CY 411 Reverse Software Engineering

Overview Reverse Engineering

Dr. Qasem Abu Al-Haija

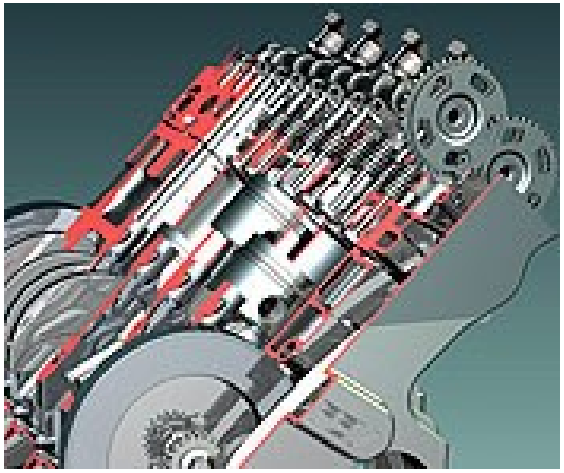
Department of Cybersecurity
Faculty of Computer & Information Technology
Jordan University of Science and Technology

Reverse Engineering

- Process of analyzing a subject system to create representations of the system at a higher level of abstraction”
- “Going backward through the development cycle.”
- Discovering how a device usually works by taking it apart.
- Generally considered lawful if the system was obtained legitimately.

REing Mechanical Devices

- Not what you may think.
- Actually the reverse of the engineering process, going from a finished product to design.
- Used to “digitize” old parts and systems.

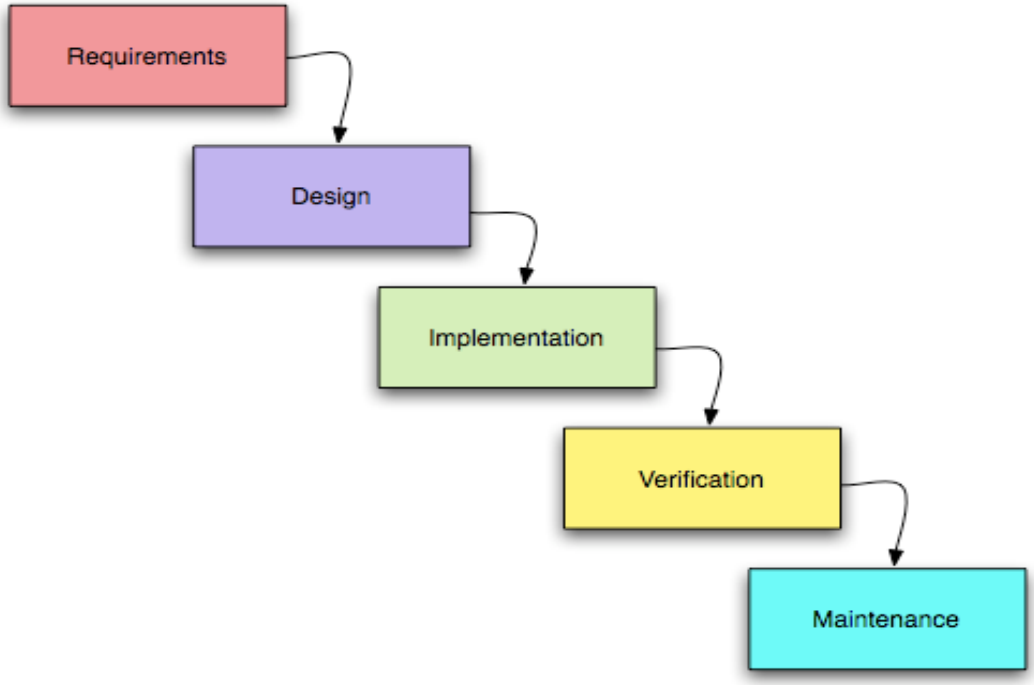


Antikythera mechanism

- A famous example of reverse engineering
- Ancient mechanical computer
- Discovered in a wreck in 1900, dated around 150–100 BC

Development Cycle

- The waterfall model



- Reverse Engineering moves through this process in reverse.

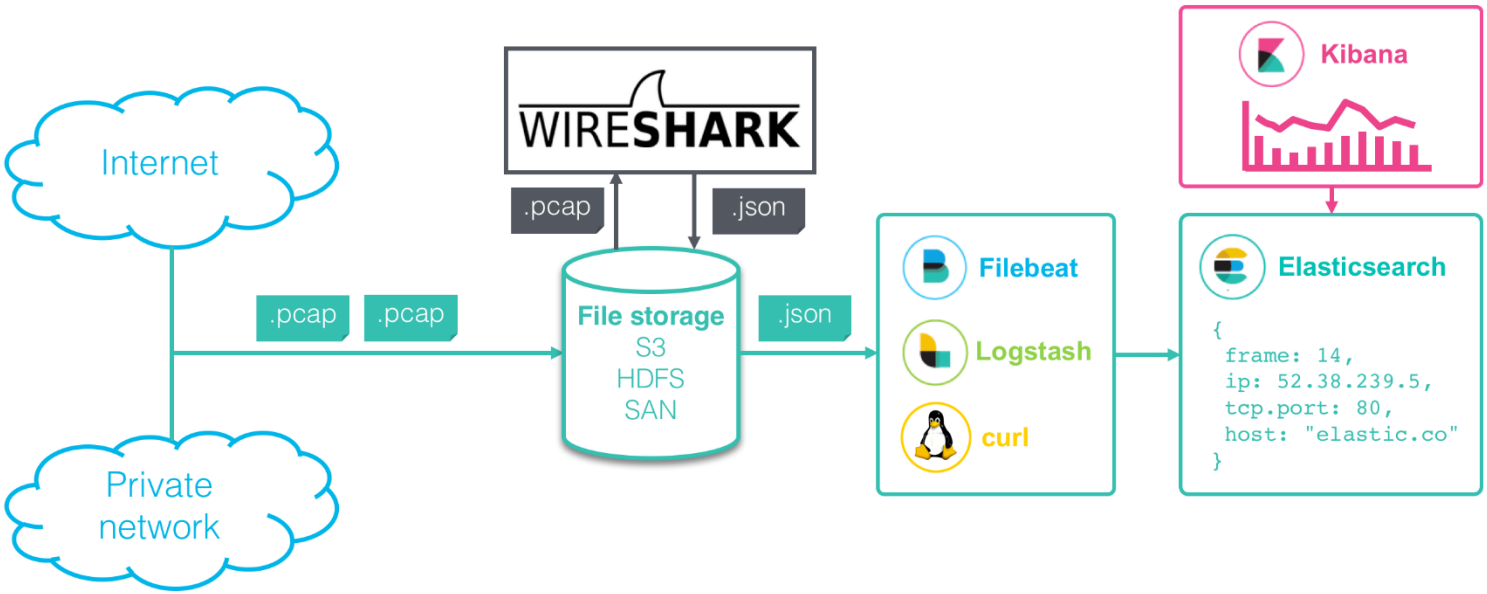
- May not end up with the same implementation.

Software Techniques

- Analysis through observation of information exchange
- Disassembly
- Decompilation

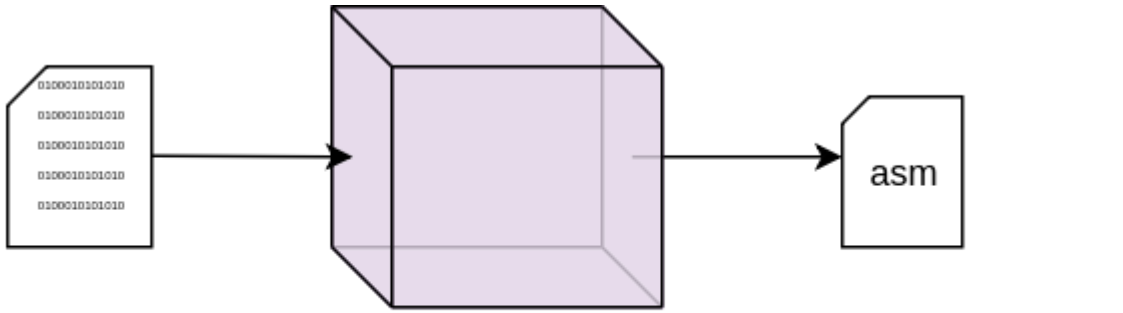
Analysis Through Observation

- Very common for protocol reverse engineering.
- Usually use a bus analyzer and or packet sniffers.
- Can be assisted through the use of low-level debuggers
- Example of tools: SoftICE, WireShark, ...



Disassembly

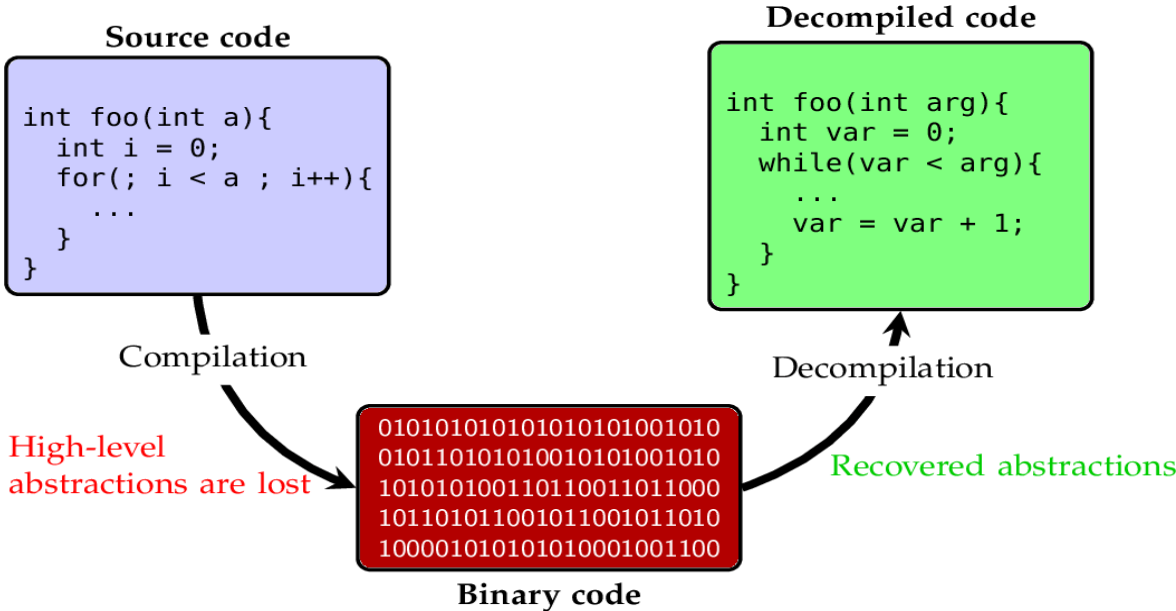
- Most programs, when compiled, are turned into architecture-specific machine code.
- Disassemblers take the binary executable and display its assembly code.
- Need a good understanding of assembly and usually a hex editor.
- Example of tools: W32Dasm, IDA Pro, ...



Decompilation

- A decompiler is a computer program that translates an executable file to a high-level source file that can be recompiled successfully.
- It is the opposite of a typical compiler, which translates a high-level language to a low-level language.
- Example of tools:

  Mocha, JAD,...



Motivations of Reverse Engineering

Motivation of RE

- Interoperability
- Lost documentation
- Product analysis
- Security auditing
- Removal of access restrictions
- Creation of duplicates
- Fraud

Interoperability

- Getting a device/piece of software to work on another platform.

- Example: Reversing systems developed for windows to work over Unix environment

Lost Documentation

- Need to re-learn how the device operates, how the device communicates

- Usually only done on antiquated devices or integrated circuits

Product Analysis

- To determine how the product works
- Can be used to estimate product costs
- Check product legalities: Determine if a product infringes on patent rights.

Security Auditing

- An audit determines if systems safeguard assets, maintain data integrity, and operate effectively.
- The company usually knows about its own products.
- Used to evaluate the risk of new products it may create or use from other companies.

Access Restriction Removal

- Possible legal issues
- Usually done to demo programs, the full version released as warez
- Sometimes, it becomes legal when a program or game becomes very old.

Create Duplicates

- This can be very difficult, trying to reproduce the entire system.
- Reverse engineering of copy restrictions on CDs and other media.
- In certain cases, the user is allowed a duplicate.

Fraud

- Any system (usually embedded or integrated) that stores critical information
- Most common example is credit cards / smart cards
- Passwords and other information are often stored on the card

Reverse Engineering Tools of Software Systems

Topics

- Basic background on assembly language
- Types of reverse engineering tools and demonstrations of these tools:
  - **Hex editors**: WinHex, Tsearch
  - **Decompilers**: REC, DJ
  - **Disassemblers/Debuggers**: IDAPro, OllyDbg, Win32Dasm, BORG

Program Abstractions

Computers understand binary code

Binary code can be written in hexadecimal

Hexadecimal code can be encoded in assembly language

Assembly language is human-readable but not as intuitive as source code

Decompilers convert assembly into an easier-to-read source code

11001111 10101 == CD21 == int 21

Assembly language is an abstraction of hexadecimal code

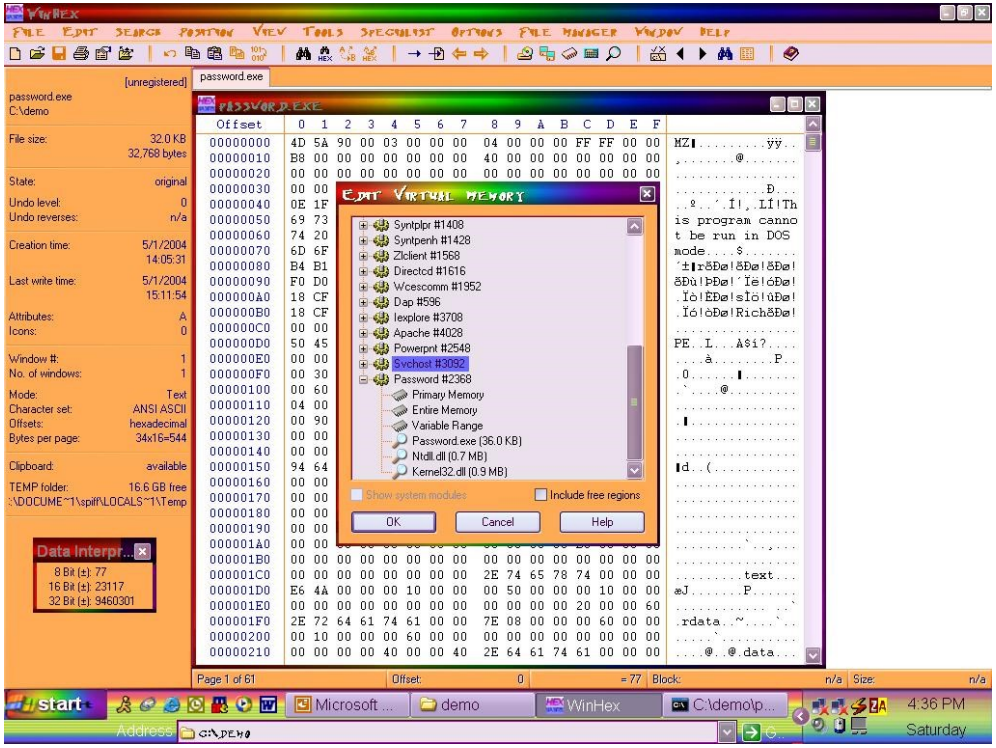
```
C:\>debug
-a
0B0C:0100 mov ah,9
0B0C:0102 mov dx,109
0B0C:0105 int 21
0B0C:0107 int 20
0B0C:0109 db "Hello World$"
0B0C:0115
-nhello.com
-rcx
CX 0000
:115
-w
Writing 00115 bytes
-q

C:\>hello
Hello World
C:\>debug hello.com
-d
0B69:0100  B4 09 BA 09 01 CD 21 CD-20 48 65 6C 6C 6F 20 57   ......!. Hello W
0B69:0110  6F 72 6C 64 24 75 30 80-CF 02 80 3E 34 00 FB 0A   orld$u0....>4...
0B69:0120  04 EB 24 EB 78 B4 07 80-3E 35 99 00 74 02 B4 02   ..$.x...>5..t...
0B69:0130  B0 3F 2A 26 34 99 72 EB-86 E1 E3 09 86 E1 E8 C8   .?*&4.r.........
0B69:0140  00 86 E1 E2 F7 86 E1 E8-15 E3 75 21 80 CF 04 80   ..........u!....
0B69:0150  3E 6E 99 00 74 05 F6 C7-02 75 48 89 3E 32 99 FF   >n..t....uH.>2..
0B69:0160  06 32 99 C6 06 34 99 FF-C6 06 35 99 00 E8 99 00   .2...4....5.....
0B69:0170  AC E8 61 E2 74 38 3C 0D-74 34 3A 06 02 96 74 2E   ..a.t8<.t4:...t.
-u
0B69:0100 B409          MOV     AH,09
0B69:0102 BA0901        MOV     DX,0109
0B69:0105 CD21          INT     21
0B69:0107 CD20          INT     20
0B69:0109 48            DEC     AX
0B69:010A 65            DB      65
0B69:010B 6C            DB      6C
0B69:010C 6C            DB      6C
0B69:010D 6F            DB      6F
0B69:010E 20576F        AND     [BX+6F],DL
0B69:0111 726C          JB      017F
0B69:0113 64            DB      64
0B69:0114 2475          AND     AL,75
0B69:0116 3080CF02      XOR     [BX+SI+02CF],AL
0B69:011A 803E3400FB    CMP     BYTE PTR [0034],FB
0B69:011F 0A04          OR      AL,[SI]
-_
```

Hex Editors

- Hex editors read executing programs from RAM.
- Display their contents in hexadecimal code.
- Enable the editing of the running hexadecimal code.

Example: WinHex

(http://www.sf-soft.de/)



Decompilers

- Decompile binary programs into readable source code.
- Replace all binary code that could not be decompiled with assembly code.

Example: REC

(http://www.backerstreet.com/rec)

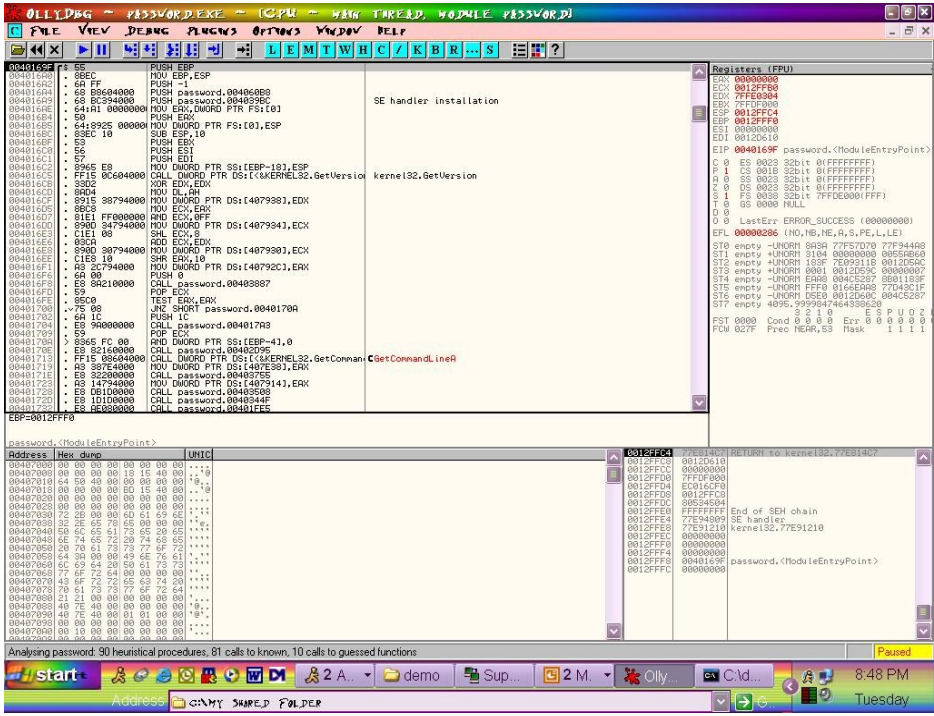
- Decompiles a program from binary code to C pseudo-code.
- Translates any binary it cannot decompile into assembly code.
- Typically generates about 60-70% of the program source code.

Disassemblers/Debuggers

- Convert binary code into its assembly equivalent.
- Extract ASCII strings and used libraries.
- View memory, stack, and CPU registers.
- Run the program (with breakpoints).
- Edit the assembly code at runtime.

Example: OllyDbg

http://home.t-online.de/home/Ollydbg/



Disassemblers/Debuggers Programs & Features chart

Product	Dis-Assembly	Processor options	Debugger	String extraction	Disk Hex editor	Memory Hex editor	Memory Dumper	Library's used	Decryptor
IDAPro	x	x		x	x			x	
OllyDbg	x	x	x	x	x	x	x	x	
W32Dasm	x	x	x	x	x	x	x	x	
BORG	x	x					x		x

Dr. Qasem Abu Al-

Reverse Engineering Prevention Tools

"Code Obfuscators"

Such as Y0da's Cryptor, NFO

Code Obfuscation

- The process of modifying an executable so that it is no longer useful to a hacker but remains fully functional.
  - Modify actual method instructions or metadata
  - Does not alter the program's output.

- However, with enough time and effort, almost all code can be reverse-engineered.

- The goal is to distract the reader with the complicated syntax of what they are reading and make it difficult for them to determine the true content of the message.

Code Obfuscation can be done in several ways.

❑ Example#1: Rename Obfuscation

- ➢ Use naming that make the code difficult for the reader to understand.

Original Source Code Before Rename Obfuscation

```
private void
CalculatePayroll (SpecialList employee-
Group) {
    while (employeeGroup.HasMore()) {
        employee =
employeeGroup.GetNext(true) ;
        employee.UpdateSalary() ;
        Distribute Check(employee) ;
    }
}
```

Reverse-Engineered Source Code After Rename Obfuscation

```
private void a(a b) {
    while (b.a()) {
        a = b.a(true);
        a.a () ;
        a.(a) ;
    }
}
```

❑ Example#2: String Encryptions

- ➢ Encrypting the code of a program so you cannot view it in assembly.

Original Source Code Before String Encryption

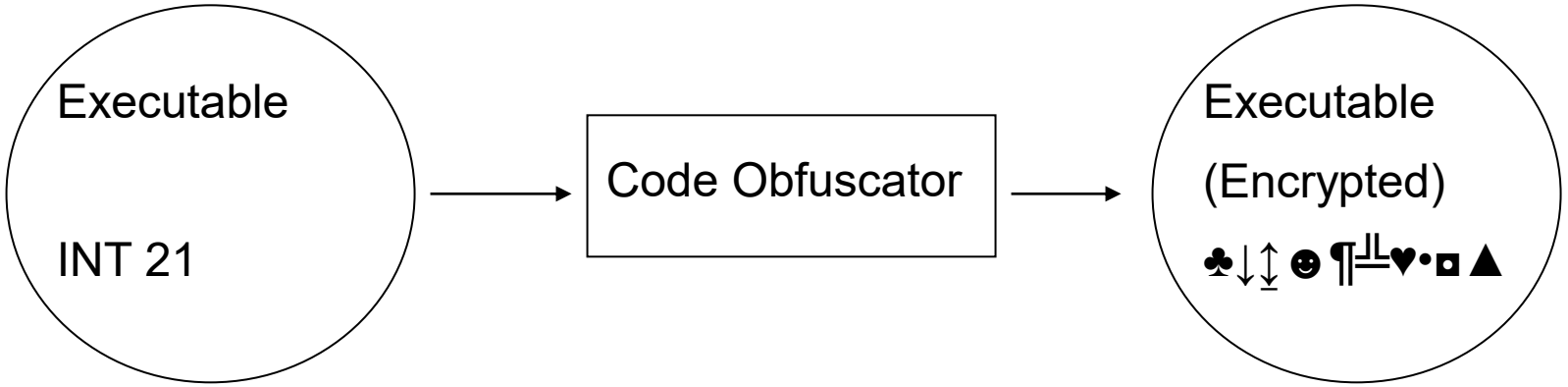
```
...
MessageBox.show("Invalid
Authentication - Try Again")
...
```

Reverse-Engineered Source Code After String Encryption

```
...
MessageBox.show(a.b("¥∑¶§ƒжњж•₡") )
...
```

Code Obfuscators/Encryption tools

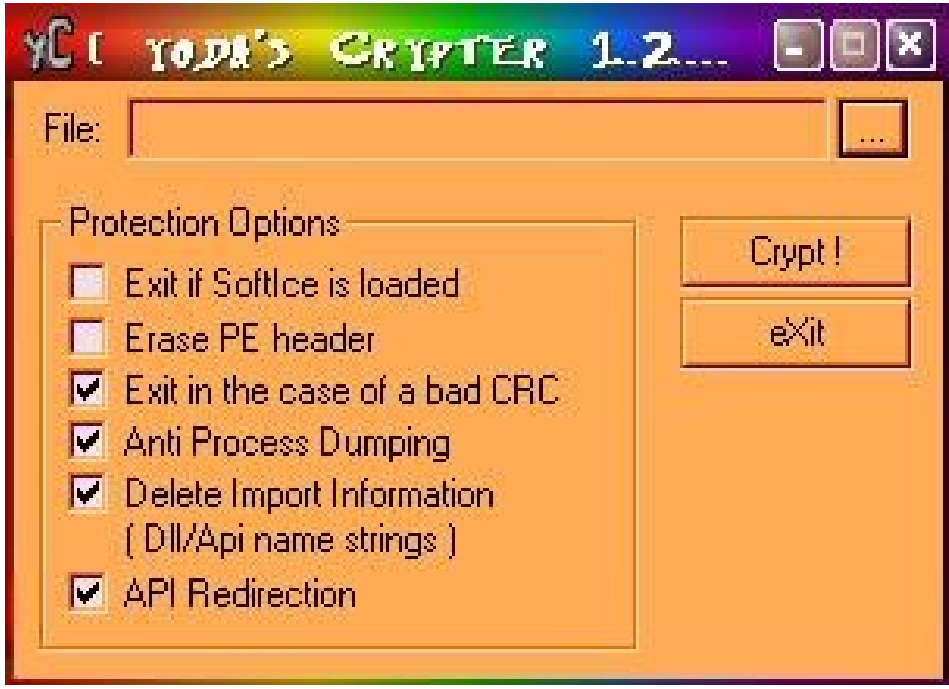
- Encrypts the code of a program so you cannot view it in assembly.



Obfuscators	Obfuscation	Anti-debugging techniques	GUI
Y0da's Cryptor	x	x	x
NFO	x	x	

Example: YOda's Cryptor

- Code Obfuscation.
  - Encrypts the code of a program.
- Anti-Debugging.
  - Detects all major debuggers and disassemblers.
- GUI platform.
  - Graphical user interface.

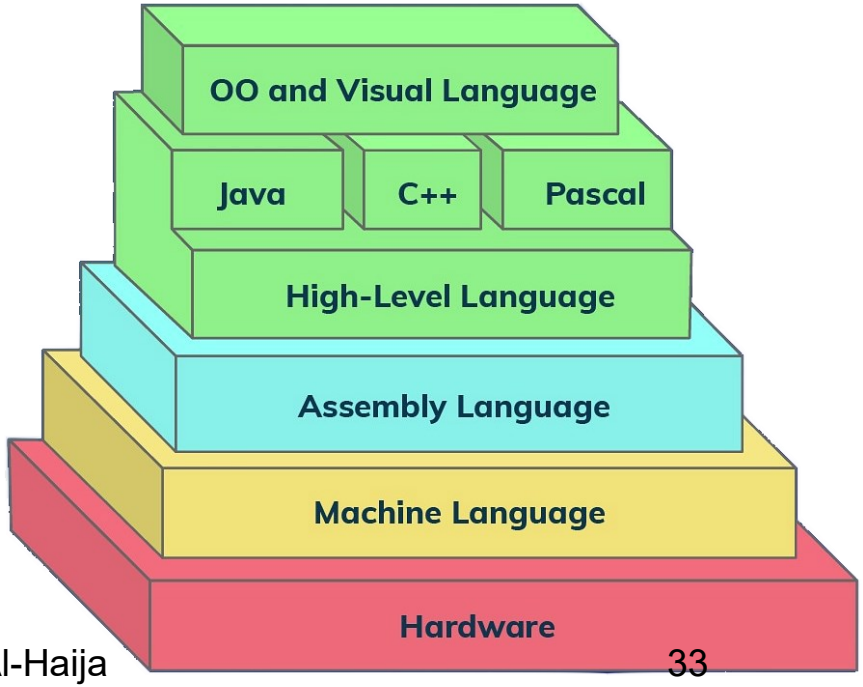


y0da.cjb.net

Assemble: Converting Assembly Language Code To Machine Language Code

Assembly Programming

- Machine Language
  - binary
  - hexadecimal
  - machine code or object code
- Assembly Language
  - mnemonics
  - assembler
- High-Level Language
  - Pascal, Basic, C
  - compiler



Why Assemble for Cybersecurity Experts?

Understanding assembly code is so important in Code interpretation.

Irrespective of the type of high-level language being used, it must first be translated into assembly language before the code gets translated to machine code. This makes assembly language still important despite the evolution of high-level languages.

Understanding assembly code is so important in Control System Resources.

It helps in taking complete control over the system and its resources. By learning assembly language, the programmer can write the code to access registers and retrieve the memory address of pointers and values.

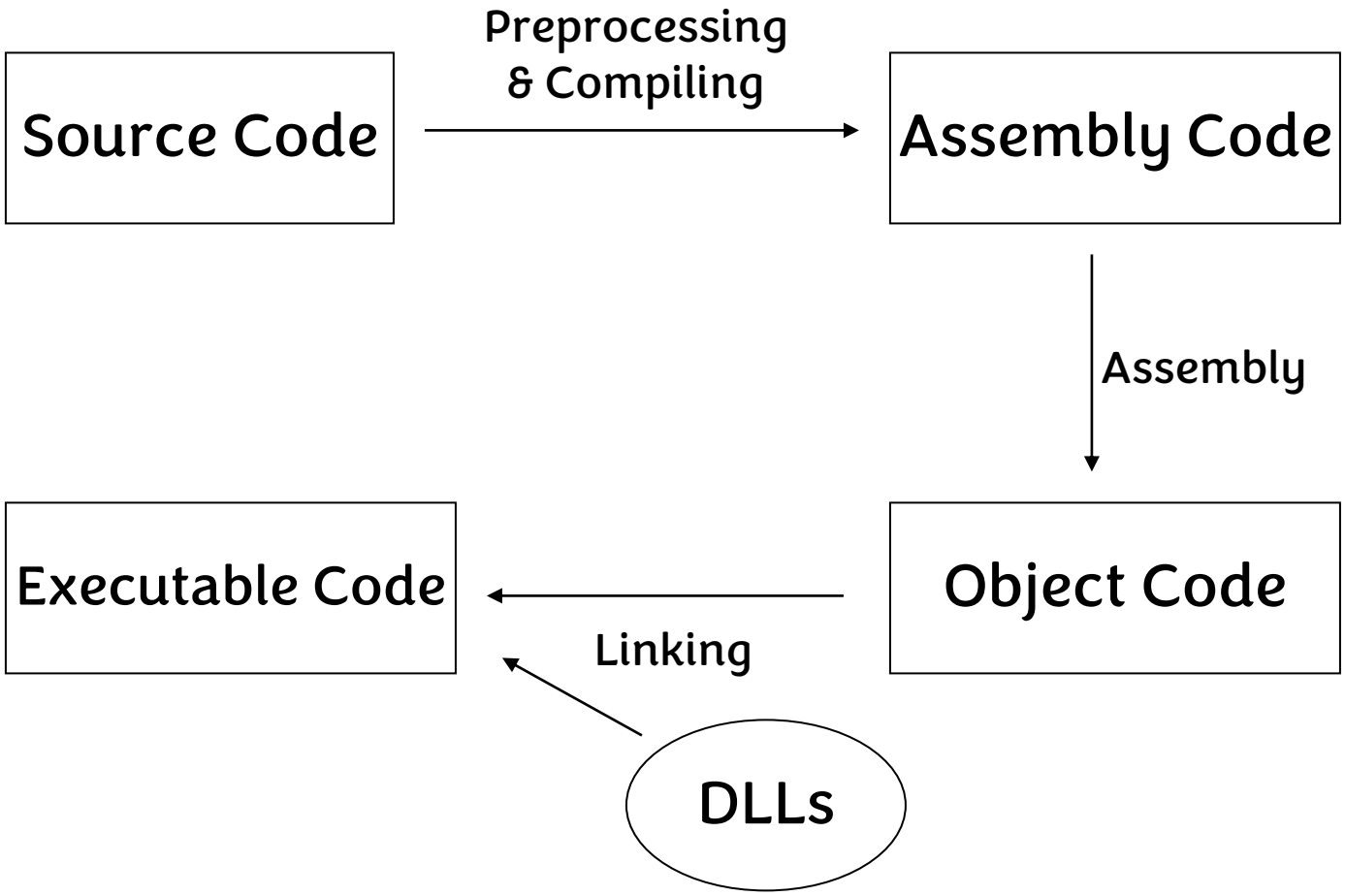
Understanding assembly code is so important in Malware analysis.

Assembly is an essential programming language as cybersecurity experts might use it to interpret malware and understand their modes of attack. Cybersecurity professionals defend against traditional and contemporary malware continuously, so it's essential to understand how malware functions.

Understanding assembly code is so important in malware reverse engineering.

Knowledge of assembly language programming is a must in malware reverse engineering because malware authors do not normally publish their source code, and for that reason, reverse engineering is done

What Does It Mean to Assemble Code?



Key Benefits of Assembly Language

- There is a one-to-one relationship between the assembly and machine language instructions
- What is found is that a compiled machine code implementation of a program written in high-level language results in inefficient code
  - More machine language instructions than an assembled version of an equivalent handwritten assembly language program
- Two key benefits of assembly language programming
  - It takes up less memory
  - It executes much faster

Languages in terms of applications

- One of the most beneficial uses of assembly language programming is real-time applications.

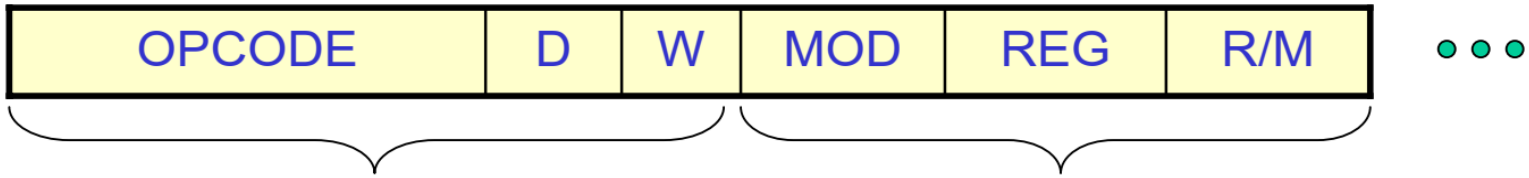
- Real time means the task required by the application must be completed before any other input to the program that will alter its operation can occur.

- For example, the device service routine which controls the operation of the floppy disk drive is a good example that is usually written in assembly language

Languages in terms of applications

- Assembly language is not only good for controlling hardware devices but also for performing pure software operations
  - Searching through a large table of data for a special string of characters
  - Code translation from ASCII to EBCDIC
  - Table sort routines
  - Mathematical routines
- Assembly language: perform real-time operations
- High-level languages: Those operations mostly not critical in time

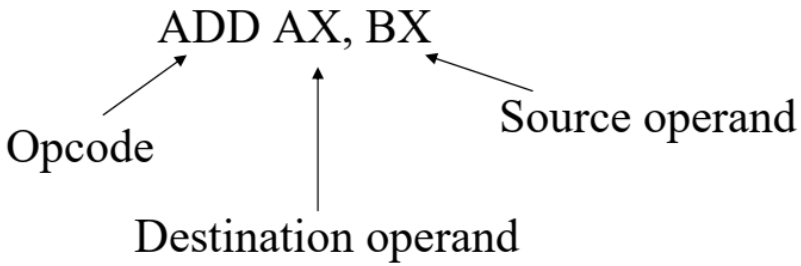
Converting Assembly Language Instructions to Machine Cod



- An instruction can be coded with 1 to 6 bytes
- **Byte 1 contains three kinds of information:**
  - Opcode field (6 bits) specifies the operation such as add, subtract, or move
  - Register Direction Bit (D bit)
    - Tells the register operand in REG field in byte 2 is source or destination operand
      - 1:Data flow to the REG field from R/M
      - 0: Data flow from the REG field to the R/M
  - Data Size Bit (W bit)
    - Specifies whether the operation will be performed on 8-bit or 16-bit data
      - 0: 8 bits
      - 1: 16 bits
- **Byte 2 has two fields:**
  - Mode field (MOD) – 2 bits
  - Register field (REG) - 3 bits
  - Register/memory field (R/M field) – 3 bits

Converting Assembly Language Instructions to Machine Cod

- The sequence of commands used to tell a microcomputer what to do is called a **program**
- Each command in a program is called an **instruction**
- 8086 understands and performs operations for **117 basic instructions**
- The native language of the **IBM PC** is the machine language of 8086/8088
- A program written in machine code is referred to as **machine code**
- In 8086 assembly language, each of the operations is described by alphanumeric symbols instead of just 0s or 1s



Converting Assembly Language Instructions to Machine Cod

- **REG field is used to identify the register for the first operand**

REG	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Converting Assembly Language Instructions to Machine Cod

- 2-bit MOD field and 3-bit R/M field together specify the second operand

CODE	EXPLANATION
00	Memory Mode, no displacement follows*
01	Memory Mode, 8-bit displacement follows
10	Memory Mode, 16-bit displacement follows
11	Register Mode (no displacement)

*Except when R/M = 110, then 16-bit displacement follows

(a)

MOD = 11			EFFECTIVE ADDRESS CALCULATION			
R/M	W = 0	W = 1	R/M	MOD = 00	MOD = 01	MOD = 10
000	AL	AX	000	(BX) + (SI)	(BX) + (SI) + D8	(BX) + (SI) + D16
001	CL	CX	001	(BX) + (DI)	(BX) + (DI) + D8	(BX) + (DI) + D16
010	DL	DX	010	(BP) + (SI)	(BP) + (SI) + D8	(BP) + (SI) + D16
011	BL	BX	011	(BP) + (DI)	(BP) + (DI) + D8	(BP) + (DI) + D16
100	AH	SP	100	(SI)	(SI) + D8	(SI) + D16
101	CH	BP	101	(DI)	(DI) + D8	(DI) + D16
110	DH	SI	110	DIRECT ADDRESS	(BP) + D8	(BP) + D16
111	BH	DI	111	(BX)	(BX) + D8	(BX) + D16

(b)

Example:

- MOV BL,AL
- Opcode for MOV = 100010
- We'll encode AL so
  - D = 0 (AL source operand)
- W bit = 0 (8-bits)
- MOD = 11 (register mode)
- REG = 000 (code for AL)
- R/M = 011 **(Code for BL)**

OPCODE	D	W	MOD	REG	R/M
100010	0	0	11	000	011

MOV BL,AL => 10001000 11000011 = 88 C3h

ADD AX,[SI] => 00000011 00000100 = 03 04 h

ADD [BX][DI] + 1234h, AX => 00000001 10000001 __ __ h

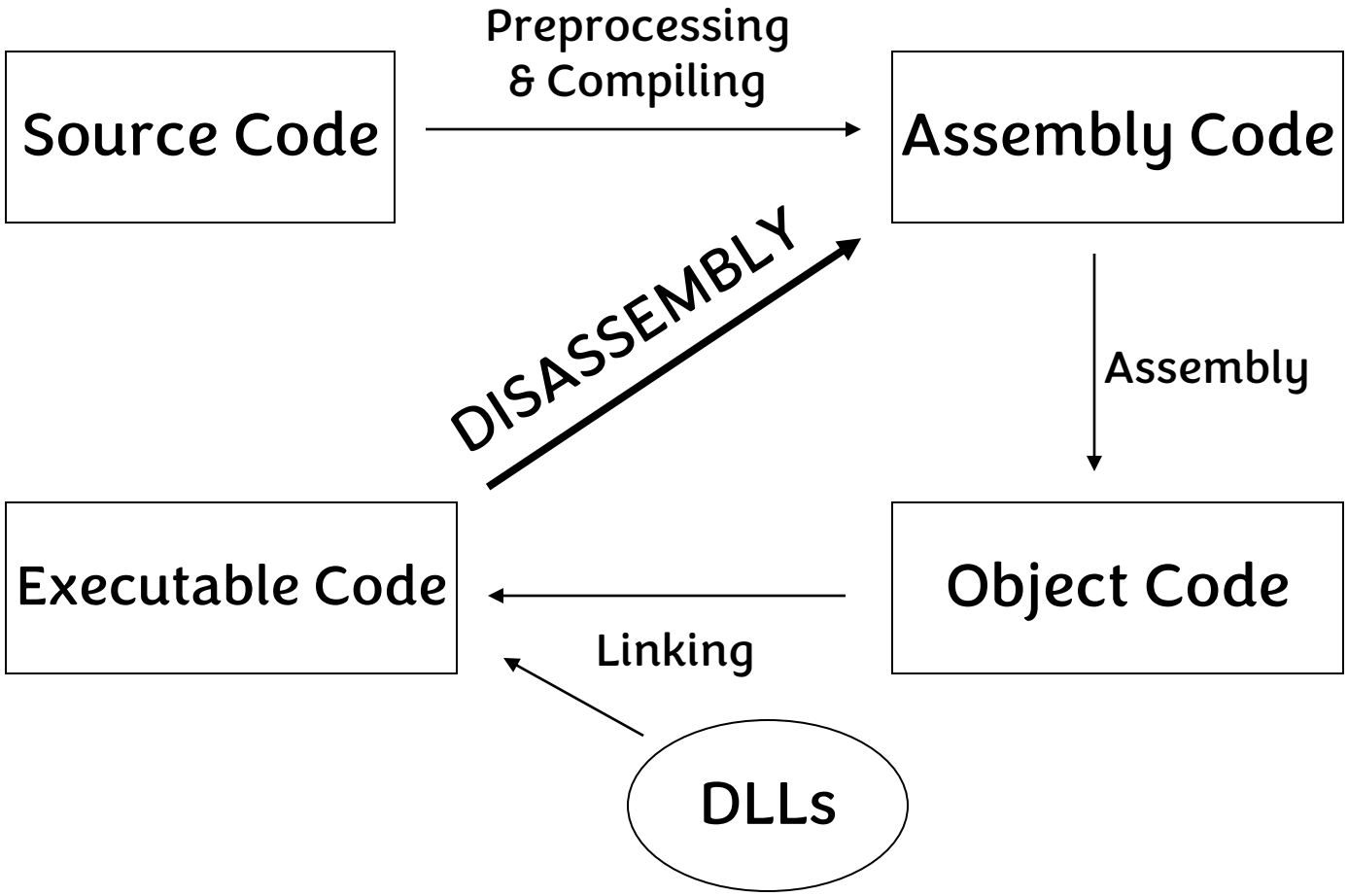
=> 01 81 34 12 h

More Examples

ADD 5678H[BX][SI], CX	01 88 78 56 H
SUB DX, AX	29 C2 H
CMP AX, CX	39 C8 H
MOV [0004], AX	A3 04 00 H
PUSH AX	50 H
POP DX	5A H

Disassemble: Converting Machine Language Code To Assembly Language Code

What Does It Mean to Disassemble Code?



Why is Disassembly Useful in Malware Analysis?

- It is not always desirable to execute malware: disassembly provides a static analysis.
- Disassembly enables an analyst to investigate all parts of the code, something that is not always possible in dynamic analysis.
- Using a disassembler and a debugger in combination creates synergy.

Disassembly of Machine Codes

- Indeed, there's no real difference between machine language and any other programming language; machine language is just a little harder to read.
  - Understanding it requires patience and the right reference.

- Finding the right reference is a large matter of knowing which CPU architecture the machine language was written for as each type of CPU has its own dialect.
  - It can also be important to know what CPU mode the machine language was written for.
  - Modern x86 CPUs, for instance, can be configured to use 16- or 32-bit operands and addressing by default, and the same sequence of machine language bytes may mean different things depending upon the CPU's state.
  - Matters become even more complex when 64-bit instructions are introduced.

Disassembly of Machine Codes

- Since we're looking at a DOS (i.e., x86 real-mode) executable, a good reference is the Instruction Set Reference (ISR) volume from the Intel Architecture Software Developer's Manual.

- This is a formidable volume, but only a few pages are immediately interesting for our purposes. For instance;
  - Pages 1-2 through 2-6 describe the basic layout of x86 machine language instructions. (Note that since we're dealing with real-mode machine language, we're only interested in 16-bit addressing modes.)
  - Pages A-1 through A-8 give the processor's opcode map. (Note that since we're dealing with such an old program, we can assume that it only uses 8086 integer opcodes; this means that we can ignore all two-byte and escape opcodes in the opcode map.)

8086 Instruction Set Opcodes (1)

Operation	Operands	Opcode
ADC	see ADD	ADD opcode + $10, and xx010xxx (ModR/M byte) for $80-$83
ADD	r/m8, reg8	$00
ADD	r/m16, reg16	$01
ADD	reg8, r/m8	$02
ADD	reg16, r/m16	$03
ADD	AL, imm8	$04
ADD	AX, imm16	$05
ADD	r/m8, imm8	$80 xx000xxx (ModR/M byte)
ADD	r/m16, imm16	$81 xx000xxx (ModR/M byte)
ADD	r/m16, imm8	$83 xx000xxx (ModR/M byte)
AND	see ADD	ADD opcode + $20, and xx100xxx (ModR/M byte) for $80, $81,$83
CALL	32-bit displacement	$9A
CALL	16-bit displacement	$E8
CLD		$FC
CMP	See ADD	ADD opcode + $38, and xx111xxx (ModR/M byte) for $80, $81,$83
CMPSB	ES:[DI]==DS:[SI]	$A6
CMPW	ES:[DI]==DS:[SI]	$A7
DEC	r/m8	$FE, xx001xxx (ModR/M byte)
DEC	r/m16	$FF, xx001xxx (ModR/M byte)
DEC	reg16	$48 + reg16 code
DIV	r/m8	$F6, xx110xxx (ModR/M byte)
DIV	r/m16	$F7, xx110xxx (ModR/M byte)
HLT		$F4
IDIV	r/m8	$F6, xx111xxx (ModR/M byte)
IDIV	r/m16	$F7, xx111xxx (ModR/M byte)
IMUL	r/m8	$F6, xx101xxx (ModR/M byte)
IMUL	r/m16	$F7, xx101xxx (ModR/M byte)

8086 Instruction Set Opcodes (2)

Operation	Operands	Opcode
IN	AL, addr8	$E4
IN	AX, addr8	$E5
IN	AL, port[DX]	$EC
IN	AX, port[DX]	$ED
INC	r/m8	$FE, xx000xxx (ModR/M byte)
INC	r/m16	$FF, xx000xxx (ModR/M byte)
INC	reg16	$40 + reg16 code
IRET	48-bit POP	$CF
JA	8-bit relative	$77
JAE	8-bit relative	$73
JB	8-bit relative	$72
JBE	8-bit relative	$76
JE	8-bit relative	$74
JG	8-bit relative	$7F
JGE	8-bit relative	$7D
JL	8-bit relative	$7C
JLE	8-bit relative	$7E
JMP	32-bit displacement	$EA
JNE	8-bit relative	$75
JZ	8-bit relative	$74
LDS	reg16, mem32	$C4
LES	reg16, mem32	$C5
LODSB	AL = DS:[SI]	$AC
LODSW	AX = DS:[SI]	$AD

8086 Instruction Set Opcodes (3)

Operation	Operands	Opcode
LOOP	8-bit relative	$E2
MOV	r/m8, reg8	$88
MOV	r/m16, reg16	$89
MOV	AL, mem8	$A0
MOV	AX, mem16	$A1
MOV	mem8, AL	$A2
MOV	mem16, AX	$A3
MOV	reg8, imm8	$B0 + reg8 code
MOV	reg16,imm16	$B8 + reg16 code
MOV	r/m8, imm8	$C6, xx000xxx(ModR/M byte)
MOV	r/m16, imm16	$C7, xx000xxx(ModR/M byte)
MOV	r/m16,sreg	$8C, xx0 sreg xxx(ModR/M byte)
MOV	sreg, r/m16	$8E, xx0 sreg xxx(ModR/M byte)
MOVSB	ES:[DI] = DS:[SI]	$A4
MOVSW	ES:[DI] = DS:[SI]	$A5
MUL	r/m8	$F6, xx100xxx (ModR/M byte)
MUL	r/m16	$F7, xx100xxx (ModR/M byte)
NEG	r/m8	$F6, xx011xxx (ModR/M byte)
NEG	r/m16	$F7, xx011xxx (ModR/M byte)
NOT	r/m8	$F6, xx010xxx (ModR/M byte)
NOT	r/m16	$F7, xx010xxx (ModR/M byte)
OR	see ADD	ADD opcode + $08, and xx001xxx (ModR/M byte) for $80, $81,$83

8086 Instruction Set Opcodes (4)

Operation	Operands	Opcode
OUT	addr8, AL	$E6
OUT	addr8, AX	$E7
OUT	port[DX], AL	$EE
OUT	port[DX], AX	$EF
POP	r/m16	$8F
POP	reg16	$58 + reg16 code
POP	sreg	$07 + ES = 0, CS = 8, SS = $10, DS = $18
PUSH	r/m16	$FF, xx110xxx (ModR/M byte)
PUSH	reg16	$50 + reg16 code
PUSH	sreg	$06 + ES = 0, CS = 8, SS = $10, DS = $18
REP		$F3
REPNE		$F2
RET	32-bit POP	$CA
RET	16-bit POP	$C2
SBB	see ADD	ADD opcode + $18, and xx011xxx (ModR/M byte) for $80, $81,$83
SCASB	ES:[DI] == AL	$AE
SCASW	ES:[DI] == AX	$AF
STD		$FD
STOSB	ES:[DI] = AL	$AA
STOSW	ES:[DI] = AX	$AB
SUB	see ADD	ADD opcode + $28, and xx101xxx (ModR/M byte) for $80, $81,$83
XOR	see ADD	ADD opcode + $30, and xx110xxx (ModR/M byte) for $80, $81,$83

8086 Instruction Set Opcodes (5)

addr8 = 8-bit address of I/O port
reg8 = AL = 0, CL = 1, DL = 2, BL = 3, AH =4, CH = 5, DH = 6, BH = 7
reg16 = AX = 0, CX =1, DX =2, BX =3, SP = 4, BP = 5, SI = 6, DI = 7
sreg = ES = 0, CS = 1, SS = 2, DS = 3
mem8 = memory byte (direct addressing only)
mem16 = memory word (direct addressing only)
r/m8 = reg8 or mem8
r/m16 = reg16 or mem16
imm8 = 8 bit immediate
imm16 = 16 bit immediate

Example 1 of Code Disassembly

Assume the first bytes of machine language code are located at offset 01000H. They are:

8C C0 05 10 00 0E 1FA3 04 00 03 06 0C 00 8E C0 ...

Disassemble this code to obtain an assembly language code?

Example of Code Disassembly

8C C0	MOV AX ES
05 10 00	ADD AX 0010H
0E	PUSH CS
1F	POP DS
A3 04 00	MOV [0004], AX
03 06 0C 00	ADD AX , [000C]
8E C0 ...	MOV ES AX

Example 2 of Code Disassembly

Assume the first bytes of machine language code are located at offset 01000H. They are:

01 81 56 78 8E C0 8A D8 0E 04 3D 03 06 0C 00 1F 8C C0

Disassemble this code to obtain an assembly language code.

Example of Code Disassembly

00000H: 01 81 56 78	ADD [BX] [DI] + 7856H, AX
00004H: 8E C0	MOV ES, AX
00006H: 8A D8	MOV BL, AL
00008H: 0E	PUSH CS
00009H: 04 3D	ADD AL, 3DH
0000DH: 03 06 0C 00	ADD AX, [000CH]
00011H: 1F	POP DS
00012H: 8C C0	MOV AX, ES

Online Assembler and Disassembler

Provide Assembler and Disassembler tools for different microprocessor architectures

Try it here

https://shell-storm.org/online/Online-Assembler-and-Disassembler/

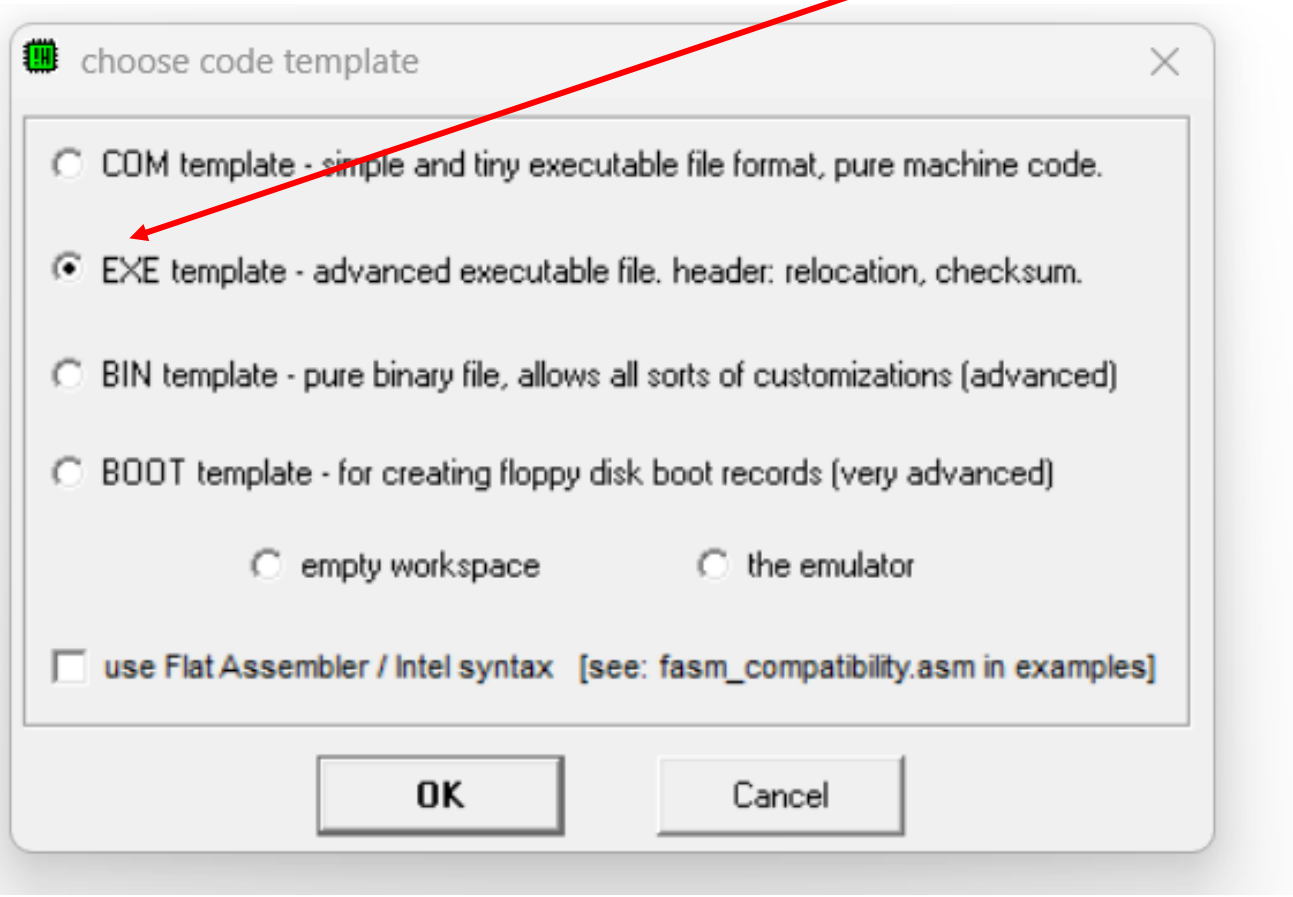
Creating.EXE applications

Using MASM Assembler

Using MASM

- Developed by Microsoft
- Used to translate 8086 assembly language into machine language
- 3 steps:
  - Prepare.ASM file using a text editor
  - Compile your.ASM file using MASM
  - Create. EXE file using LINKer
  - Once you have the. EXE file, debug can be used to test and run the program

Using MASM

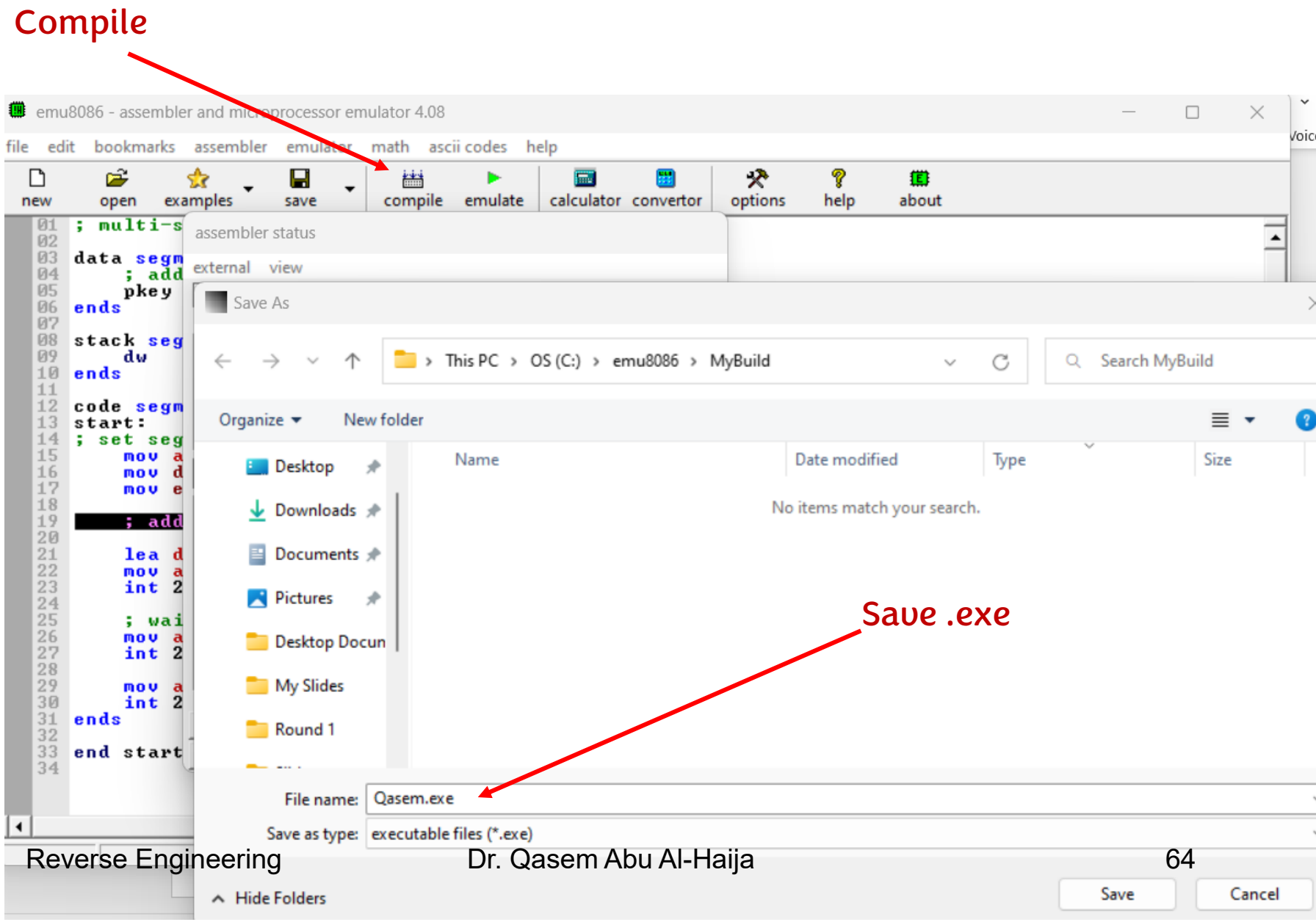

Select this option
choose code template
COM template - simple and tiny executable file format, pure machine code.
EXE template - advanced executable file. header: relocation, checksum.
BIN template - pure binary file, allows all sorts of customizations (advanced)
BOOT template - for creating floppy disk boot records (very advanced)
empty workspace
the emulator
use Flat Assembler / Intel syntax [see: fasm_compatibility.asm in examples]
OK
Cancel

Using MASM

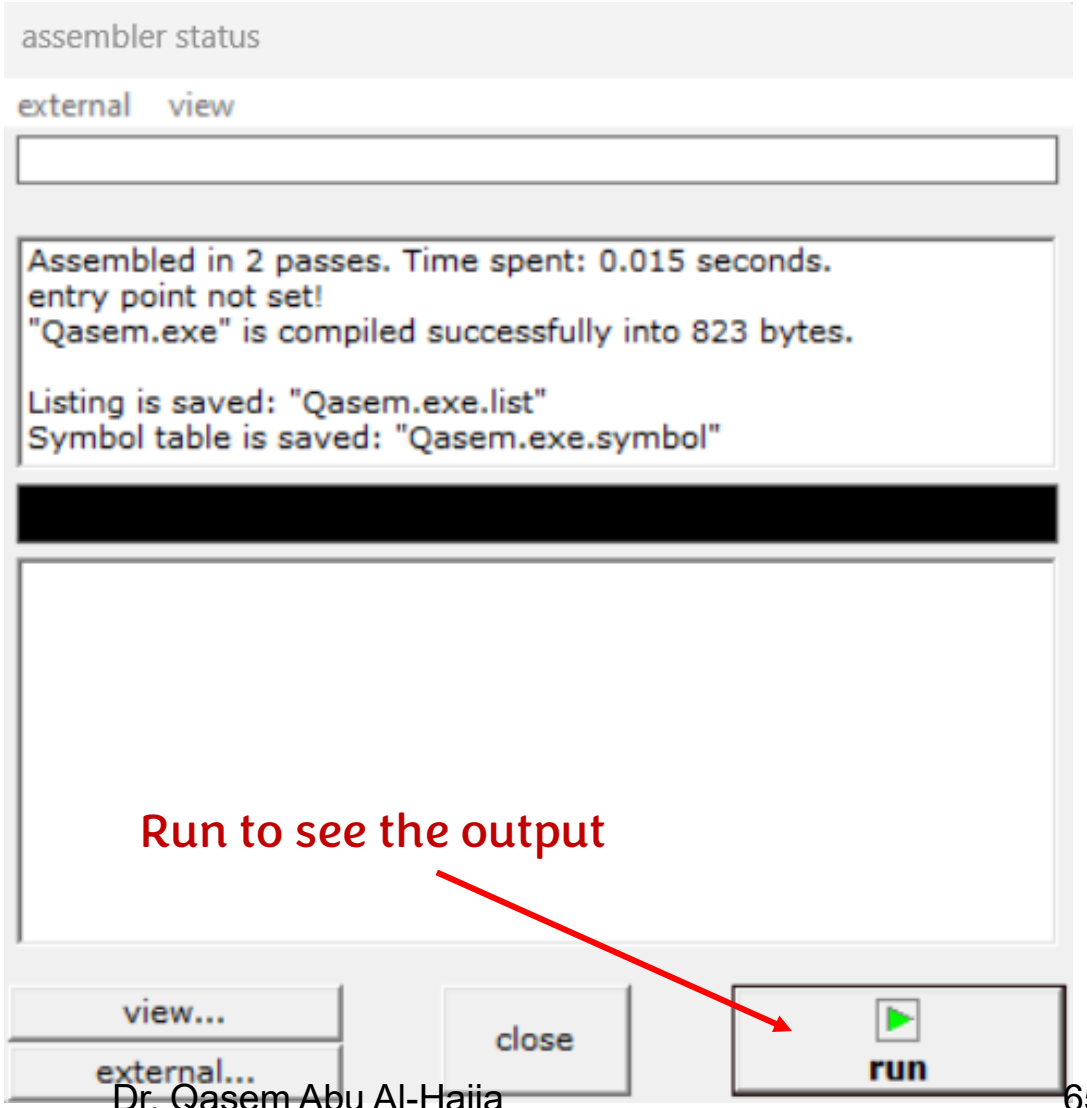
```
01 ; multi-segment executable file template.
02
03 data segment
04     ; add your data here!
05     pkey db "press any key...$"
06 ends
07
08 stack segment
09     dw   128  dup(0)
10 ends
11
12 code segment
13 start:
14 ; set segment registers:
15     mov ax, data
16     mov ds, ax
17     mov es, ax
18
19     ; add your code here
20
21     lea dx, pkey
22     mov ah, 9
23     int 21h        ; output string at ds:dx
24
25     ; wait for any key....
26     mov ah, 1
27     int 21h
28
29     mov ax, 4c00h ; exit to operating system.
30     int 21h
31 ends
32
33 end start ; set entry point and stop the assembler.
34
```

Prepare your ASM code

Using MASM


Compile
emu8086 - assembler and microprocessor emulator 4.08
file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate calculator convertor options help about
assembler status
external view
Save As
This PC › OS (C:) › emu8086 › MyBuild
Search MyBuild
Organize New folder
Desktop
Downloads
Documents
Pictures
Desktop Docun
My Slides
Round 1
Name
Date modified
Type
Size
No items match your search.
Save .exe
File name: Qasem.exe
Save as type: executable files (*.exe)
Hide Folders
Save
Cancel

Using MASM


assembler status
external
view
Assembled in 2 passes. Time spent: 0.015 seconds.
entry point not set!
"Qasem.exe" is compiled successfully into 823 bytes.
Listing is saved: "Qasem.exe.list"
Symbol table is saved: "Qasem.exe.symbol"
Run to see the output
view...
external...
close
run

Using MASM

The output screen

Using MASM

The exe file can be reversed:

disassembled using IDA Pro

or

debugged using OlyDbg

Thank you