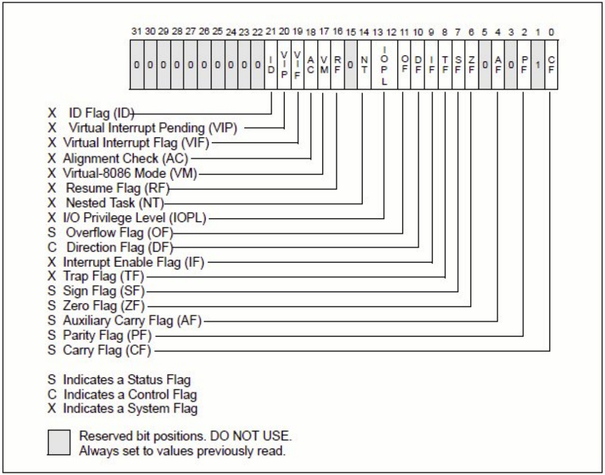
- The processor has 32-bits internal registers to manage and execute operations on data
  They are EAX, EBX, ECX, EDX, ESI, EDI, EBP, EIP and ESP
- Among them EAX, EBX, ECX, EDX are for general purposes
- EBP (BP = base pointer) and ESP (SP = stack pointer) are the stack bounds (see slide 24)
- EDI and ESI are extra registers
- EIP (IP = instruction pointer) is the register that contains the address of the next instruction

(2) Does AI look like?



- The register name system is a porting from 16-bits IA where the registers were called AX, BX and so on.
  "E" means extended. Without it we consider the corrisponding 16-bits register

(1) What about EFLAGS?



(2) What about EFLAGS?

- It's another 32-bits register
- Only 8 bits out of 32 are of interest for us. The others are either for the kernel mode function or are of little interest for programmers
- These 8 bits are called flags. We consider them singularly. They are boolean (true/false)
- They represent overflow, direction, interrupt disable, sign, zero, auxiliary carry, parity and carry flags
- Since they represent information about the instruction last executed, they change at every execution step. They are VERY important for the control flow of the program (see slide 18)

Syntax

- In the assembly world we can find two main syntaxes: the AT&T and the Intel
- AT&T syntax is used by all UNIX program (e.g. gdb)
- Intel syntax is used by Microsoft programs (IDApro and others)

(1) Differences in the notation

- Consider the following operation:

  "move the value 0 to EAX"

- AT&T:

  mov $0x0,%eax

- Intel:

  mov eax, 0h

- Comments:
  - As you can see in AT&T syntax the destination is the second operand instead as in the Intel syntax
  - In the AT&T syntax the register are denoted with % and the [immediate | costant] with $. In the Intel syntax these tokens are not used.

(2) Differences in the notation

- Consider this new operation:

  "move the value 0 to the address contained in EBX+4"
- AT&T:

  mov $0x0,0x4(%ebx)
- Intel:

  mov [ebx+4h],0h
- Comments:
  - This case shows how the deferentiation is done in assembly
  - In AT&T we use parenthesis. In the Intel syntax we have to use square brackets
  - The way to manage the offset is another syntax difference. In the first case we have to put it out of the parenthesis in the second one inside the square brackets

(1) Basic instructions overview

- Every processor has a huge instruction set (see Intel Manual[1])
- A subset of the whole instruction set is usually processor dependent
- We will focus on the subset of instructions that is common among the processors
- We will use the Intel syntax as it is the same syntax used in IDApro by default

[1] http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

(2) Basic Instruction MOV

- General: MOV **destination**, **source**
  source can be an immediate, a register, a memory location
  destination can be either a register or a memory location
  NB: Every combination is possible except memloc to memloc!!!It is invalid in every instruction!!!
- With this instruction, as said in the example above, we move a value from a source place to a destination. There are a ton of different versions. They change depending on the operands ex 32 bits operands, 16 bits operands, immediate to reg, immediate to memory
- Examples

MOV eax, ebx	MOV eax, FFFFFFFFh	MOV ax, bx
MOV [eax],ecx	MOV [eax],[ecx] NO!!!	MOV al, FFh

(3) Basic Instruction ADD

- General: ADD **destination**, **source**
  source can be an immediate, a register, a memory location
  destination can be either a register or a memory location
  NB: The destination register has to be as big as the source or greater
- With this instruction we can add a value from **source** to the destination operand and put the new value inside the destination (dest = dest + src)
- Examples

ADD esp, 44h	ADD eax, ebx	ADD al, dh
ADD edx, cx	ADD [eax],[ecx] NO!!!	ADD [eax],1h

(4) Basic Instruction SUB

- General: SUB **destination**, **source**
  source can be an immediate, a register, a memory location
  destination can be either a register or a memory location
  NB: The destination register has to be as big as the source or greater
- With this instruction we can subtract the value **source** from the destination operand and put the new value inside the destination (dest = dest - src)
- Examples

SUB esp, 33h	SUB eax, ebx	SUB al, dh
SUB edx, cx	SUB [eax],[ecx] NO!!!	SUB [eax],1h

(5) Basic Instruction MUL

- General: MUL **Operand**
  Operand can be an immediate, a register, a memory location
- With this instruction we can multiply **Operand** by the value of corresponding byte-length in the EAX,AX,AL register

OperandSize:	1 byte	2 bytes	4 bytes
Other Operand	AL	AX	EAX
Higher Part of result:	AH	DX	EDX
Lower Part of result:	AL	AX	EAX

- Examples

OperandSize:	1 byte	2 bytes	4 bytes
Immediate	MUL 44h	MUL 4455h	MUL 44556677h
Register	MUL cl	MUL dx	MUL ebx

(6) Basic Instruction DIV

- General: DIV **Operand**
  Operand can be an immediate, a register, a memory location
- With this instruction we can divide the value in the dividend register(s) by "Operand"

OperandSize:	1 byte	2 bytes	4 bytes
Dividend	AX	DX:AX	EDX:EAX
Remainder	AH	DX	EDX
Quotient	AL	AX	EAX

- Examples

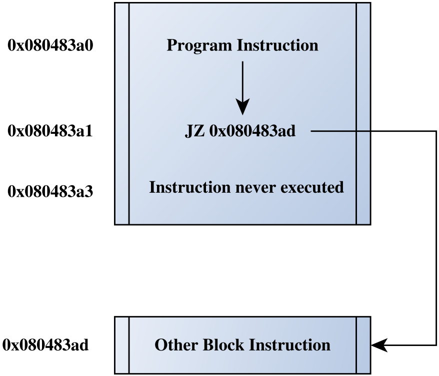
OperandSize:	1 byte	2 bytes	4 bytes
Register	DIV bl	DIV bx	DIV ebx
Immediate	DIV 66h	DIV 6677h	DIV 66778899h

(7) Basic Instruction CMP

- General: CMP **Operand_1**, **Operand_2**
- This instruction performs a subtraction between two operands and sets the flags (ZF,CF,OF etc.), it doesn't store the result
- Examples

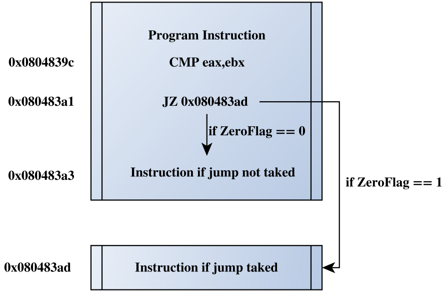
CMP eax, ebx	CMP eax, 44BBCCDDh	CMP al, dh
CMP al, 44h	CMP ax,FFFFh	CMP [eax],4h

(8) Basic Instruction JMP



- General: JMP **<u>address</u>**
- This instruction is called "unconditional jump": when called, it sets the EIP to the address passed to the instruction. We say that the execution jumps to **address** and it's unconditional because always the execution jump.

(9) Basic Instruction JZ,JNZ and so on



- General: JX **address**
  $X \in \{O, NO, S, NS, E, Z, NE \ldots\}$
- This set of instructions are called conditional jump. It means that[2] the execution will go to **address** if and only if the specific flag of the condition is verified.
  For example: jz jumps only if zero flag is 1

[2] http://www.unixwiz.net/techtips/x86-jumps.html

(10) Basic Instruction INT

- General: INT **VALUE**
- **VALUE** is the software interrupt that should be generated (0-255)
- Famous values are 21h for call service under windows and 80 for linux
- look the manual for the other

How much is x86_64 different from x86?

- The prefix of the registers is R instead of E so we have (rip, rax etc.)
- There are 8 new registers (r8 to r15)
- each of them can be consider at 8, 16, 32, 64 bits
  with $X \in \{8..15\}$ we have

bits	8	16	32	64
reg	rXb	rXw	rXd	rX

- for better syntax information look at
  `http://www.x86-64.org/documentation/assembly.html`

Binary File Formats

- PE (Portable Executable): used by Microsoft binary executable.
- ELF: common binary format for Unix, Linux, FreeBSD and others

How a program is seen in memory in linux (ELF)

Executable	Description
.bss	This section holds uninitialized data that contribute to the program's memory image. By definition, the system initializes the data with zeros when the program begins to run.
.comment	This section holds version control information.
.data/.data1	These sections hold initialized data that contribute to the program's memory image
.debug	This section holds information symbolic debugging.
.text	This section holds the "text," or executable instructions, of a program.
.init	This section holds executable instructions that contribute to the process initialization code.
	That is, when a program starts to run, the system arranges to execute the code in this section before calling the main program entry point (called main for C programs).
.got	This section holds the global offset table.

How a program is seen in memory in windows (PE)

Executable	Description
.text	Contains the executable code
.rdata	Holds read-only data that is globally accessible within the program
.data	Stores global data accessed throughout the program
.idata	Sometimes present and stores the import function information; if this section is not present, the import function information is stored in the .rdata section
.edata	Sometimes present and stores the export function information; if this section is not present, the export function information is stored in the .rdata section
.pdata	Present only in 64-bit executables and stores execption-handling information
.rsrc	Stores resources needed by the executable
.reloc	Contains information for relocation of library files

A more realistic view of an elf in memory

↑ Lower addresses (0x08000000)

`Shared libraries`

`.text`

`.bss`

`Heap (grows` ↓)

`Stack (grows` ↑)

`env pointer`

`Argc`

↓ Higher addresses (0xbfffffff)

The stack

- Is a data structure, more specifically a Last In First Out data structure, which means that the most recent data placed, or pushed, onto the stack is the next item to be removed, or popped, from the stack
- A LIFO data structure is ideal for storing transitory information, or information that does not need to be stored for a lengthy period of time.
- The stack stores local variables, information relating to function calls, and other information used to clean up the stack after a function or procedure is called.
- Another important feature of the stack is that it grows down the address space.

Stack management: PUSH

- It is the instruction that stores information onto the stack
- It is used for example as PUSH **immediate** or PUSH **register**
- This instruction save the value at the top of the stack and then decreases the ESP of the dimension of data stored

Stack management: POP

- It is the instruction that removes information from the stack
- It is used for example as POP **destination**
- This instruction removes the value at the top of the stack and then increases the ESP of the dimension of data that was retrieved. The information retrieved is store in destination

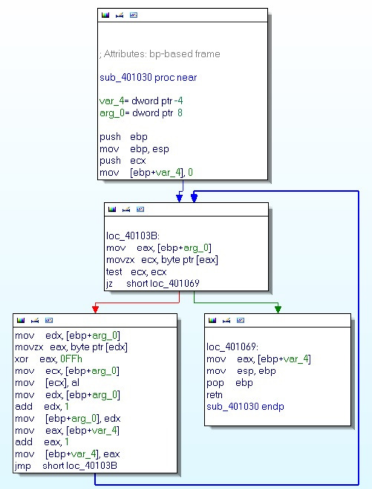
Function

- The concept of function in assembly is the same of the common function in almost all the programming languages
- A piece of code that receives data from the caller and returns some value after the elaboration
- Differently from all high level languages the way to pass parameters to a function can be done in more than one way
- Exist also several ways to call function in assembly

CALL and RET Instruction

- The CALL instruction used CALL **func** moves the exectuion to the **func** location. It stores ebp and return address of the instruction after the call onto the stack and put into EIP the address of the first instruction of **func**
- The RET instruction used RET **value** returns to the previous execution function. It restores the base pointer and puts the ret value saved by the CALL into the EIP.

Function, An example



Calling Convention

- It's the way how a program receives parameters,how a function returns its return value and who cleans the stack
- There are different implementation of the Call Convention that dictates exactly where a caller should place any parameters that a function requires
- Everything is dipendent by the compiler(gcc/g++, visual studio c++ etc.) and by the high-level language from which the assembly comes from (c, c++, visualbasic and so on)
- Let's look at some of those

The C Calling Convention

- It's the default calling convention used by most C compilers for the x86 arch
- When a compiler doesn't use this convention we can force it using the modifier _cdecl
- It specifies that the **caller places** parameters to a function on a stack in the right to left order and that the **caller removes** the parameters from the stack after the called function completes

The C Calling Convention example

```
void demo_cdecl(int w, int x, int y, int z);
```

```
; demo_cdecl(1, 2, 3, 4);   //programmer calls demo_cdecl
push    4             ; push parameter z
push    3             ; push parameter y
push    2             ; push parameter x
push    1             ; push parameter w
call    demo_cdecl    ; call the function
add     esp, 16       ; adjust esp to its former value
```

```
demo_cdecl(1, 2, 3, 4);   //programmer calls demo_cdecl
 mov    [esp+12], 4    ; move parameter z to fourth position on stack
 mov    [esp+8], 3     ; move parameter y to third position on stack
 mov    [esp+4], 2     ; move parameter x to second position on stack
 mov    [esp], 1       ; move parameter w to top of stack
 call    demo_cdecl   ; call the function
```

The Standard Calling Convention

- This is the Miscrosoft Calling Convention standard
- When a compiler doesn't use this convention we can force it using the modifier _stdcall
- Also here the parameters are passed all using only the stack, the difference is that the called function is responsible for clearing the function parameters from the stack when the function has finished. To do this the function has to know the right number of parameter passed. It's valid only with function with fixed number of parameters so such as printf can't use it.

```
void _stdcall demo_stdcall(int w, int x, int y);
```

```
ret 12      ; return and clear 12 bytes from the stack
```

fastcall Convention for x86

- It's a variation of the stdcall convention, the fastcall calling convention passes up to two parameters in CPU registers which are faster to access than stack.
- The Microsoft Visual C/ C++ and GNU gcc/g++ compilers recognize the fastcallmodifier in function declaration.
- In this case the first two parameters are passed in the register (ECX and EDX), the remaining parameters are place on the stack in right to left order similar to stdcall.The function is responsible for removing parameters from the stack when they return to their caller.

```
void fastcall demo_fastcall(int w, int x, int y, int z);
```

```
; demo_fastcall(1, 2, 3, 4);    //programmer calls demo_fastcall
  push   4               ; move parameter z to second position on stack
  push   3               ; move parameter y to top position on stack
  mov    edx, 2          ; move parameter x to edx
  mov    ecx, 1          ; move parameter w to ecx
  call   demo_fastcall   ; call the function
```

What we need is the C standard convention

```
void function (int a, int b){
    int array [5];
}


int main(int argc, char** argv)
{
    function(1,2);
    printf("This is where the \
           return address points");
}
```

Low memory addresses and top of the stack

Array
EBP
RET
A
B

High memory addresses and Bottom of the stack

Asm view of the same example

```
public main
main proc near
push   ebp
mov    ebp, esp
and    esp, 0FFFFFFF0h
sub    esp, 10h
mov    dword ptr [esp+4], 2
mov    dword ptr [esp], 1
call   function
mov    dword ptr [esp], offset format ; "This is '
call   _printf
leave
retn
main endp
```

```
public function
function proc near
push   ebp
mov    ebp, esp
sub    esp, 20h
leave
retn
function endp
```

Our friend gdb

- **What is GDB?**
  GDB is the GNU Project debugger, allows you to see what is going on 'inside' another program while it executes – or what another program was doing at the moment it crashed.'[3]

[3] http://www.gnu.org/software/gdb/

Start, break and navigate the execution with gdb

- Suppose to have the program 'first' and you want run it into gdb
  - **gdb ./first** load the binary information in gdb
- Now you decide to start the program with two parameters
  - **run 1 "abc"** pass an integer as arg[1] and "abc" as arg[2]
  - **run ‘printf "AAAAAAAAAAA"‘** in this case we're passing the output of the bash command
- Suppose you want, now, to stop the execution at the address of a certain instruction
  - **break *0xDEADBEAF** points a break at that address
  - **break *main+1** if you have the debugging symbols this can be less painful
  - **catch syscall** block the execution when a syscall happens

Start, break and navigate the execution with gdb

- Now the execution has been stoped by our break point. Here we can do several things...
- To procede we can:
  - **ni** allows to procede instruction per instruction
  - **next 4** move, if you have the lines number in the binary, 4 lines ahead
  - **continue** goes directly to the next breakpoint
- To see info about the execution state:
  - **info registers** to see the values assumed by the registers
  - **info frame** to see the values of the stack frame releted to the function where we are in
  - **info file** print the information about the sections of the binary

Navigate the stack

- We are always stopped somewhere in the code and we want to evaluate the stack
- Some useful view of the stack is achievable with:
  - **x/100wx $esp** prints 100 elements of the stack from the esp to down in exadecimal word by word form
  - **x/10wo $ebp-100** prints 10 elements of the stack from the ebp-100 to down in octal word by word form
  - **x/s $eax** prints the elements pointed by eax as string form in byte form
- Have you the debug symbols?
  - **print args** prints info about the main parameters
  - **print a** prints the variable a value
  - **print *b** prints the value pointed by b

Our friend gdb

- **The '∼/.gdbinit' file**
  Gdb is a command line tool and it supports the configuration script as almost all the *nix software.
  Some options that you may want to tune are:
  - **set history save on**
    To have the lastest commands always available also when we re-open gdb
  - **set follow-fork-mode child**
    Allows you, if the process spones childs, to follow them and not only wait their end.
  - **set disassembly-flavor [intel | att]**
    This option sets in which predefined syntax your disassembled will be showed up. The default one is at&t

Layout in gdb

- Aren't you a fun of the gdb command line?
- Give a simple text interface to it
  - **layout asm** turn the interface to the assembly view always visible during debugging
  - **layout src** if your binary has the dubugging symbols you will have your c source view visible
  - **layout reg** add to the interface the register status view. It could be used in combination with one of the view described above
  - **gdb -tui ./mybin** runs gdb directly in this Text User Interface

Bibliography

- The Ida Pro Book 2 Edition
- The Shellcoder Handbook
- Reverse Engineering Code with IDA Pro
- Secrets of Reverse Engineering

End

- Thanks Folk...Questions?