03-Basic Dynamic Analysis & Assembly Language

CYS5120 - Malware Analysis

Bahcesehir University

Cyber Security Msc Program

Dr. Ferhat Ozgur Catak [1] Mehmet Can Doslu [2]

[1] ozgur.catak@tubitak.gov.tr

[2] mehmetcan.doslu@tubitak.gov.tr

2017-2018 Fall

Table of Contents

Table of Contents

Introduction I

Definition

- *Dynamic analysis* is any examination performed **after executing malware**.
- Dynamic analysis techniques are **the second step** in the malware analysis process.
- Dynamic analysis is typically performed after basic static analysis has reached a dead end, whether due to
  - *obfuscation*, *packing*, or *the analyst having exhausted* the available static analysis techniques.
- It can involve monitoring malware as it runs or examining the system after the malware has executed.

Introduction II

Static vs Dynamic

- Unlike static analysis, dynamic analysis lets you observe the malware's **true functionality**.
  - for example, the existence of an action string in a binary does not mean the action will actually execute.
- Dynamic analysis is also an **efficient way to identify malware functionality**.
  - For example, if your malware is a keylogger, dynamic analysis can allow you to locate the **keylogger's log file on the system**, discover the **kinds of records it keeps**, **decipher where it sends its information**, and so on.

Caution!

- Although dynamic analysis techniques are extremely powerful, they should be performed only after basic static analysis has been completed
- **Dynamic analysis can put your network and system at risk.**

Sandboxes I

Sandboxes

- Several all-in-one software products can be used to perform basic dynamic analysis,
  - and the most popular ones use *sandbox technology*.
- A **sandbox** is a security mechanism for **running untrusted programs** in a **safe environment** without fear of harming *real* systems.

Sandboxes II

Sandbox Drawbacks

- The sandbox **may not record all events**, because neither you nor the sandbox may **wait long enough**.
  - If the malware is set to sleep for a day before it performs malicious activity, you may miss that event.
- Malware **often detects when it is running in a virtual machine**, and if a virtual machine is detected, the **malware might stop running or behave differently**.
- Some malware requires the presence of **certain registry keys** or **files** on the system that might not be found in the sandbox.
  - StuxNet requires Siemens S7-300 PLCs
- If the malware is a DLL, certain exported functions will not be invoked properly, because a DLL will not run as easily as an executable.
- The sandbox environment OS may not be correct for the malware.
- A sandbox cannot tell you what the malware does. It may report basic functionality,
  - It cannot tell you that the malware is a custom Security Accounts Manager (SAM) hash dump utility or an encrypted keylogging backdoor,

Running Malware

- In this course, we focus on running the majority of malware you will encounter (EXEs and DLLs).

Running DLL

- The program rundll32.exe is included with all modern versions of Windows. It provides a container for running a DLL using this syntax:
  `C:\>rundll32.exe DLLname, Export arguments`
- The *Export* value must be a function name or ordinal selected from the exported function table in the DLL.
  - You can use a tool such as *PEview* or *PE Explorer* to view the Export table.

Monitoring with Process Monitor I

Process Monitor (Procmon)

- **Process Monitor**, or **procmon** [1], is an advanced monitoring tool for Windows that provides a way to monitor certain **registry**, **file system**, **network**, **process**, and **thread** activity.

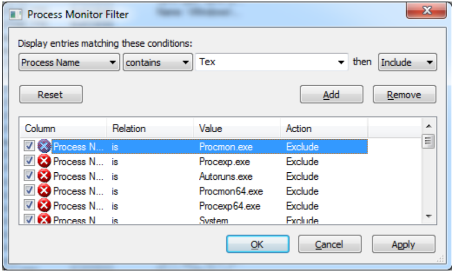
Time ...	Process Name	PID	Operation	Path	Result	Detail
16:12:...	AvastSvc.exe	1484	ReadFile	C:\Windows\System32\ntdll.dll	SUCCESS	Offset: 1.322.496, ...
16:12:...	AvastSvc.exe	1484	ReadFile	C:\Windows\System32\ntdll.dll	SUCCESS	Offset: 1.310.208, ...
16:12:...	AvastSvc.exe	1484	ReadFile	C:\Windows\System32\ntdll.dll	SUCCESS	Offset: 1.199.104, ...
16:12:...	AvastSvc.exe	1484	FileSystemControl	C:\USERS\Public\Desktop\TeXstudio.l...	SUCCESS	Control: FSCTL_R...
16:12:...	AvastSvc.exe	1484	ReadFile	C:\USERS\Public\Desktop\TeXstudio.l...	SUCCESS	Offset: 0, Length: 1...
16:12:...	AvastSvc.exe	1484	QueryNameInfo...	C:\Windows\explorer.exe	SUCCESS	Name: \Windows\...
16:12:...	Explorer.EXE	1748	ReadFile	C:\Windows\System32\KernelBase.dll	SUCCESS	Offset: 373.760, Le...
16:12:...	Explorer.EXE	1748	ReadFile	C:\Windows\System32\kernel32.dll	SUCCESS	Offset: 1.057.792, ...
16:12:...	Explorer.EXE	1748	ReadFile	C:\Windows\System32\shell32.dll	SUCCESS	Offset: 5.163.520, ...
16:12:...	Explorer.EXE	1748	ReadFile	C:\Windows\System32\shell32.dll	SUCCESS	Offset: 5.179.904, ...
16:12:...	Explorer.EXE	1748	ReadFile	C:\Windows\System32\shell32.dll	SUCCESS	Offset: 5.017.600, ...
16:12:...	Explorer.EXE	1748	ReadFile	C:\Windows\System32\shell32.dll	SUCCESS	Offset: 5.033.984, ...
16:12:...	Explorer.EXE	1748	ReadFile	C:\Windows\System32\shell32.dll	SUCCESS	Offset: 5.257.728, ...
16:12:...	vmware-authd....	4336	RegQueryValue	HKLM\System\CurrentControlSet\servic...	NAME NOT FOUND	Length: 20
16:12:...	Explorer.EXE	1748	ReadFile	C:\Windows\System32\shell32.dll	SUCCESS	Offset: 5.278.208, ...
16:12:...	vmware-authd....	4336	CreateFile	C:\Windows\SysWOW64\sysmain.dll	NAME NOT FOUND	Desired Access: R...
16:12:...	Explorer.EXE	1748	ReadFile	C:\Program Files (x86)\Dropbox\Client\...	SUCCESS	Offset: 237.056, Le...

Monitoring with Process Monitor II

Filtering in Procmon

- It's not easy to **find information** in procmon when you are looking through **thousands of events**, one by one.
- You can set procmon to filter on one executable running on the system.
  - This feature is particularly useful for malware analysis, because you can set a filter on the piece of malware you are running.
  - You can also filter on individual system calls such as *RegSetValue*, *CreateFile*, *WriteFile*, or other suspicious or destructive calls.
- To set a filter, choose *Filter* $\rightarrow$ *Filter* to open the *Filter* menu,

Monitoring with Process Monitor III



Seq...	Time...	Process Name	Operation	Path	Result
0	4:18:5...	mm32.exe	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS
1	4:18:5...	mm32.exe	RegSetValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders\C...	SUCCESS
2	4:18:5...	mm32.exe	RegSetValue	HKLM\SOFTWARE\SAXP32\F4KL\Options	SUCCESS
3	4:18:5...	mm32.exe	RegSetValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\Sys32V2Contoller	SUCCESS
4	4:18:5...	mm32.exe	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS
5	4:18:5...	mm32.exe	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS
6	4:18:5...	mm32.exe	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS
7	4:18:5...	mm32.exe	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS
8	4:18:5...	mm32.exe	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS
9	4:18:5...	mm32.exe	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS
10	4:18:5...	mm32.exe	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS

Monitoring with Process Monitor IV

Filtering in Procmon

- The most important filters for malware analysis are *Process Name*, *Operation*, and *Detail*.
- Next, select a comparator, choosing from options such as *Is*, *Contains*, and *Less Than*.
- Finally, choose whether this is a filter to **include** or **exclude** from display.

Monitoring with Process Monitor V

Procmon Filters

- Procmon provides helpful automatic filters on its toolbar.

Registry By examining registry operations, you can tell how a piece of malware installs itself in the registry.

File system Exploring file system interaction can show all files that the malware creates or configuration files it uses.

Process activity Investigating process activity can tell you whether the malware spawned additional processes.

Network Identifying network connections can show you any ports on which the malware is listening.

Monitoring with Process Monitor VI

All four filters are selected by **default**. To **turn off** a filter, simply **click the icon** in the toolbar corresponding to the category.

Procmon Analysis

- Analysis of procmon's recorded events takes **practice** and **patience**, since *many events* are simply part of the standard way that **executables start up**. The more you use procmon, the easier you will find it to quickly review the event listing.

[1] https://docs.microsoft.com/en-us/sysinternals/downloads/procmon

Viewing Processes with Process Explorer I

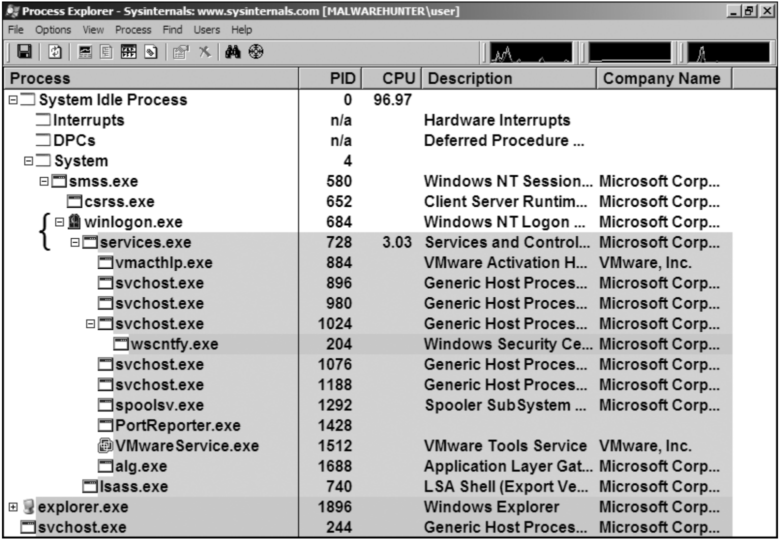
- **The Process Explorer**, free from Microsoft, is an extremely **powerful task manager** that should be running when you are performing dynamic analysis.
- It can provide **valuable insight** into the **processes currently running on a system**.
- You can use Process Explorer to list active *processes*, *DLLs loaded by a process*, *various process properties*, and *overall system information*.
- You can also use it to **kill a process**, **log out users**, and **launch and validate processes**.

Viewing Processes with Process Explorer II

The Process Explorer Display

- Process Explorer **monitors the processes** running on a system and **shows them in a tree structure** that displays **child and parent relationships**
- Process Explorer shows five columns:
  - Process (the process name)
  - PID (the process identifier)
  - CPU (CPU usage)
  - Description
  - Company Name

Viewing Processes with Process Explorer III



Process Explorer - Sysinternals: www.sysinternals.com [MALWAREHUNTER\user]

File Options View Process Find Users Help

Process	PID	CPU	Description	Company Name
System Idle Process	0	96.97		
Interrupts	n/a		Hardware Interrupts	
DPCs	n/a		Deferred Procedure ...	
System	4			
smss.exe	580		Windows NT Session...	Microsoft Corp...
csrss.exe	652		Client Server Runtim...	Microsoft Corp...
winlogon.exe	684		Windows NT Logon ...	Microsoft Corp...
services.exe	728	3.03	Services and Control...	Microsoft Corp...
vmacthlp.exe	884		VMware Activation H...	VMware, Inc.
svchost.exe	896		Generic Host Proces...	Microsoft Corp...
svchost.exe	980		Generic Host Proces...	Microsoft Corp...
svchost.exe	1024		Generic Host Proces...	Microsoft Corp...
wscntfy.exe	204		Windows Security Ce...	Microsoft Corp...
svchost.exe	1076		Generic Host Proces...	Microsoft Corp...
svchost.exe	1188		Generic Host Proces...	Microsoft Corp...
spoolsv.exe	1292		Spooler SubSystem ...	Microsoft Corp...
PortReporter.exe	1428			
VMwareService.exe	1512		VMware Tools Service	VMware, Inc.
alg.exe	1688		Application Layer Gat...	Microsoft Corp...
lsass.exe	740		LSA Shell (Export Ve...	Microsoft Corp...
explorer.exe	1896		Windows Explorer	Microsoft Corp...
svchost.exe	244		Generic Host Proces...	Microsoft Corp...

Viewing Processes with Process Explorer IV

The Process Explorer Display

- The view updates every second.
- ***services*** *are highlighted in* ***pink***, ***processes*** *in* ***blue***, ***new processes*** *in* ***green***, and ***terminated processes*** *in* ***red***.

Viewing Processes with Process Explorer V

Using the Verify Option

- One particularly useful Process Explorer feature is the **Verify button** on the **Image tab**.
- **Microsoft uses digital signatures** for most of its core executables, when Process Explorer verifies that a signature is valid,
  - you can be sure that the **file is actually the executable from Microsoft**.
  - Malware often **replaces authentic Windows files** with its own in an attempt to hide.
- The Verify button **verifies the image on disk rather than in memory**
  - it is useless if an attacker uses *process replacement*
  - which involves running a process on the system and **overwriting its memory space** with **a malicious executable**.

Comparing Registry Snapshots with Regshot I

Regshot

- Regshot is an open source registry comparison tool that allows you to take and compare two registry snapshots.
- To use Regshot for malware analysis,
    - simply take the first shot by clicking the **1st Shot button**
    - and then **run the malware** and wait for it to finish making any system changes
    - Next, take the second shot by clicking the **2nd Shot button**.
    - Finally, click the **Compare** button to compare the two snapshots.

Comparing Registry Snapshots with Regshot II

```
Regshot
Comments:
Datetime: <date>
Computer: MALWAREANALYSIS
Username: username

----------------------------------
Keys added: 0
----------------------------------
```

```
----------------------------------
Values added:3
----------------------------------
❶ HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\ckr:C:\WINDOWS\system32\
ckr.exe
...
...

----------------------------------
Values modified:2
----------------------------------
❷ HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed: 00 43 7C 25 9C 68 DE 59 C6 C8
9D C3 1D E6 DC 87 1C 3A C4 E4 D9 0A B1 BA C1 FB 80 EB 83 25 74 C4 C5 E2 2F CE
4E E8 AC C8 49 E8 E8 10 3F 13 F6 A1 72 92 28 8A 01 3A 16 52 86 36 12 3C C7 EB
5F 99 19 1D 80 8C 8E BD 58 3A DB 18 06 3D 14 8F 22 A4
...

----------------------------------
Total changes:5
----------------------------------
```

Faking a Network I

Faking a Network

- Malware often beacons out and eventually communicates with a commandand-control server.
- You can create a fake network and quickly obtain network indicators, without actually connecting to the Internet.
- These indicators can include DNS names, IP addresses, and packet signatures.

Faking a Network II

Using ApateDNS

- ApateDNS, a free tool from Mandiant [2] is the quickest way to see DNS requests made by malware.
- ApateDNS spoofs DNS responses to a user-specified IP address by listening on UDP port 53 on the local machine.
- ApateDNS can display the hexadecimal and ASCII results of all requests it receives.

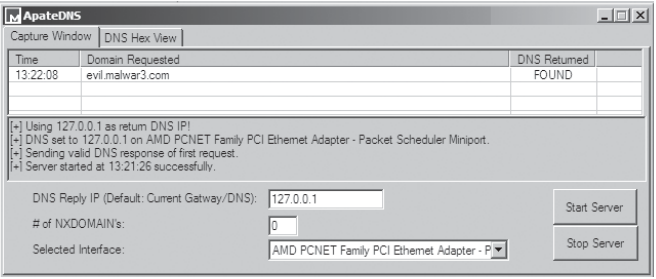


Figure: ApateDNS responding to a request for evil.malwar3.com

Faking a Network III

Monitoring with Netcat

- Netcat, the "*TCP/IP Swiss Army knife*," can be used over both inbound and outbound connections for port scanning, tunneling, proxying, port forwarding, and much more.
- In listen mode, Netcat acts as a server, while in connect mode it acts as a client.
- All the data it receives is output to the screen via standard output.

```
MacBook-Pro:files ozgurcatak$ sudo nc  -l 80
Password:
GET / HTTP/1.1
Host: 127.0.0.1
Upgrade-Insecure-Requests: 1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6) AppleWebKit/604
.3.5 (KHTML, like Gecko) Version/11.0.1 Safari/604.3.5
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

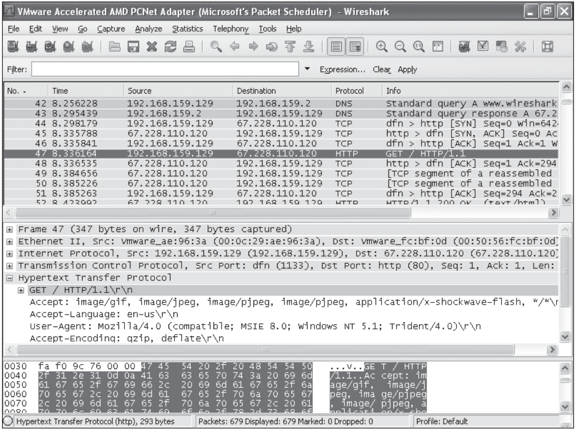
[2] https://www.fireeye.com/services/freeware/apatedns.html

Packet Sniffing with Wireshark I

Wireshark

- Wireshark is an *open source sniffer*, a packet capture tool that intercepts and logs network traffic.
- Wireshark provides *visualization*, *packet-stream analysis*, and *in-depth analysis of individual packets*.

Packet Sniffing with Wireshark II



- To use Wireshark to view the contents of a TCP session, right-click any TCP packet and select **Follow TCP Stream**.

Packet Sniffing with Wireshark III

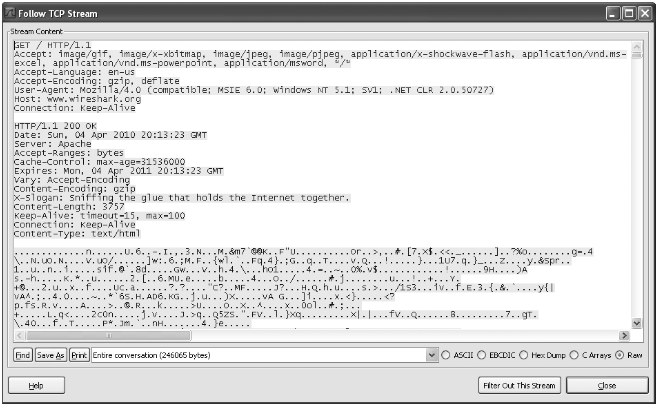


Table of Contents

Assembly Language I

Assembly Language

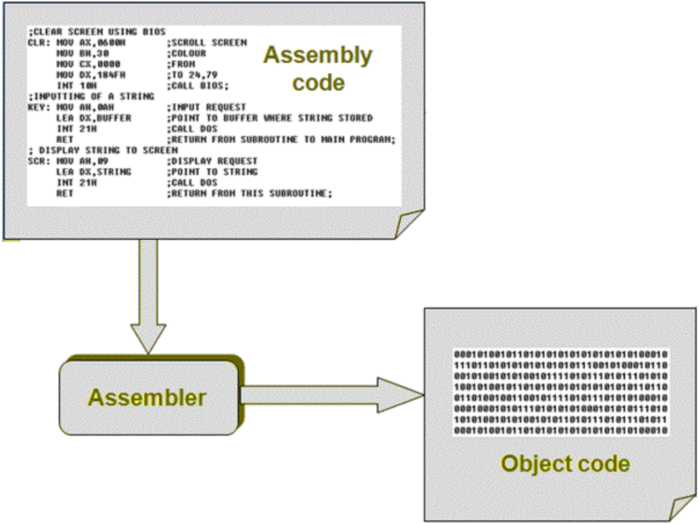
- **An assembly language**, is a **low-level programming language** for a **computer**, or **other programmable device**, in which there is a very strong correspondence **between the language** and **the architecture's machine code instructions**.
- Each assembly language is s**pecific to a particular computer architecture**.
- Assembly language may also be called **symbolic machine code**.

Assembly Language II

Assembler

- Assembly language is converted into executable machine code by a **utility program** referred to as an **assembler**.
- The conversion process is referred to as **assembly**, or **assembling** the source code.
- **Assembly time** is the computational step where an assembler is run.
- The most important feature that **distinguishes an assembler from a compiler** is that it performs an one-by-one transformation.

Assembly Language III



Assembly Language IV

Definitions

Linker A **linker** or **link editor** is a computer program that takes one or more object files generated by a compiler and combines them into a **single executable file**, **library file**, or **another 'object'** file.

Compiler A compiler is computer software that **transforms computer code** written in one programming language (the source language) into **another computer language** (the target language).

- The name **compiler** is primarily used for programs that translate **source code** from a **high-level programming language** to a **lower level language** (e.g., *assembly language*, *object code*, or *machine code*) to create an **executable** program.

Interpreter An **interpreter** is a computer program that directly executes, i.e. *performs*, instructions written in a **programming** or **scripting language**, without requiring them previously to have been compiled into a **machine language program**.

Assembly Language V

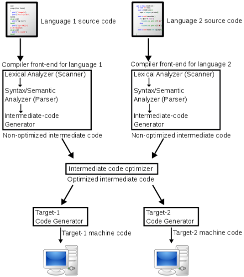


Figure: Compiler

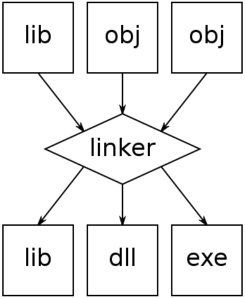


Figure: Linker

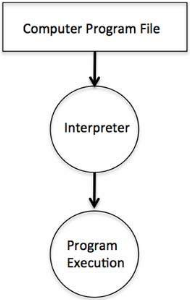


Figure: Interpreter

Assembly Language VI

The Advantages of Assembly

- Speed
- Space
- Capacity
- Algorithm Skill

The Disadvantages of Assembly

- Learning the language
- Readibility
- Development Time
- Sustainability

Development Environment I

Windows

- FASM
- Debug.exe
- Fresh IDE

Linux

- GNU Assembler
- nasm

```
; hello.asm - a "hello, world" program using NASM

section .text

global mystart              ; make the main function externally visible

mystart:

; 1 print "hello, world"

    ; 1a prepare the arguments for the system call to write
    push dword mylen        ; message length
    push dword mymsg        ; message to write
    push dword 1            ; file descriptor value

    ; 1b make the system call to write
    mov eax, 0x4            ; system call number for write
    sub esp, 4              ; OS X (and BSD) system calls needs "extra space" on stack
    int 0x80                ; make the actual system call

    ; 1c clean up the stack
    add esp, 16             ; 3 args * 4 bytes/arg + 4 bytes extra space = 16 bytes

; 2 exit the program

    ; 2a prepare the argument for the sys call to exit
    push dword 0            ; exit status returned to the operating system

    ; 2b make the call to sys call to exit
    mov eax, 0x1            ; system call number for exit
```

Figure: hello-world.asm

Development Environment II

```
nasm -f elf64 -o hello.o hello.asm
ld -o hello hello.o
```

```
section .data
   msg db "Hello World!"

section .text
   global _start
_start:
  mov rax, 1
  mov rdi, 1
  mov rsi, msg
  mov rdx, 13
  syscall

  mov rax, 60
  mov rdi, 0
  syscall
```

Figure: hello-world.asm

Memory Management I

- The a**llocation of the main memory** between **operations** is called **memory management**.
- **The segment of the operating system** created for this purpose is called the **memory manager**.
- **The aims of the memory manager are**
  - to track which parts of the memory are in use
  - what parts are not being used
  - allocate memory to processes
  - recover allocated memory
  - and perform swap operations between memory and disk.

Memory Management II

- **The memory management results of an operating system are**:
  - Any transaction in memory should be able to transfer them to another place.
  - In the case of multiple transactions or users, one user should be prevented from entering the other user's space.
  - It should provide resource sharing between users.
  - It should facilitate access to information for users and operations by ensuring that **the memory is divided into logical areas**.
  - If your memory is not enough, it should be able to use **other physical memory areas**, such as hard disks. (**Virtual Memory**)

Memory Management Basic Definitions

Memory Management Basic Definitions

- Relocation
- Protection
- Sharing
- Logical Organization
- Physical Structuring

Relocation

Relocation

- In **virtual memory systems**, programs in memory need to be available at different times and at different locations in memory.
- The main reason for this is that **it is not always possible to place the program in the same memory area** when it is brought back out of memory after **being taken out of memory for a period of time.**
- For this reason, **the memory management of the operating system** must be able to **relocate programs in memory**, and after this relocation should be able to **correctly point references in the program code** to always point to the **correct location in memory**.

Protection

Protection

- Processes should not make memory references for other processes without the **corresponding process permission**.
- This mechanism, called **memory protection**, prevents **malicious** or **malfunctioning code** in the program from affecting the **operation of other programs**.

Sharing

Sharing

- Although the memory between the different processes is under protection, it should be **possible to access different memory areas** and **share information in different situations in appropriate situations**.

Logical Structuring

Logical Structuring

- Programs are usually configured as modules.
- Some of these modules can be used by other programs in the form of reading only, or changing data.
- Memory management is responsible for the organization of this logical structure, which is different from the physical space.
- One of the methods of achieving this is **segmentation**.

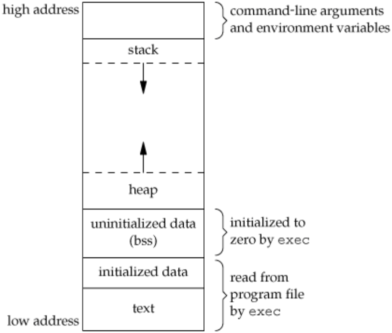
Physical Structuring

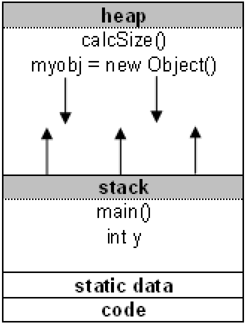
Physical Structuring

- Memory is often partitioned into **fast primary storage** and **slow secondary storage** (such as random access memory RAM and Hard Disk).
- Memory management in operating systems is responsible for moving information between these memory layers.

Memory Management I

- The running code is stored in a special area on the platform.
- The **stack** automatically **wipes** itself after the corresponding section is used. The delete operation in the **heap** is done by the user.
- Any process related to the **Stack** takes less time than the **Heap**.





Memory Management II

- In the case of **Stack** and **Heap overload**, the program to be run does not work correctly.
- Malware can perform attacks taking advantage of such vulnerabilities.
- Various measures can be taken regarding this situation.
  - There is 'stack smashing protection' in the GNU C compiler (GCC).
  - There is a **warning** and **detection system** to be used for any function that may cause this condition.

Stack Overflow

```
void stack_overflow(const char *x)
{
    char y[3];
    strcpy(y, x);
}
```

Heap Overflow

```
void heap_overflow(const char *x)
{
    char *y = malloc(strlen(x));
    strcpy(y, x);
}
```

StackOverflow Lab

Example:	Stackoverflow
Operating System:	Linux, Mac
Language:	C
Compiler:	gcc
Parameters:	-fno-stack-protector -g -D_FORTIFY_SOURCE=0
Link:	`http://www.cse.scu.edu/~tschwarz/coen152_05/Lectures/BufferOverflow.html`

Memory Management

Memory Management

- In the case of stack / heap overload, the attacker can load the command by activating the executable file.
- This **increases the authority** of the malicious software and **causes damage** to the platform.

Table of Contents

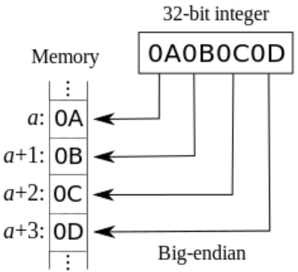
Processor Architectures

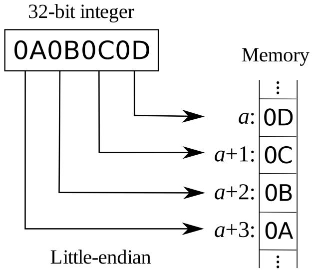
- The malware is stored on the disk in **binary format** (Machine Code).
- When an malicious software is resolved, there is an output format in assembly format.
- The assembly form also depends on the **type of hardware** the program is running.
- **Instruction sets**, **register lengths** are some of the changing parameters.
- Some common processor architectural families are as follows:
  - MIPS
  - x86
  - Power PC
  - ARM
  - SPARC

Big Endian vs Little Endian

Big & Little Endian

- **Problem: Computers speak different languages, like people**. Some write data *left-to-right* and others *right-to-left*.
  - A machine can read its own data just fine - problems happen when one computer stores data and a different type tries to read it.
- **Solutions** :
  - Agree to a common format (i.e., all network traffic follows a single format), or
  - Always include a header that describes the format of the data. If the header appears backwards, it means data was stored in the other format and needs to be converted.





PowerPC

PowerPC

- PowerPC is a RISC microprocessor introduced by the Apple-IBM-Motorola partnership in 1991, known as AIM.
- Generally for personal computers.
- PowerPC central processing units (CPUs) are popular because they are embedded and high performance processors.
- Big Endian
- 32-64 bit

ARM

ARM

- ARM architecture (original name Acorn RISC Machine) is a RISC-based processor architecture.
- 32-64 bit
- Because **low power consumption**, **higher performance** than other RISC-based processors and it is more **cost-effective than x86-x64 processors**, ARM processors are generally preferred for **embedded systems** and chipsets used in p**ortable devices**.
- Big Endian

Sparc

Sparc

- SPARC (Scalable Processor ARChitecture) is a processor architecture and family operating with the RISC method.
- Designed by Sun Microsystems in 1985.
- In 2006, Sun released an extended architecture called UltraSPARC, compatible with SPARC V9.
- In March 2006, all source code for the processor design was published under the OpenSPARC project.
- Big Endian
- 32-64 bit

MIPS

MIPS

- MIPS is a reduced instruction set type microprocessor architecture developed by the company MIPS Technologies in 1985.
- **Each command is the same size** and the **command can be easily solved by the computer hardware**.
- Because of the RISC structure, the system is based on **improving simple operations** that are often done to create structures that support complex operations.
- Because of its **simple** and **robust** design, most modern microprocessor architectures (PowerPC, ARM) are **inspired by the MIPS architecture**.
- Bi Endian
- 32-64 bit

x86 I

x86

- Intel x86 is a complex command-line computer.
- The sizes of the commands are different and microcode is required to solve the commands.
- x86 is a description of the software rules for the 8086, one of Intel's first microprocessors.
- One of Intel's key features, "backward compatibility" has made such a definition.
- Any assembly software you create on a computer system with an 8086 microprocessor will run on all X86 compatible computers.
- Little Endian
- 16-32-64 bit

x86 II

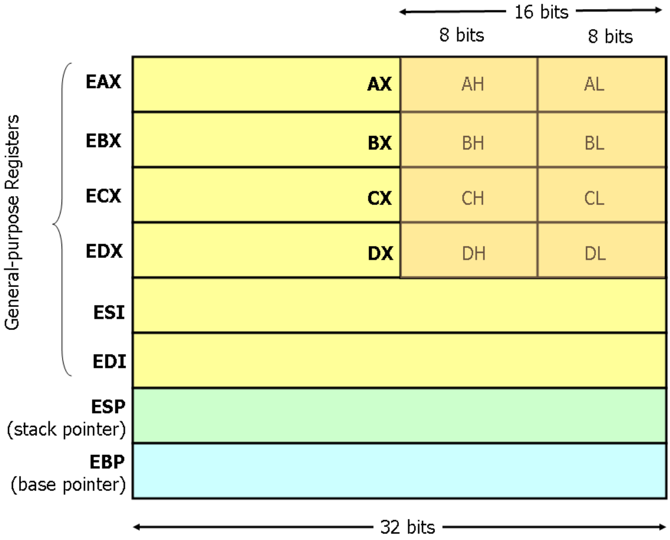
x86 Registers

- 8 general purpose registers (GPR), 6 segment registers, 1 flag writer and directive markers.

General Purpose Registers

- EAX: The arithmetic operations are performed.
- ECX: Counters writer. They are used in the shift and rotate directives.
- EDX: Register where the variable values are held. They are used for input / output operations and arithmetic operations.
- EBX: Basic register. They are used to represent the values. (They are found in the DS section.)
- ESP: Stack Pointer. It shows the top of the stack.
- EBP: Stack Base Pointer. Shows the bottom of the stack.
- ESI: Source writer. They are used as indicators in flow processes.
- EDI: The target writer. They are used as indicators in flow processes.

x86 III


16 bits
8 bits
8 bits
General-purpose Registers
EAX
AX
AH
AL
EBX
BX
BH
BL
ECX
CX
CH
CL
EDX
DX
DH
DL
ESI
EDI
ESP
(stack pointer)
EBP
(base pointer)
32 bits

x86 IV

- Each GPR is 32 bits wide.
- 16 least significant bits (LSBs) are named AX, CX, DX, BX, SP, BP, SI and DI as name references.
- These parts are unexpanded parts.

Segment Registers

The related 6 segment registers are as follows.

- **SS**: Stack Segment
- **CS**: Code Segment
- **DS**: Data Segment
- **ES**: Extra Segment
- **FS**: F Segment
- **GS**: G Segment

In modern operating systems, the memory model is located in the same partition for all segment registers. (Linux, Windows)

x86 V

Flags Registers

- **CF**: Carry Flag, takes 1 if there is an increasing value when it is a collection.
- **PF**: Parity Flag, if the LSB value is a multiple of 2, it is 1.
- **ZF**: Zero Flag. If the result of a sum is 0, it is set.
- **IF**: Interrupt Flag. If the interrupts are active, it takes value 1.
- **OF**: Overflow Flag. If the value in the register is too large for the size of the register, this flag is set.

x86 VI

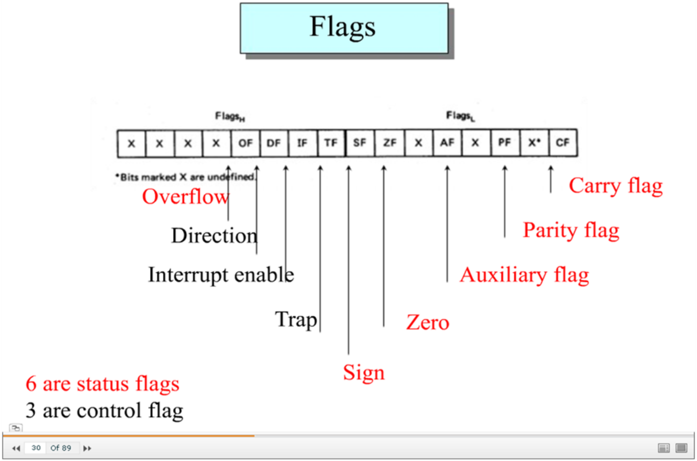


Figure: Flag Register

Table of Contents

Memory and Addressing Modes I

Declaring Static Data Regions

- You can declare static data regions (analogous to global variables) in x86 assembly using special assembler directives for this purpose.
- Data declarations should be preceded by the .DATA directive. Following this directive, the directives **DB**, **DW**, and **DD** can be used to declare **one**, **two**, and **four** byte data locations, respectively.
- Declared locations can be labeled with names for later reference — this is similar to declaring variables by name, but abides by some lower level rules. For example, locations declared in sequence will be located in memory next to one another.

```
.DATA
var     DB 64 ; Declare a byte, referred to as location var,
                ; containing the value 64.
var2    DB ? ; Declare an uninitialized byte, referred to as location var2.
DB 10 ; Declare a byte with no label, containing the value 10.
        ; Its location is var2 + 1.
X       DW ? ; Declare a 2-byte uninitialized value, referred to as location X.
Y       DD 30000 ; Declare a 4-byte value, referred to as location Y,
              ; initialized to 30000.
```

Memory and Addressing Modes II

- Unlike in high level languages where arrays can have many dimensions and are accessed by indices, arrays in x86 assembly language are simply a number of cells located contiguously in memory.
- An array can be declared by just listing the values, as in the first example below.
- Two other common methods used for declaring arrays of data are the DUP directive and the use of string literals.
- The **DUP** directive tells the assembler to duplicate an expression a given number of times. For example, 4 **DUP**(2) is equivalent to 2, 2, 2, 2.

```
Z       DD 1, 2, 3      ; Declare three 4-byte values, initialized to 1, 2,
                         ; and 3. The value of location Z + 8 will be 3.
bytes   DB 10 DUP(?)    ; Declare 10 uninitialized bytes starting
                            ;at location bytes.
arr     DD 100 DUP(0) ; Declare 100 4-byte words starting at location arr,
                        ; all initialized to 0
str     DB 'hello',0  ; Declare 6 bytes starting at the address str,
                    ; initialized to the ASCII character values for hello
                    ; and the null (0) byte.
```

Memory and Addressing Modes III

Addressing Memory

- Modern x86-compatible processors are capable of addressing up to 2^{32} bytes of memory: memory addresses are 32-bits wide.
- In the examples above, where we used labels to refer to memory regions, these labels are actually replaced by the assembler with 32-bit quantities that specify addresses in memory.
- Some examples using the *mov* instruction that moves data between registers and memory. **This instruction has two operands**: the first is the **destination** and the second specifies the **source**.

```
mov eax, [ebx]  ; Move the 4 bytes in memory at the address contained
                          ; in EBX into EAX
mov [var], ebx  ; Move the contents of EBX into the 4 bytes at memory
                          ; address var. (Note, var is a 32-bit constant).
mov eax, [esi-4] ; Move 4 bytes at memory address ESI + (-4) into EAX
mov [esi+eax], cl ; Move the contents of CL into the byte at address ESI+EAX
mov edx, [esi+4*ebx] ; Move the 4 bytes of data at address ESI+4*EBX into EDX
```

Some examples of invalid address calculations include:

```
mov eax, [ebx-ecx]      ; Can only add register values
mov [eax+esi+edi], ebx          ; At most 2 registers in address computation
```

Memory and Addressing Modes IV

Size Directives

- However, in some cases the size of a referred-to memory region is ambiguous. Consider the instruction *mov [ebx], 2*.
- Should this instruction move the value 2 into the single byte at address EBX? Perhaps it should move the 32-bit integer representation of 2 into the 4-bytes starting at address EBX.
- Since either is a valid possible interpretation, the assembler must be explicitly directed as to which is correct.
- The size directives *BYTE PTR*, *WORD PTR*, and *DWORD PTR* serve this purpose, indicating sizes of 1, 2, and 4 bytes respectively.

```
mov BYTE PTR [ebx], 2    ; Move 2 into the single byte at the address
                         ; stored in EBX.
mov WORD PTR [ebx], 2    ; Move the 16-bit integer representation of 2 into
                         ; the 2 bytes starting at the address in EBX.
mov DWORD PTR [ebx], 2   ; Move the 32-bit integer representation of 2
                         ; into the 4 bytes starting at the address in EBX.
```

Instructions I

Instructions

- Machine instructions generally fall into three categories:
  - data movement
  - arithmetic/logic
  - control-flow

Data Movement Instructions I

mov - Move

- The *mov* instruction copies the data item referred to by **its second operand** (i.e. register contents, memory contents, or a constant value) into the location referred to by **its first operand** (i.e. a register or memory).
- Synthax

```
mov <reg>,<reg>
mov <reg>,<mem>
mov <mem>,<reg>
mov <reg>,<const>
mov <mem>,<const>
```

- Examples

```
mov eax, ebx ; copy the value in ebx into eax
mov byte ptr [var], 5 ; store the value 5 into the byte at location var
```

Data Movement Instructions II

push - Push stack

- The *push* instruction places its operand onto the top of the hardware supported stack in memory.
- Specifically, push first decrements *ESP* by 4, then places its operand into the contents of the 32-bit location at address *[ESP]*.
- ESP (the stack pointer) is decremented by push since the x86 stack grows down - i.e. the stack grows from high addresses to lower addresses.
- *Synthax*

```
push <reg32>
push <mem>
push <con32>
```

- Examples

```
push eax ; push eax on the stack
push [var] ; push the 4 bytes at address var onto the stack
```

Data Movement Instructions III

pop - Pop stack

- The *pop* instruction removes the 4-byte data element from the top of the hardware-supported stack into the specified operand (i.e. register or memory location).
- It first moves the 4 bytes located at memory location *[SP]* into the specified register or memory location, and then increments *SP* by 4.
- syntax

```
pop <reg32>
pop <mem>
```

- Examples

```
pop edi ; pop the top element of the stack into EDI.
pop [ebx] ; pop the top element of the stack into memory at the 4 bytes star
```

Data Movement Instructions IV

lea — Load effective address

- The *lea* instruction places the *address* specified by its second operand into the register specified by its first operand.
- Note, the *contents* of the memory location are not loaded, only the effective address is computed and placed into the register.
- This is useful for obtaining a pointer into a memory region.
- Syntax

```
lea <reg32>,<mem>
```

- Examples

```
lea edi, [ebx+4*esi] ; the quantity EBX+4*ESI is placed in EDI.
lea eax, [var] ; the value in var is placed in EAX.
lea eax, [val] ; the value val is placed in EAX.
```

Arithmetic and Logic Instructions I

add - Integer Addition

- The *add* instruction adds together its two operands, storing the result in its first operand. Note, whereas both operands may be registers, at most one operand may be a memory location.
- Syntax

```
add <reg>,<reg>
add <reg>,<mem>
add <mem>,<reg>
add <reg>,<con>
add <mem>,<con>
```

- Examples

```
add eax, 10 ; EAX = EAX + 10
add BYTE PTR [var], 10 ; add 10 to the single byte
                       ; stored at memory address var
```

Arithmetic and Logic Instructions II

sub - Integer Subtraction

- The *sub* instruction stores in the value of its first operand the result of subtracting the value of its second operand from the value of its first operand. As with *add*
- Syntax

```
sub <reg>,<reg>
sub <reg>,<mem>
sub <mem>,<reg>
sub <reg>,<con>
sub <mem>,<con>
```

- Examples

```
sub al, ah ; AL = AL - AH
sub eax, 216 ; subtract 216 from the value stored in EAX
```

Arithmetic and Logic Instructions III

inc, dec — Increment, Decrement

- The *inc* instruction increments the contents of its operand by one. The *dec* instruction decrements the contents of its operand by one.
- Syntax

```
inc <reg>
inc <mem>
dec <reg>
dec <mem>
```

- Examples

```
dec eax ; subtract one from the contents of EAX.
inc DWORD PTR [var] ; add one to the 32-bit integer stored at location var
```

Arithmetic and Logic Instructions IV

imul — Integer Multiplication

- The *imul* instruction has two basic formats: two-operand and three-operand.
- The two-operand: multiplies its two operands together and stores the result in the first operand. The result (i.e. first) operand must be a register.
- The three operand: multiplies its second and third operands together and stores the result in its first operand. Again, the result operand must be a register. Furthermore, the third operand is restricted to being a constant value.
- Syntax

```
imul <reg32>,<reg32>
imul <reg32>,<mem>
imul <reg32>,<reg32>,<con>
imul <reg32>,<mem>,<con>
```

- Examples

```
imul eax, [var] ; multiply the contents of EAX by the 32-bit contents
                ; of the memory location var. Store the result in EAX.
imul esi, edi, 25 ; ESI -> EDI * 25
```

Arithmetic and Logic Instructions V

idiv — Integer Division

- The *idiv* instruction divides the contents of the 64 bit integer EDX:EAX (constructed by viewing EDX as the most significant four bytes and EAX as the least significant four bytes) by the specified operand value. **The quotient result of the division is stored into EAX, while the remainder is placed in EDX**.
- Syntax

```
idiv <reg32>
idiv <mem>
```

- Examples

```
idiv ebx ; divide the contents of EDX:EAX by the contents of EBX.
         ; Place the quotient in EAX and the remainder in EDX.
idiv DWORD PTR [var] ; divide the contents of EDX:EAX by the 32-bit
                     ; value stored at memory location var. Place the
                     ; quotient in EAX and the remainder in EDX.
```

Arithmetic and Logic Instructions VI

and, or, xor - Bitwise logical and, or and exclusive or

- These instructions perform the specified logical operation (logical bitwise and, or, and exclusive or, respectively) on their operands, placing the result in the first operand location.
- Syntax

```
and <reg>,<reg>
and <reg>,<mem>
and <mem>,<reg>
and <reg>,<con>
and <mem>,<con>

or ...

xor ...
```

- Examples

```
and eax, 0fH ; clear all but the last 4 bits of EAX.
xor edx, edx ; set the contents of EDX to zero.
```

Arithmetic and Logic Instructions VII

not - Bitwise Logical Not

- Logically negates the operand contents (that is, flips all bit values in the operand).
- Syntax

```
not <reg>
not <mem>
```

- Examples

```
not BYTE PTR [var] ; negate all bits in the byte at the memory location var.
```

Arithmetic and Logic Instructions VIII

neg — Negate

- Performs the two's complement negation of the operand contents.
- Syntax

```
neg <reg>
neg <mem>
```

- Examples

```
neg eax ; EAX -> -EAX
```

Arithmetic and Logic Instructions IX

shl, shr - Shift Left, Shift Right

- These instructions shift the bits in their first operand's contents left and right, padding the resulting empty bit positions with zeros.
- The shifted operand can be shifted up to 31 places.
- The number of bits to shift is specified by the second operand, which can be either an 8-bit constant or the register CL.
- In either case, shifts counts of greater then 31 are performed modulo 32.
- Syntax

```
shl <reg>,<con8>
shl <mem>,<con8>
shl <reg>,<cl>
shl <mem>,<cl>

shr ...
```

- Examples

```
shl eax, 1 ; Multiply the value of EAX by 2
           ; (if the most significant bit is 0)
shr ebx, cl ; Store in EBX the floor of result of dividing the value
           ; of EBX by 2^n wheren is the value in CL.
```

Arithmetic and Logic Instructions X

jmp - Jump

- Transfers program control flow to the instruction at the memory location indicated by the operand.
- Syntax

```
jmp <label>
```

- Examples

```
jmp begin ; Jump to the instruction labeled begin.
```

Arithmetic and Logic Instructions XI

jcondition - Conditional Jump

- These instructions are conditional jumps that are based on the status of a set of condition codes that are stored in a special register called the machine status word.
- A number of the conditional branches are given names that are intuitively based on the last operation performed being a special compare instruction, *cmp* . For example, conditional branches such as *jle* and *jne* are based on first performing a *cmp* operation on the desired operands.
- Syntax

```
je <label> (jump when equal)
jne <label> (jump when not equal)
jz <label> (jump when last result was zero)
jg <label> (jump when greater than)
jge <label> (jump when greater than or equal to)
jl <label> (jump when less than)
jle <label> (jump when less than or equal to)
```

- Examples

```
cmp eax, ebx
jle done; If the contents of EAX are less than or equal to
        ; the contents of EBX, jump to the label done.
        ; Otherwise, continue to the next instruction.
```

Arithmetic and Logic Instructions XII

cmp - Compare

- Compare the values of the two specified operands, setting the condition codes in the machine status word appropriately.
- This instruction is equivalent to the *sub* instruction, except the result of the subtraction is discarded instead of replacing the first operand.
- Syntax

```
cmp <reg>,<reg>
cmp <reg>,<mem>
cmp <mem>,<reg>
cmp <reg>,<con>
```

- Examples

```
cmp DWORD PTR [var], 10 ; If the 4 bytes stored at location var are
        ; equal to the 4-byte integer constant 10,
        ; jump to the location labeled loop.
jeq loop
```

Arithmetic and Logic Instructions XIII

call, ret - Subroutine call and return

- The *call* instruction first pushes the current code location onto the hardware supported stack in memory (see the *push* instruction for details), and then performs an unconditional jump to the code location indicated by the label operand.
- Unlike the simple jump instructions, the call instruction saves the location to return to when the subroutine completes.
- Syntax

```
call <label>
ret
```