Forensics II

Usage of RCE?

Managed code and obfuscation

Assembly basics

Executable formats [some repetition]

What is Reverse Code Engineering?

- Some people say "Reverse Engineering is an art"
  - It is more an application of standard methods that evolve constantly, actually, everybody can learn these methods and start to RE executables
- Reverse engineering is like solving a jigsaw puzzle
  - In order to see the whole picture you need to find the corner pieces, then the frame, and then work your way forward from there
- The corner pieces for reversing are strings, constants and function names
  - The function names that people normally start with are the one's imported from shared libraries (e.g. Dlls)
  - Strings contain human readable hints about the functionality
  - Specific constants add more clues to solve the puzzle or can sometimes even be used to identify certain (types of) algorithms
- The major problem is that a lot of experience is needed to identify strings, constants and to know what the combination of imported functions may result in

Usage of Reverse Engineering

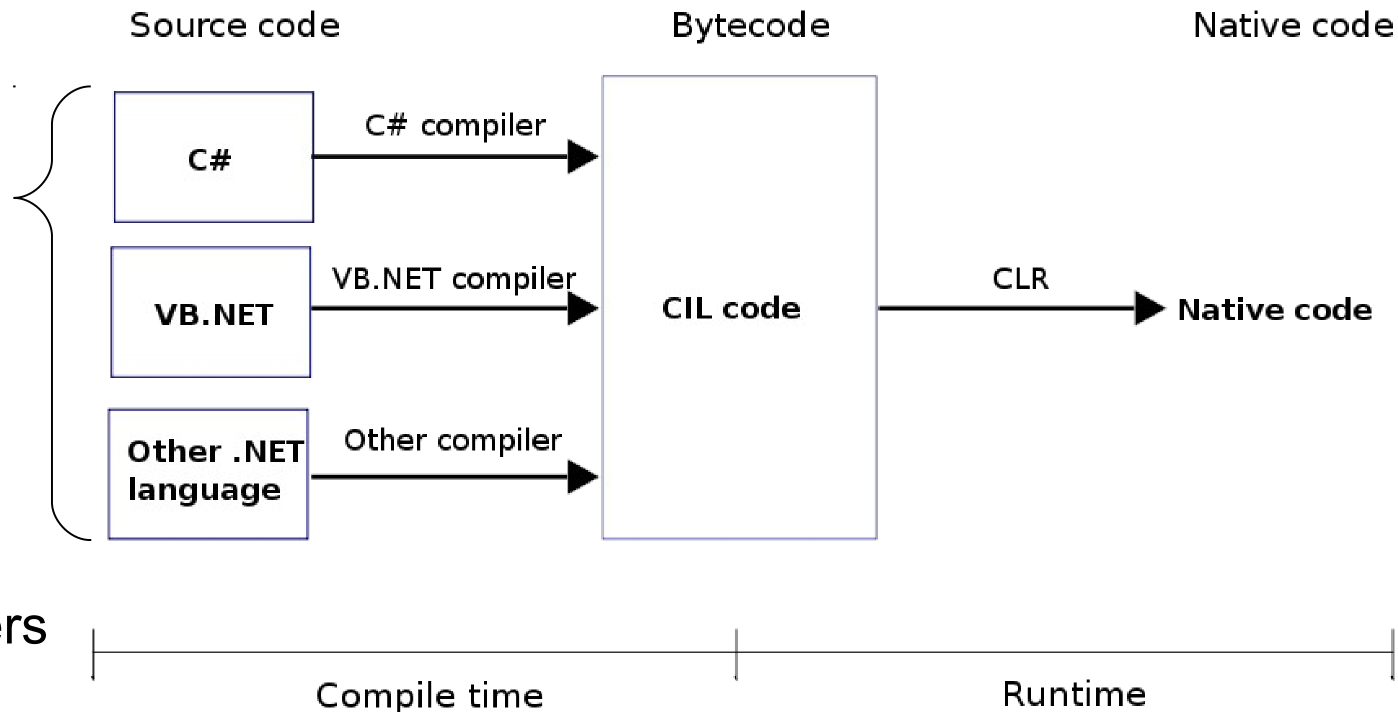
- Common uses of reverse engineering include
  - Recovery of business data from proprietary file formats
  - Creation of hardware documentation from binary drivers, often for producing Linux drivers from Windows or Apple drivers
  - Enhancing consumer electronics devices
  - Malware analysis and creation, often involving a search for security holes when systems inter-operate
  - Discovery of undocumented APIs that may be useful
  - Military or commercial espionage
  - Copyright and patent litigation
  - Breaking software copy protection (legally and not), often for games and expensive engineering software
  - Academic/Learning purposes and curiosity
  - Etc. ...

Patent troll

- Example Rockstar
  - Owned by Apple, Microsoft, BlackBerry, Sony and Ericsson
  - Bought Nortel patents (former Canadian telecom company) worth $4,5 billion in 2012
  - “Rockstar produces no products and practices no patents. Instead, Rockstar employs a staff of engineers in Ontario, Canada, who examine other companies’ successful products to find anything that Rockstar might use to demand and extract licenses to its patents under threat of litigation.” - Google 2013-12

RCE of managed code 1

- Managed code
  - .NET family of languages, Java etc.
  - CLR (Common Language Runtime), JVM (JRE)
  - Java bytecode and the Common Intermediate Language CIL (previously known as MSIL - Microsoft Intermediate Language) bytecode
- CLI (Common Language Infrastructure)
- Obfuscation
  - Fight decompilers and disassemblers



RCE of managed code 2

- Managed code is much more simple than native code to RCE, Why?
- If not obfuscated more or less a source backup can be made!
- Red Gate .NET Reflector pro (trial)
  - VS debug and decompile support
  - http://www.red-gate.com/products/dotnet-development/reflector/
- Free .NET Reflector v6 and add-ins
  - http://27.am/posts/how-to-download-net-reflector-6-for-free
  - Many useful add-ins
  - https://reflectoraddins.codeplex.com/
  - http://www.red-gate.com/products/dotnet-development/reflector/add-ins
- Reflector demo of LookingGlassReflector program
  - Export disasseblies to files

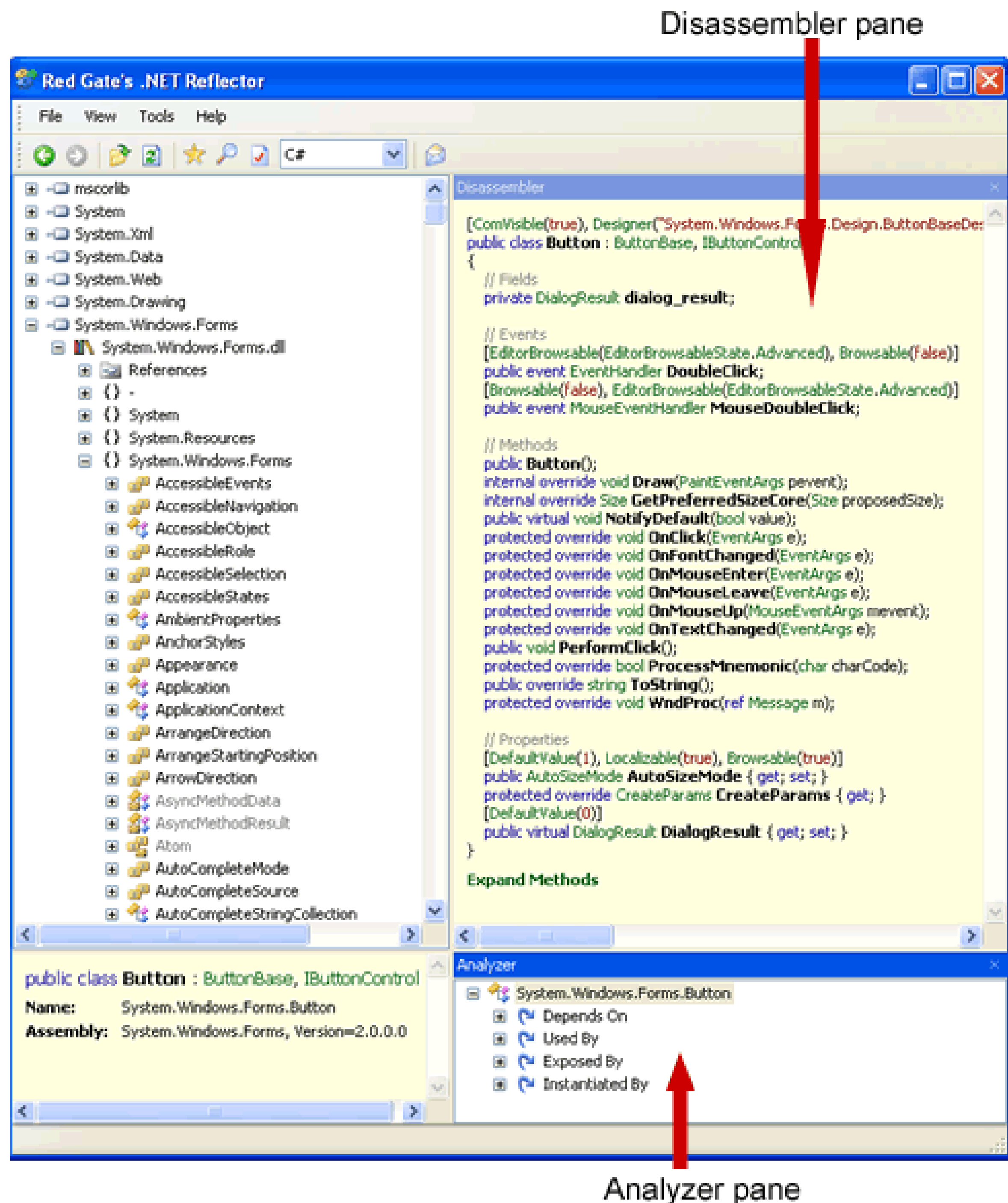
RCE of managed code 3

Tutorial

- https://www.simple-talk.com/dotnet/.net-tools/first-steps-with-.net-reflector/

Free alternatives to Reflector (.NET decompiler)

http://blog.wibeck.org/2013/02/free-options-for-reflector-net-decompiler/



Protect managed code 1

Why you need obfuscation?

- Goal of obfuscation – Create confusion
- Benefits of obfuscation
  - Post-development recompilation system
  - It analyzes applications and makes them **smaller**, **faster**, and harder to reverse-engineer (**more secure**) - In short, it makes them better!
- Renaming (code renamed to compact names) and overload induction
  - Rename as many methods and variables as possible to the same name, use return type and parameters as a criterion in determining uniqueness
- String Encryption – Makes it very hard to locate strategic logic
- Control Flow Obfuscation – Produces spagetti logic, hard to analyze
- Pruning – Removes unused code
- Assembly Linking – Merge multiple assemblies into one
- Watermarking – Embed a signature

Debugging Obfuscated Code - bug reports, stack traces etc. is a problem

- Using the renaming map file with special tools can decode stack traces

Obfuscation examples C#

Renaming and Overload Induction

Original Source Code Before Obfuscation

```
private void CalcPayroll(SpecialList employeeGroup) {
  while (employeeGroup.HasMore()) {
      employee = employeeGroup.GetNext(true);
      employee.UpdateSalary();
      DistributeCheck(employee);
   }
}
```

Reverse-Engineered Source Code
After Overload Induction Dotfuscation

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

Renaming and Control Flow Obfuscation

Original Source Code Before Obfuscation
(Snippet from WordCount.cs C# example code)

```
public int CompareTo(Object o) {
   int n = occurrences – ((WordOccurrence)o).occurrences;
   if (n == 0) {
      n = String.Compare(word, ((WordOccurrence)o).word);
   }
   return(n);
}
```

Reverse-Engineered Source Code
After Control Flow Obfuscation

```
public virtual int _a(Object A_0) {
   int local0, local1;
   local0 = this.a – (c) A_0.a;
   if (local0 != 0) goto i0;
   goto i1;
   while (true) {
     return local1;
     i0: local1 = local0;
   }
   i1: local0 = System.String.Compare(this.b, (c) A_0.b);
   goto i0;
}
```

Protect managed code 2

- PreEmptive Dotfuscator Community edition bundled with VS 20*

- String encryption

Disassembler

```
private void □(int □)
{
    □.□("□", new object[] { this, □ });
}
```

PreEmptive Dotfuscator and Analytics CE - Dotfuscator1.xml*

File View Build Tools Help

Start
Dotfuscator1
Inputs
Properties
Configuration Options
Analytics
Renaming
Control Flow
String Encryption
Removal
Linking
PreMark
Results

Getting Started
Exclusions
Built-In Rules
Options

Secure .NET Software Development Lifecycle

Develop and Build (Use Secure Coding Principles)
Design (Threat Modeling)
Protect
- Protect .NET Code
- Thwart Decompilers
- Reduce Size of .NET Applications
- Improve Application Efficiency

dotfuscator
Deploy (Monitor for Breaches)
Test (Penetration Testing)

Build Output

Ready.

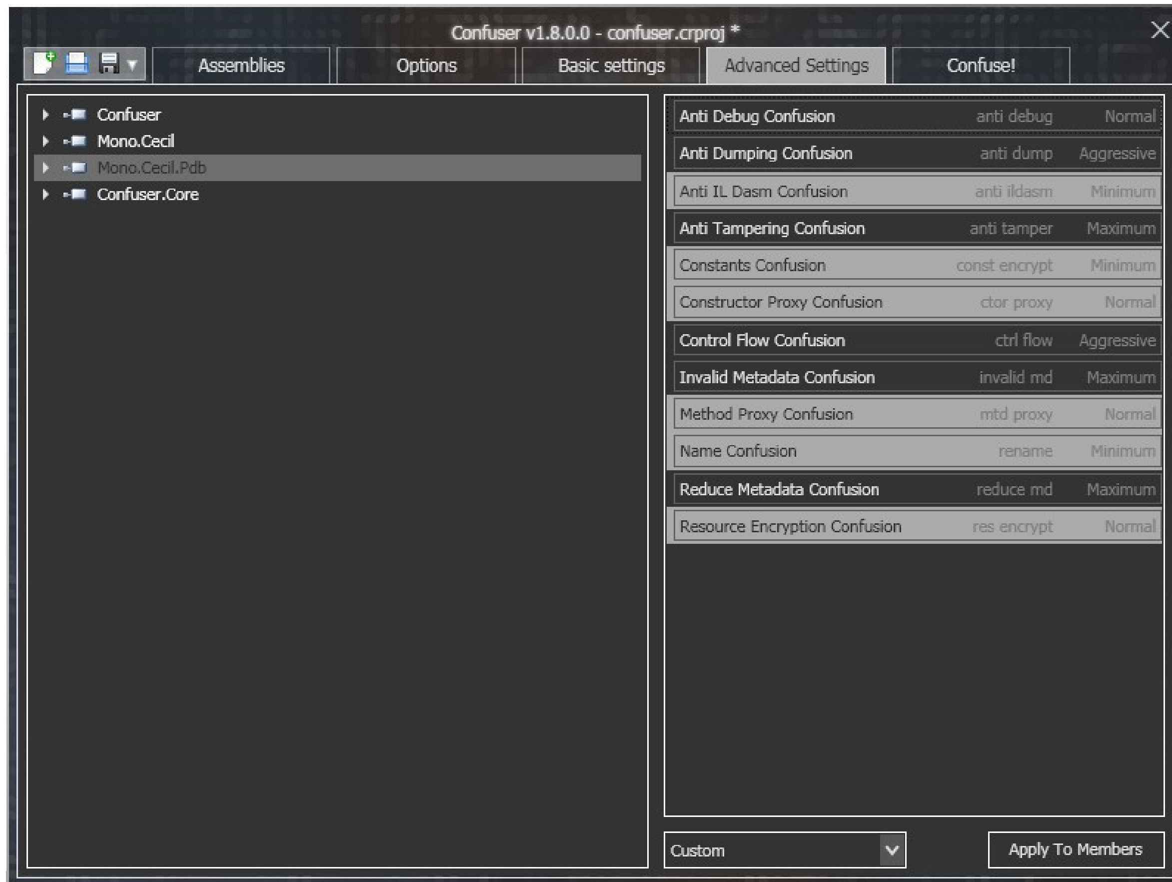
Good list: http://www.csharp411.com/net-obfuscators/

Protect managed code 3

- Confuser - free
- http://confuser.codeplex.com/

Features:

- Anti debugger
- Anti memory dumping
- Anti decompiler
- Prevent any tampering of the assemblies
- Encrypt codes
- Encrypt constants (i.e. numbers & strings)
- Encrypt resources
- Control flow obfuscation
- External/Internal reference proxy
- Renaming



List: http://en.wikipedia.org/wiki/List_of_obfuscators_for_.NET

Java decompilers, debuggers and obfuscators

- The builtin one

javap -c class-file

- Best java decompiler is free! →
- Others
  - DJ Java
  - Jad
- ProGuard
  - Obfuscator
- JDebugTool

Java Decompiler

File Edit Navigate Help

petstore-ejb3.0.jar | Object.class | String.class | CharSequence.class | Integer.class

- META-INF
- petstore
  - ejb
    - dao
      - EJBCatalogDAO.class
      - EJBCatalogDAORemote.class
    - model
      - EntityCategory.class
      - EntityCategoryDetail.class
      - EntityItem.class
      - EntityItemDetail.class
      - EntityProduct.class
      - EntityProductDetail.class
    - util

EJBCatalogDAO.class | EntityCategory.class

```
import javax.persistence.PersistenceContext;
import javax.persistence.Table;

@Entity
@Table(name="CATEGORY")
public class EntityCategory
{
  @PersistenceContext
  private EntityManager em;
  private int id;
  private Map<String, EntityCategoryDetail> details;

  public EntityCategory()
  {
    this.id = 0;
    this.details = new HashMap();
  }

  @Id
  @GeneratedValue
  @Column(name="ID")
  public int getId()
  {
    return this.id;
```

Find: id | Next | Previous | Case senstive

JD-GUI

demo if time permits

http://jd.benow.ca/

http://java-source.net/open-source/obfuscators

IA-32 (x86) assembly
Internal buses and registers

Segment registers
15 0
CS
DS
SS
ES
FS
GS

Control unit

Address bus
Data bus
Control bus

Memory
I/O peripherals

ALU

31 0
EFLAGS
EIP

General purpose registers

	31	15	7	0	
EAX		AH	AL		AX
EBX		BH	BL		BX
ECX		CH	CL		CX
EDX		DH	DL		DX
ESI		SI			Offset registers
EDI		DI			
EBP		BP			
ESP		SP			

Figure 7-1 Diagram of the inside of a modern Intel processor

- Address bus
  - Select addresses to read/write to memory
- Data bus
  - Move data around the CPU and to/from memory
- Control bus
  - Control external devices and execute instructions

Floating point registers, ST(0) through ST(7) , 80 bits wide

Debug registers DR0 - DR7

General Purpose 32-bit Registers

EAX	Contains the return value of a function call.
ECX	Used as a loop counter. "this" pointer in C++.
EBX	General Purpose
EDX	General Purpose
ESI	Source index pointer
EDI	Destination index pointer
ESP	Stack pointer
EBP	Stack base pointer

Segment Registers

CS	Code segment
SS	Stack segment
DS	Data segment
ES	Extra data segment
FS	Points to Thread Information Block (TIB)
GS	Extra data segment

Misc. Registers

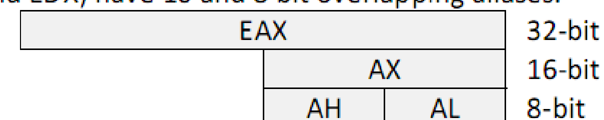
EIP	Instruction pointer
EFLAGS	Processor status flags.

Status Flags

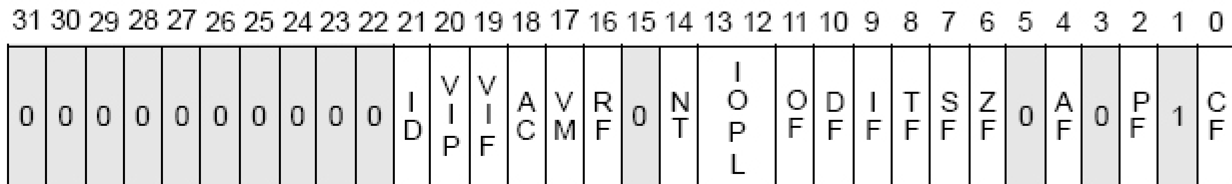
ZF	Zero: Operation resulted in Zero
CF	Carry: source > destination in subtract
SF	Sign: Operation resulted in a negative #
OF	Overflow: result too large for destination

16-bit and 8-bit Registers

The four primary general purpose registers (EAX, EBX, ECX and EDX) have 16 and 8 bit overlapping aliases.



EFLAGS



X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)

RCE S flags

S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag

Reserved bit positions. DO NOT USE.
Always set to values previously read.

Addressing mode

<mnemonic> <dest>, <src>

The Netwide Assembler
http://www.nasm.us/

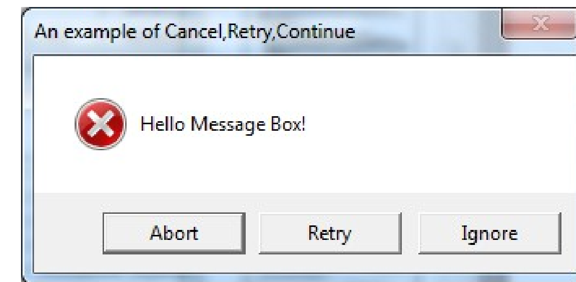
Addressing Mode	Description	NASM Examples
Register	Registers hold the data to be manipulated. No memory interaction. Both registers must be the same size.	mov ebx, edx add al, ch
Immediate	Source operand is a numerical value. Decimal is assumed; use **h** for hex.	mov eax, 1234h mov dx, 301
Direct	First operand is the address of memory to manipulate. It's marked with brackets.	mov bh, 100 mov[4321h], bh
Register Indirect	The first operand is a register in brackets that holds the address to be manipulated.	mov [di], ecx
Based Relative	The effective address to be manipulated is calculated by using **ebx** or **ebp** plus an offset value.	mov edx, 20[ebx]
Indexed Relative	Same as Based Relative, but **edi** and **esi** are used to hold the offset.	mov ecx, 20[esi]
Based Indexed-Relative	The effective address is found by combining based and indexed modes.	mov ax, [bx][si]+1

- Intel Hex Opcodes (the binary instructions) And Mnemonics
  - [server]\tools\IDA Pro\opcodes.hlp

Microsoft Macro Assembler

```
; MASM Hello World! Console program
; Visual Studio vcvars32.bat or CMD prompt
; ml.exe cons.asm /link /subsystem:console
.386
.model flat, c
includelib kernel32.lib
.data
szHello db 'Hello, world!',0dh,0ah
HelloLen equ 15
STD_OUT_HANDLE equ -11
.code
GetStdHandle PROTO stdcall :DWORD
WriteConsoleA PROTO stdcall :DWORD,
      :DWORD, :DWORD, :DWORD, :DWORD
WriteConsole equ WriteConsoleA
ExitProcess PROTO stdcall :DWORD
start proc c public
local hStdout: DWORD
local dwNumWrit: DWORD
invoke GetStdHandle, STD_OUT_HANDLE
mov [hStdout], eax
lea edx, [dwNumWrit]
invoke WriteConsole, hStdout, offset
      szHello, HelloLen, edx, 0
invoke ExitProcess, 0
start endp
end start
```

```
; ml.exe mbox.asm /link /subsystem:windows
.386
.model flat, stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib
include \masm32\include\user32.inc
includelib \masm32\lib\user32.lib
.data
MsgBoxCaption     db "An example of Cancel,
                  Retry,Continue",0
MsgBoxText  db "Hello Message Box!",0
.code
start:
invoke MessageBox, NULL, addr MsgBoxText,
      addr MsgBoxCaption,
      MB_ICONERROR OR MB_ABORTRETRYIGNORE
.IF eax==IDABORT
      ; Abort was pressed
.ELSEIF eax==IDRETRY
      ; Retry was pressed
.ELSEIF eax==IDCANCEL
      ; Cancel was pressed
.ENDIF
invoke ExitProcess,NULL
end start
```



ASM commands/operators

- In most cases you will only be dealing with the general purpose registers, the instruction pointer, the segment registers and the status register
- OFFSET - Returns the offset from the beginning of the data segment
- PTR - Used to override the default size of an operator (casting in C)
- SIZEOF - As in C
- Hex dump - opcodes
  - 0x55, 0x8BEC, 0x83C4F8, 0x6AF5, 0x…
- Shellcode to x86 (asm, exe) converter

```
.data
myDouble DWORD 1234h
.code
mov ax, myDouble ; error is raised!
; two ways to fix it, how?
```

http://zeltser.com/reverse-malware/convert-shellcode.html

Hello World (cons.asm) as OllyDbg show it with MASM disasm syntax

Address	Hex dump	Disassembly	Comment
00401000	$ 55	PUSH EBP	
00401001	. 8BEC	MOV EBP,ESP	
00401003	. 83C4 F8	ADD ESP,-8	
00401006	. 6A F5	PUSH -0B	DevType = STD_OUTPUT_HANDLE
00401008	. E8 1F000000	CALL <JMP.&KERNEL32.GetStdHandle>	GetStdHandle
0040100D	. 8945 FC	MOV DWORD PTR SS:[EBP-4],EAX	
00401010	. 8D55 F8	LEA EDX,DWORD PTR SS:[EBP-8]	
00401013	. 6A 00	PUSH 0	pReserved = NULL
00401015	. 52	PUSH EDX	pWritten
00401016	. 6A 0F	PUSH 0F	CharsToWrite = F (15.)
00401018	. 68 00304000	PUSH cons.00403000	Buffer = cons.00403000
0040101D	. FF75 FC	PUSH DWORD PTR SS:[EBP-4]	hConsole
00401020	. E8 0D000000	CALL <JMP.&KERNEL32.WriteConsoleA>	WriteConsoleA
00401025	. 6A 00	PUSH 0	ExitCode = 0
00401027	. E8 0C000000	CALL <JMP.&KERNEL32.ExitProcess>	ExitProcess
0040102C	$-FF25 08204000	JMP DWORD PTR DS:[<&KERNEL32.GetStdHand	kernel32.GetStdHandle
00401032	$-FF25 00204000	JMP DWORD PTR DS:[<&KERNEL32.WriteConso	kernel32.WriteConsoleA
00401038	.-FF25 04204000	JMP DWORD PTR DS:[<&KERNEL32.ExitProces	kernel32.ExitProcess

MASM - Irvine

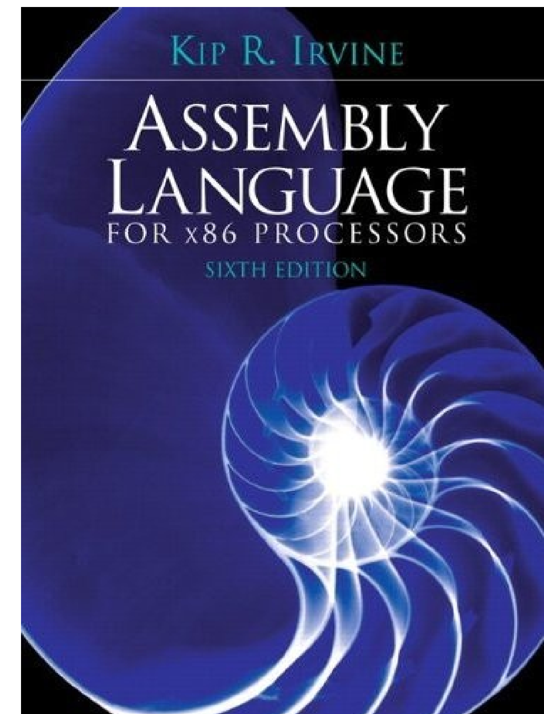
```
TITLE Krypteringsprogram by hjo
INCLUDE     C:\ASM_IA32\Irvine32.inc
XORVAL = 239              ; cryptkey

.data
plainString BYTE 80 DUP(0),0
cryptString BYTE 80 DUP(0),0
copyString  BYTE 80 DUP(0),0
byteCount   DWORD ?

.code
main  PROC
xor   eax,eax     ; nollställ eax
mov   edx,OFFSET plainString
mov   ecx,SIZEOF plainString
;call DumpRegs
call  ReadString              ; Reads string from stdin
mov   byteCount,eax           ; spara antal tecken
mov   esi,0
lp1:
mov   al,plainString[esi]     ; char in al
xor   al,XORVAL               ; kryptera bokstaven
mov   cryptString[esi],al     ; spara krypterade char
inc   esi
dec   byteCount
jnz   lp1
mov   edx,OFFSET cryptString
call  WriteString             ; echo crypt string
mov   byteCount,esi           ; get len of crypt string
mov   esi,0                   ; reset index
lp2:
mov   al,cryptString[esi]
xor   al,XORVAL               ; dekryptera bokstaven
mov   copyString[esi],al      ; spara dekrypterade char i en kopia
inc   esi
dec   byteCount
jnz   lp2
mov   al,0Ah                  ; skriv nyrad LF
call  WriteChar
mov   edx,OFFSET copyString
call  WriteString             ; echo decrypted copy string
exit
main  ENDP
END   main
```

- MASM
  - http://en.wikipedia.org/wiki/Microsoft_Macro_Assembler
  - http://www.masm32.com/
- Easy Code
  - http://www.easycode.cat
- http://en.wikipedia.org/wiki/Comparison_of_assemblers

Easy PC ASM start!
http://kipirvine.com/asm/



Function calls and the stack

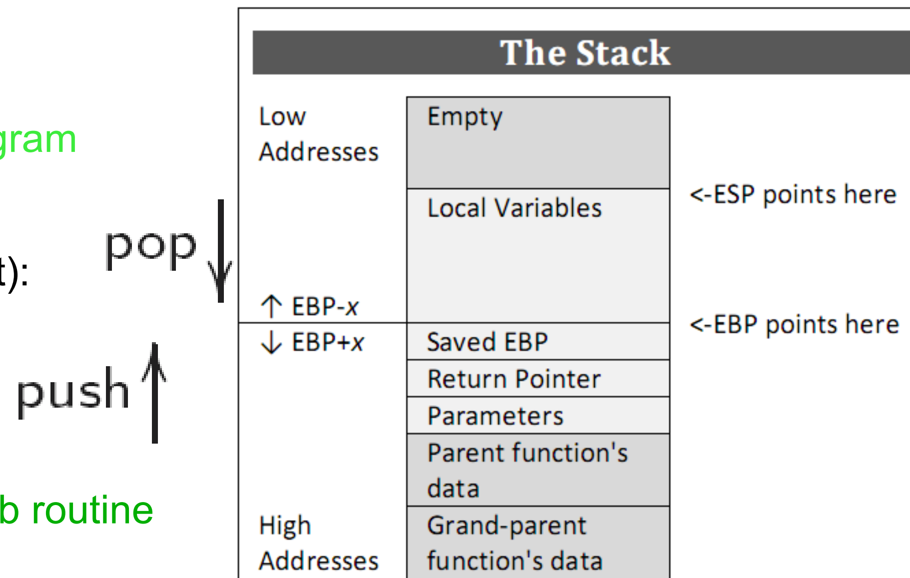
http://en.wikipedia.org/wiki/X86_calling_conventions#cdecl

- The cdecl calling convention is used by many C systems for the x86 architecture. In cdecl, function parameters are pushed on the stack in a right-to-left order.
  - Function return values are returned in the EAX register (except for floating point values, which are returned in the first floating point register fp0). Registers EAX, ECX, and EDX are available for use in the function.
- For instance, the following C code function prototype and function call:

```
int func(int, int, int);
int a, b, c, x;
…
x = func(a, b, c);     // somewhere else in the program
```

Will produce the following x86 Assembly code (written in MASM syntax, with destination first):

```
push c
push b
push a
call func ; We goto the label "func:" assembly sub routine
add esp, 12 ; Stack cleaning
mov x, eax ; EAX will be set in sub
```



- The calling function "cleans" the stack after the function call returns

RCE and calling conventions

- The main differences between the calling conventions
  - **__cdecl** is the default calling convention for C and C++ programs. The advantage of this calling convention is that it allows functions with a variable number of arguments to be used.
    Example: int printf (const char * format, ...); // the dots ...
    - Stack cleanup is performed by the caller
  - **__stdcall** is used to call Win32 API functions. It does not allow functions to have a variable number of arguments
    - Stack cleanup is performed by the called function
  - **__fastcall** attempts to put arguments in registers, rather than on the stack, thus making function calls faster
  - **Thiscall** calling convention is the default calling convention used by C++ member functions that do not use variable arguments
    - Stack cleanup is performed by the called function
- ”Calling Conventions Demystified” - stack, functions prolog and epilog
  - http://www.codeproject.com/KB/cpp/calling_conventions_demystified.aspx

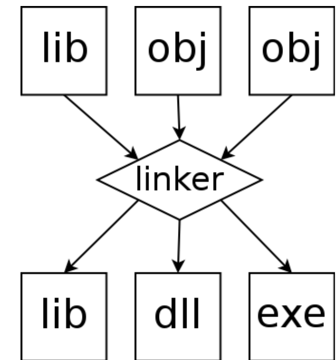
Executable formats

- Executable file formats
  - http://en.wikipedia.org/wiki/Category:Executable_file_formats
  - ELF32/64, PE32/64, COFF32/64 (.exe, executable rights)
  - Object code (.o)
  - Shared libraries (.dll, .so)
- Different versions of the application
  - Source code
  - Debug binary
    - Contains debug info
  - Regular binary
    - Dynamic linked libraries
  - Regular binary
    - Static linked libraries
  - Stripped binary
    - Symbols are removed

Going from source code to a binary executable

Executable file formats

- Symbols
  - Defined symbols, which allow it to be called by other modules
  - Undefined symbols, which call other modules where these symbols are defined
  - Local symbols, used internally within the object file to facilitate relocation
- Linker
  - Linking of libs and obj files resolving symbols
  - Arranging objects in programs address space
  - Relocation of code



- What is relocation?
  - Combine all the objects sections like .code (.text), .data, .bss, etc. to a single executable
  - Replacing symbolic references or names of libraries with actual usable (runnable) addresses in memory

PE basic concepts

There are 4 ways to refer to a location in a PE image

- Executable File Offset from beginning of the file/image (on disk)
- Relative Virtual Address (RVA)
  - Offset from the base address once the image has been mapped into memory (RAM), RVA = target address – base address
  - Various sections needs to be aligned which creates memory holes in memory (less present in the file), **called code caves**
- Section (or view) offset
  - This is the offset from the data structure you currently are in
- Virtual address
  - This is a full pointer to the address space of the process in memory
  - VA = RVA + base address
- For almost all executables the image base address is 0x400000
- For DLLs the base address can vary since it can collide with other DLLs

Note! Base address == load address and target address == Virtual Address

X86_Win32_Reverse_Engineering_Cheat_Sheet.pdf

Terminology and Formulas

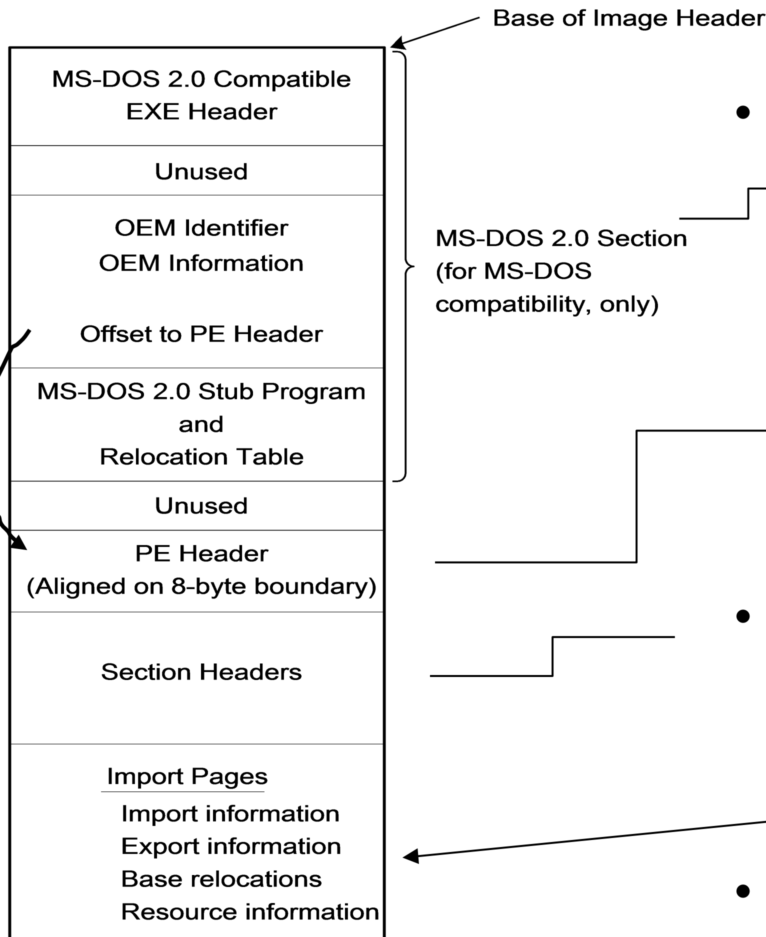
Pointer to Raw Data	Offset of section data within the executable file.
Size of Raw Data	Amount of section data within the executable file.
RVA	Relative Virtual Address. Memory offset from the beginning of the executable.
Virtual Address (VA)	Absolute Memory Address (RVA + Base). The PE Header fields named VirtualAddress actually contain Relative Virtual Addresses.
Virtual Size	Amount of section data in memory.
Base Address	Offset in memory that the executable module is loaded.
ImageBase	Base Address requested in the PE header of a module.
Module	An PE formatted file loaded into memory. Typically EXE or DLL.
Pointer	A memory address
Entry Point	The address of the first instruction to be executed when the module is loaded.
Import	DLL functions required for use by an executable module.
Export	Functions provided by a DLL which may be Imported by another module.
RVA->Raw Conversion	Raw = (RVA - SectionStartRVA) + (SectionStartRVA - SectionStartPtrToRaw)
RVA->VA Conversion	VA = RVA + BaseAddress
VA->RVA Conversion	RVA = VA - BaseAddress
Raw->VA Conversion	VA = (Raw - SectionStartPtrToRaw) + (SectionStartRVA + ImageBase)

Microsoft PE format

Microsoft Portable Executable and Common Object File Format Specification

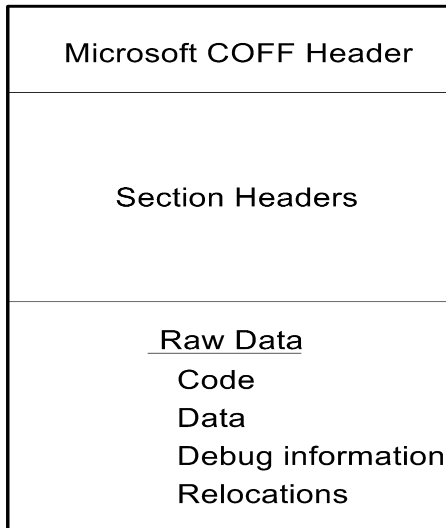
http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

File offset and RVA 0



- Portable EXE File Layout
  - Not architecture specific
- The PE file header consists of a
  - MS DOS stub (IMAGE_DOS_HEADER)
  - IMAGE_NT_HEADERS
    - The PE signature (DWORD, PE)
    - The COFF file header (IMAGE_FILE_HEADER)
    - And a **not** so optional header (IMAGE_OPTIONAL_HEADER)
- In both cases (PE and COFF), the file headers are followed immediately by a section headers table
  - Which point to .text, .data, .rdata etc.
- OpenRCE.org
  - PE Format.pdf (very good!)

Microsoft PE/COFF format



- Common Object File Format
  - PE structure is derived from COFF
- A COFF object file header consists of a
  - PE/COFF file header (IMAGE_FILE_HEADER)
  - And the optional header (IMAGE_OPTIONAL_HEADER)

PE/COFF IMAGE_FILE_HEADER

Offset	Size	Field	Description
0	2	Machine	The number that identifies the type of target machine. For more information, see section 3.3.1, "Machine Types."
2	2	NumberOfSections	The number of sections. This indicates the size of the section table, which immediately follows the headers.
4	4	TimeDateStamp	The low 32 bits of the number of seconds since 00:00 January 1, 1970 (a C run-time time_t value), that indicates when the file was created.
8	4	PointerToSymbolTable	The file offset of the COFF symbol table, or zero if no COFF symbol table is present. This value should be zero for an image because COFF debugging information is deprecated.
12	4	NumberOfSymbols	The number of entries in the symbol table. This data can be used to locate the string table, which immediately follows the symbol table. This value should be zero for an image because COFF debugging information is deprecated.
16	2	SizeOfOptionalHeader	The size of the optional header, which is required for executable files but not for object files. This value should be zero for an object file. For a description of the header format, see section 3.4, "Optional Header (Image Only)."
18	2	Characteristics	The flags that indicate the attributes of the file. For specific flag values, see section 3.3.2, "Characteristics."

Microsoft PE/COFF format

- Optional header (IMAGE_OPTIONAL_HEADER)
  - Magic - 32/64 bit application
  - Address Of Entry Point (+ image base)
  - Base of Code and Data
  - Image Base
  - Subsystem, Dll Characteristics
  - Etc...
- IMAGE_DATA_DIRECTORY
  - Size and RVA to
    - [0] Export table
    - [1] Import Descriptor Table
    - [12] Import Address Table
    - Etc. 16 entries in total (10h)

```
struct _IMAGE_OPTIONAL_HEADER {
0x00   WORD Magic;
0x02   BYTE MajorLinkerVersion;
0x03   BYTE MinorLinkerVersion;
0x04   DWORD SizeOfCode;
0x08   DWORD SizeOfInitializedData;
0x0c   DWORD SizeOfUninitializedData;
0x10   DWORD AddressOfEntryPoint;
0x14   DWORD BaseOfCode;
0x18   DWORD BaseOfData;
0x1c   DWORD ImageBase;
0x20   DWORD SectionAlignment;
0x24   DWORD FileAlignment;
0x28   WORD MajorOperatingSystemVersion;
0x2a   WORD MinorOperatingSystemVersion;
0x2c   WORD MajorImageVersion;
0x2e   WORD MinorImageVersion;
0x30   WORD MajorSubsystemVersion;
0x32   WORD MinorSubsystemVersion;
0x34   DWORD Win32VersionValue;
0x38   DWORD SizeOfImage;
0x3c   DWORD SizeOfHeaders;
0x40   DWORD CheckSum;
0x44   WORD Subsystem;
0x46   WORD DllCharacteristics;
0x48   DWORD SizeOfStackReserve;
0x4c   DWORD SizeOfStackCommit;
0x50   DWORD SizeOfHeapReserve;
0x54   DWORD SizeOfHeapCommit;
0x58   DWORD LoaderFlags;
0x5c   DWORD NumberOfRvaAndSizes;
0x60   _IMAGE_DATA_DIRECTORY DataDirectory[16];
};
```

- An In-Depth Look into the Win32 Portable Executable File Format
  - http://msdn.microsoft.com/en-us/magazine/cc301805.aspx

Import tables (IDT, IAT) 1

- Index 1 in IMAGE_DATA_DIRECTORY, the IMAGE_DIRECTORY_ENTRY_IMPORT have a RVA to where the IMAGE_IMPORT_DESCRIPTOR array begins (Import Directory Table)
- The array contains all DLLs (Name) the PE file is linked against
- The last import descriptor have all fields zeroed - marks array end
- OriginalFirstThunk is a RVA as is Name (of DLL) and FirstThunk

As the comments indicate, there are in fact two import tables for each DLL

Where the non-zero (union), OriginalFirstThunk refers to the "Unbound" Import Table

FirstThunk, on the other hand, refers to the ”bound” Import Address Table (IAT)

```
struct _IMAGE_IMPORT_DESCRIPTOR {
0x00  union {
             /* 0 for terminating null import descriptor  */
0x00         DWORD      Characteristics;
             /* RVA to original unbound IAT */
0x00         PIMAGE_THUNK_DATA OriginalFirstThunk;
      } u;
0x04  DWORD      TimeDateStamp;    /* 0 if not bound,
                                    * -1 if bound, and real date\time stamp
                                    *   in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT
                                    * (new BIND)
                                    * otherwise date/time stamp of DLL bound to
                                    * (Old BIND)
                                    */
0x08  DWORD      ForwarderChain;   /* -1 if no forwarders */
0x0c  DWORD      Name;
      /* RVA to IAT (if bound this IAT has actual addresses) */
0x10  PIMAGE_THUNK_DATA FirstThunk;
};
```

Import tables (IDT, IAT) 2

- The IMAGE_THUNK_DATA data structure describes both of these import tables - note that its type is a union
  - Usually the two tables: Import Name Table (OriginalFirstThunk) and Import Address Table (FirstThunk) looks exactly the same at rest (on disk)
- IAT is rewritten when loaded into memory and points to the actual addresses of imported functions
- The "Unbound" OriginalFirstThunk Import Name Table remains unchanged

An imported function can be listed (imported) by name (MSB=0), or it can be listed (imported) by an ordinal number (MSB=1) which represents its position in the DLL's export table

If imported by ordinal the remaining 31 bits corresponds to the ordinal number

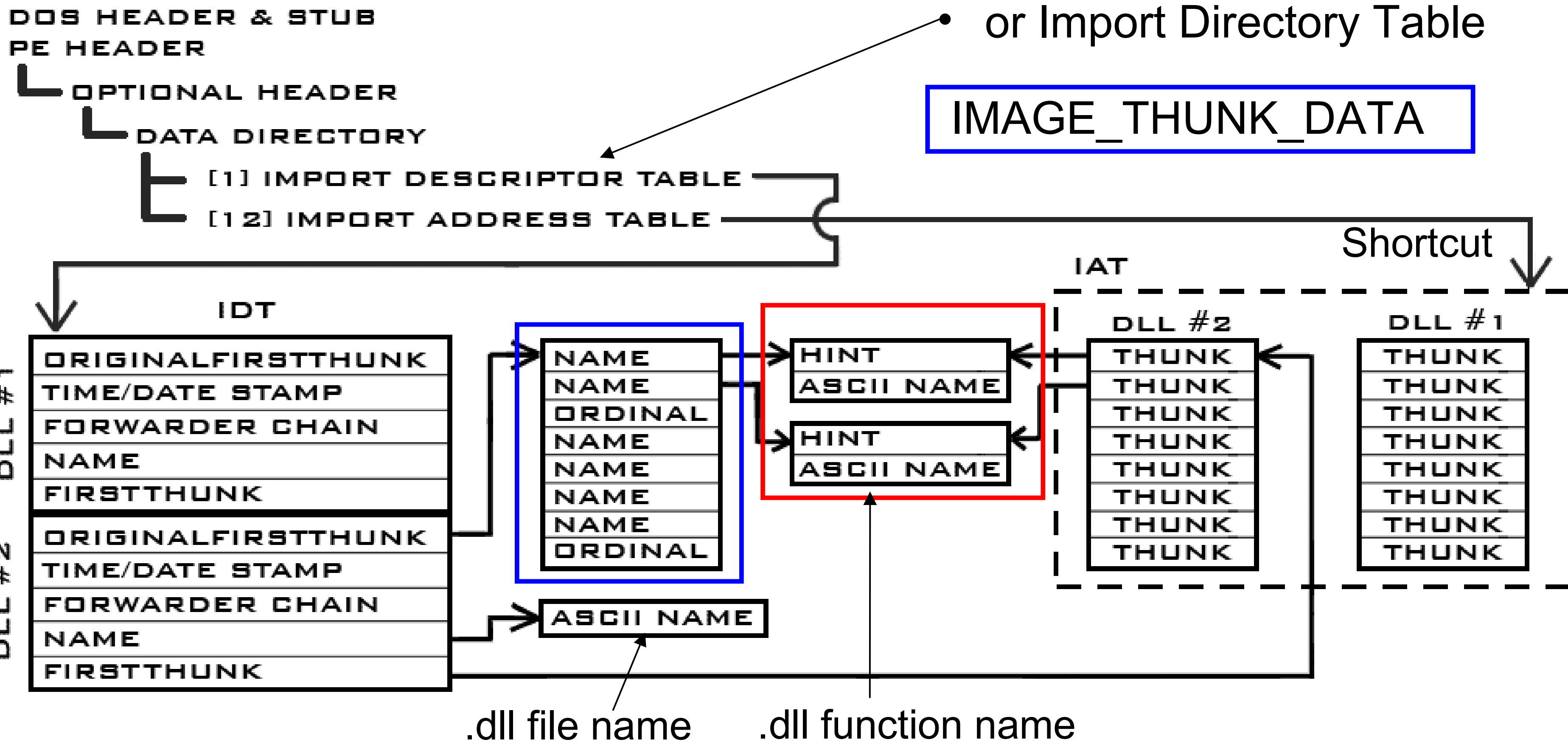
If imported by name the remaining bits are a RVA to IMAGE_IMPORT_BY_NAME

```
typedef struct _IMAGE_THUNK_DATA {
        union {
0x00            LPBYTE    ForwarderString;
0x00            PDWORD    Function;
0x00            DWORD     Ordinal;
0x00            PIMAGE_IMPORT_BY_NAME     AddressOfData;
        } u1;
} IMAGE_THUNK_DATA,*PIMAGE_THUNK_DATA;
```

```
typedef struct _IMAGE_IMPORT_BY_NAME {
0x00   WORD         Hint;
0x02   BYTE  Name[1];
} IMAGE_IMPORT_BY_NAME,*PIMAGE_IMPORT_BY_NAME;
```

Import Roadmap

- IMAGE_IMPORT_BY_NAME
  - The Hint can help the OS PE loader to look up the functions faster
  - The Name is terminated by at least one zero (needs proper alignment)



Definition: http://thunk.org/

So what's a thunk?

First a word on thunks, and why we have an IAT to begin with. Because each process is contained in its own little virtual address space, and because the OS is responsible for loading a DLL into that space, a program cannot know what base virtual address a DLL is going to be loaded at when the program is compiled.

Furthermore, the may be loaded at a different address every time the program is run (relocated DLL). To fix this problem, the program doesn't call DLL functions directly. Instead, it calls the address pointed to by a known address. In assembly, ways of doing this is: (se below)

where 00408004 is a local address in the module. Ultimately address 00408004 will contain the address of the entry point for the function which we are trying to call. This mechanism is called a thunk. We put all of these "proxy" addresses together into a thunk table when we compile the program so that our code never makes a direct call to an extramodular address.

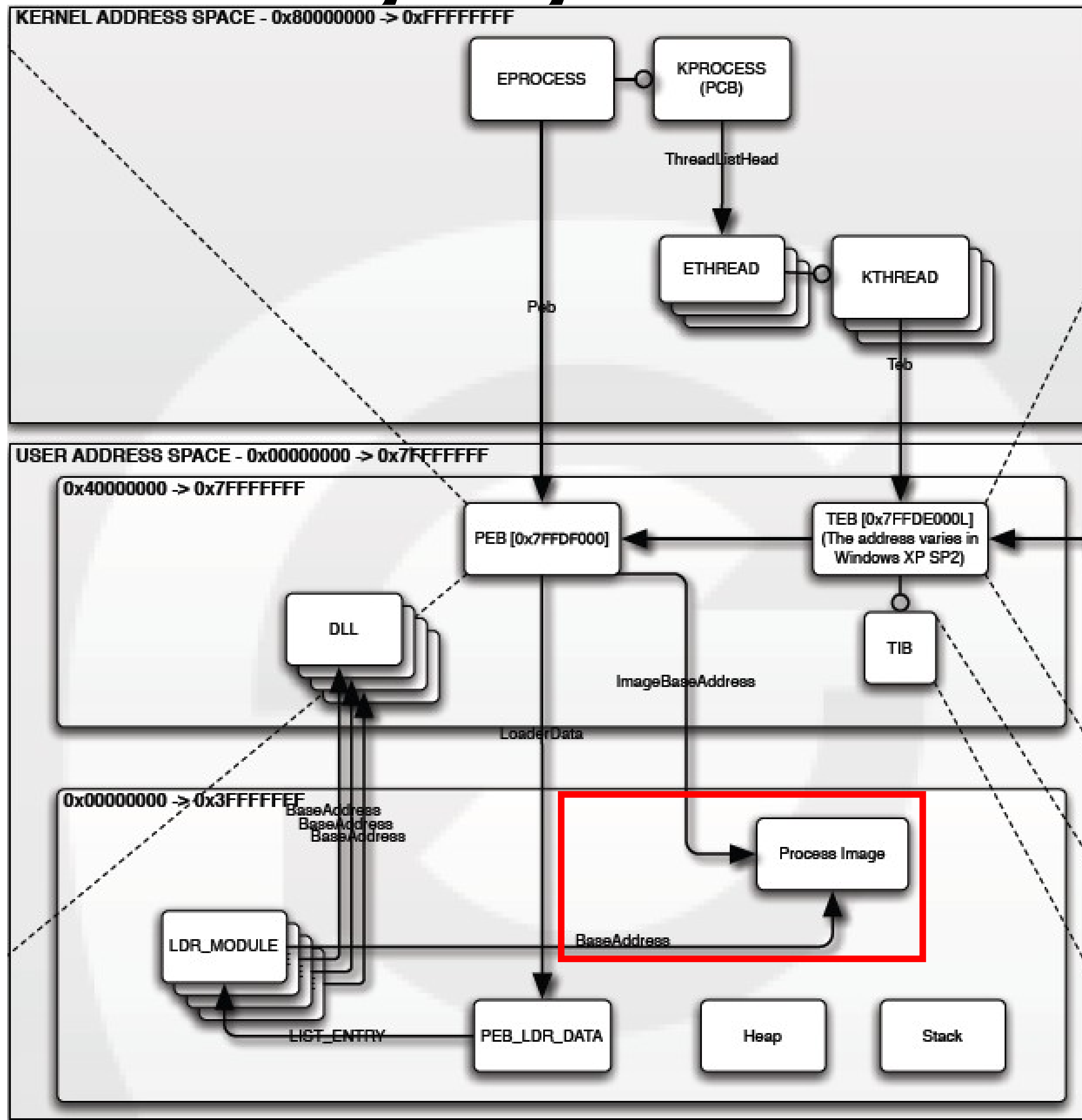
We provide the operating system with a list of all the functions we want to import, and where in the table we need their addresses to be written so that our code will wind up calling the right location at runtime. The IAT is the thunk table which the PE loader builds for us.

IAT:

```
00408000: 12 5A 36 77 ←Entry point of external function #1
00408004: 37 92 15 77 ←Entry point of external function #2
...
00401236: CALL DWORD PTR DS:[00408004]    ; call IAT direct
0040123C: CALL 004015BA                   ; second method via thunk table
...
004015B4: JMP DWORD PTR DS:[00408000]     ; jump thunk table
004015BA: JMP DWORD PTR DS:[00408004]     ; DataSegment direct@address ...
...
```

Function address
0x77159237

Memory Layout for Windows XP



Exerpt from "Windows Memory Layout, User-Kernel Address Spaces.pdf"
OpenRCE.org

Export Roadmap

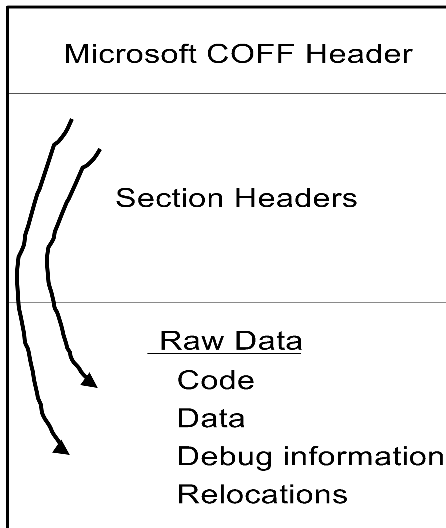
- Index 0 in IMAGE_DATA_DIRECTORY, the IMAGE_DIRECTORY_ENTRY_EXPORT have a RVA to where the IMAGE_EXPORT_DIRECTORY array begins

```
struct _IMAGE_EXPORT_DIRECTORY {
0x00  DWORD   Characteristics;
0x04  DWORD   TimeDateStamp;
0x08  WORD    MajorVersion;
0x0a  WORD    MinorVersion;
0x0c  DWORD   Name;
0x10  DWORD   Base;
0x14  DWORD   NumberOfFunctions;
0x18  DWORD   NumberOfNames;
0x1c  DWORD   AddressOfFunctions;
0x20  DWORD   AddressOfNames;
0x24  DWORD   AddressOfNameOrdinals;
};
```


If a symbol N is exported by ordinal and name then:
-Its name will be located at **AddressOfNames[N]**
-Its ordinal at **AddressOfNameOrdinals[N]**
-And its address* will be **AddressOfFunctions[*AddressOfNameOrdinals[N]*]**

The function might be forwarded, in that case the last pointer will refer to an address within the exports pointing to the forwarder string, which will contain information on the symbol and the module where to find it.

Microsoft PE/COFF format



- Section header

Offset	Size	Field	Description
0	8	Name	An 8-byte, null-padded UTF-8 encoded string. If the string is exactly 8 characters long, there is no terminating null. For longer names, this field contains a slash (/) that is followed by an ASCII representation of a decimal number that is an offset into the string table. Executable images do not use a string table and do not support section names longer than 8 characters. Long names in object files are truncated if they are emitted to an executable file.
8	4	VirtualSize	The total size of the section when loaded into memory. If this value is greater than SizeOfRawData, the section is zero-padded. This field is valid only for executable images and should be set to zero for object files.
12	4	VirtualAddress	For executable images, the address of the first byte of the section relative to the image base when the section is loaded into memory. For object files, this field is the address of the first byte before relocation is applied; for simplicity, compilers should set this to zero. Otherwise, it is an arbitrary value that is subtracted from offsets during relocation.
16	4	SizeOfRawData	The size of the section (for object files) or the size of the initialized data on disk (for image files). For executable images, this must be a multiple of FileAlignment from the optional header. If this is less than VirtualSize, the remainder of the section is zero-filled. Because the SizeOfRawData field is rounded but the VirtualSize field is not, it is possible for SizeOfRawData to be greater than VirtualSize as well. When a section contains only uninitialized data, this field should be zero.
20	4	PointerToRawData	The file pointer to the first page of the section within the COFF file. For executable images, this must be a multiple of FileAlignment from the optional header. For object files, the value should be aligned on a 4-byte boundary for best performance. When a section contains only uninitialized data, this field should be zero.
24	4	PointerToRelocations	The file pointer to the beginning of relocation entries for the section. This is set to zero for executable images or if there are no relocations.
28	4	PointerToLinenumbers	The file pointer to the beginning of line-number entries for the section. This is set to zero if there are no COFF line numbers. This value should be zero for an image because COFF debugging information is deprecated.
32	2	NumberOfRelocations	The number of relocation entries for the section. This is set to zero for executable images.
34	2	NumberOfLinenumbers	The number of line-number entries for the section. This value should be zero for an image because COFF debugging information is deprecated.
36	4	Characteristics	The flags that describe the characteristics of the section. For more information, see section 4.1, "Section Flags."

- N sections headers point out where code, data, resources etc. are stored
- Characteristics – sections flags RWX etc.
- Name can be set by programmer
- RVA = Relative Virtual Address
- Virtual (or target) Address = RVA + Load (or Base) address

PEview - cons.exe

Offset type

pFile = offset to data/value in file

PEview - C:\data\asm\cons.exe

File View Go Help

- cons.exe
  - IMAGE_DOS_HEADER
  - MS-DOS Stub Program
  - IMAGE_NT_HEADERS
    - Signature
    - IMAGE_FILE_HEADER
    - IMAGE_OPTIONAL_HEADER
  - IMAGE_SECTION_HEADER .text
  - IMAGE_SECTION_HEADER .rdata
  - IMAGE_SECTION_HEADER .data
  - SECTION .text
  - SECTION .rdata
    - IMPORT Address Table
    - IMPORT Directory Table
    - IMPORT Name Table
    - IMPORT Hints/Names & DLL Names
  - SECTION .data

pFile	Data	Description	
000001B8	2E 74 65 78	Name	
000001BC	74 00 00 00		
000001C0	0000003E	Virtual Size	
000001C4	00001000	RVA	
000001C8	00000200	Size of Raw Data	
000001CC	00000400	Pointer to Raw Data	
000001D0	00000000	Pointer to Relocations	
000001D4	00000000	Pointer to Line Numbers	
000001D8	0000	Number of Relocations	
000001DA	0000	Number of Line Numbers	
000001DC	60000020	Characteristics	
		00000020	IMAGE_SCN_CNT_CODE
		20000000	IMAGE_SCN_MEM_EXECUTE
		40000000	IMAGE_SCN_MEM_READ

offset to .text/.code in memory

offset to .text/.code in file

Viewing IMAGE_SECTION_HEADER .text

PE File: Unmapped Data, .reloc section, other sections, .data section, .text section, Section Table, PE Header, DOS Header — Higher offsets

In Memory: other sections, .data section, .text section, Section Table, PE Header, DOS Header — Higher addresses

(target address) 0x401000 - (load address) 0x400000 = (RVA) 0x1000

PEview - cons.exe

- IMAGE_NT_HEADERS
  - Signature
  - IMAGE_FILE_HEADER
  - IMAGE_OPTIONAL_HEADER

pFile	Data	Description	Value
0000013C	00000000	Size	
00000140	00002010	RVA	IMPORT Table
00000144	00000028	Size	
00000148	00000000	RVA	RESOURCE Table

- SECTION .rdata
  - IMPORT Address Table
  - IMPORT Directory Table
  - IMPORT Name Table
  - IMPORT Hints/Names & DLL Names

RVA	Data	Description	Value
00002010	00002038	Import Name Table RVA	
00002014	00000000	Time Date Stamp	
00002018	00000000	Forwarder Chain	
0000201C	00002076	Name RVA	KERNEL32.dll
00002020	00002000	Import Address Table RVA	

IAT is equal to Import Name Table on disk

- SECTION .rdata
  - IMPORT Address Table
  - IMPORT Directory Table
  - IMPORT Name Table
  - IMPORT Hints/Names & DLL Names

RVA	Data	Description	Value
00002038	00002058	Hint/Name RVA	0482 WriteConsoleA
0000203C	00002068	Hint/Name RVA	0104 ExitProcess
00002040	00002048	Hint/Name RVA	023B GetStdHandle
00002044	00000000	End of Imports	KERNEL32.dll

- SECTION .rdata
  - IMPORT Address Table
  - IMPORT Directory Table
  - IMPORT Name Table
  - IMPORT Hints/Names & DLL Names

RVA	Raw Data	Value
00002048	3B 02 47 65 74 53 74 64 48 61 6E 64 6C 65 00 00	;.GetStdHandle..
00002058	82 04 57 72 69 74 65 43 6F 6E 73 6F 6C 65 41 00	..WriteConsoleA.
00002068	04 01 45 78 69 74 50 72 6F 63 65 73 73 00 4B 45	..ExitProcess.KE
00002078	52 4E 45 4C 33 32 2E 64 6C 6C 00	RNEL32.dll.

PEview - cons.exe

Address	Hex dump	Disassembly	Comment
00401000	$ 55	PUSH EBP	
00401001	. 8BEC	MOV EBP,ESP	
00401003	. 83C4 F8	ADD ESP,-8	
00401006	. 6A F5	PUSH -0B	DevType = STD_OUTPUT_HANDLE
00401008	. E8 1F000000	CALL <JMP.&KERNEL32.GetStdHandle>	GetStdHandle
0040100D	. 8945 FC	MOV DWORD PTR SS:[EBP-4],EAX	
00401010	. 8D55 F8	LEA EDX,DWORD PTR SS:[EBP-8]	
00401013	. 6A 00	PUSH 0	pReserved = NULL
00401015	. 52	PUSH EDX	pWritten
00401016	. 6A 0F	PUSH 0F	CharsToWrite = F (15.)
00401018	. 68 00304000	PUSH cons.00403000	Buffer = cons.00403000
0040101D	. FF75 FC	PUSH DWORD PTR SS:[EBP-4]	hConsole
00401020	. E8 0D000000	CALL <JMP.&KERNEL32.WriteConsoleA>	WriteConsoleA
00401025	. 6A 00	PUSH 0	ExitCode = 0
00401027	. E8 0C000000	CALL <JMP.&KERNEL32.ExitProcess>	ExitProcess
0040102C	$-FF25 08204000	JMP DWORD PTR DS:[<&KERNEL32.GetStdHand	kernel32.GetStdHandle
00401032	$-FF25 00204000	JMP DWORD PTR DS:[<&KERNEL32.WriteConso	kernel32.WriteConsoleA
00401038	.-FF25 04204000	JMP DWORD PTR DS:[<&KERNEL32.ExitProces	kernel32.ExitProcess

- cons.exe
  - IMAGE_DOS_HEADER
  - MS-DOS Stub Program
  - IMAGE_NT_HEADERS
  - IMAGE_SECTION_HEADER .text
  - IMAGE_SECTION_HEADER .rdata
  - IMAGE_SECTION_HEADER .data
  - SECTION .text

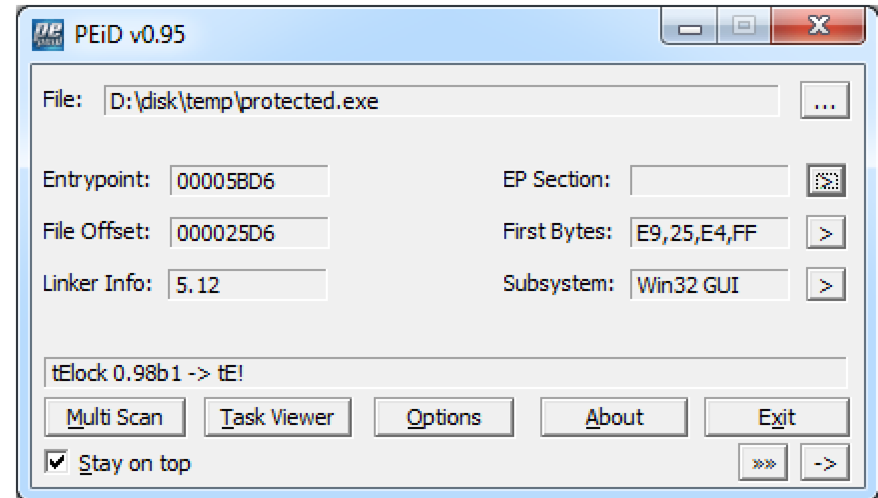
RVA	Raw Data	Value
00001000	55 8B EC 83 C4 F8 6A F5 E8 1F 00 00 00 89 45 FC	U.....j.......E.
00001010	8D 55 F8 6A 00 52 6A 0F 68 00 30 40 00 FF 75 FC	.U.j.Rj.h.0@..u.
00001020	E8 0D 00 00 00 6A 00 E8 0C 00 00 00 FF 25 08 20	.....j.......%.
00001030	40 00 FF 25 00 20 40 00 FF 25 04 20 40 00 00 00	@..%. @..%. @...
00001040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00001050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00001060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....

- cons.exe
  - IMAGE_DOS_HEADER
  - MS-DOS Stub Program
  - IMAGE_NT_HEADERS
  - IMAGE_SECTION_HEADER .text
  - IMAGE_SECTION_HEADER .rdata
  - IMAGE_SECTION_HEADER .data
  - SECTION .text
  - SECTION .rdata
  - SECTION .data

RVA	Raw Data	Value
00003000	48 65 6C 6C 6F 2C 20 77 6F 72 6C 64 21 0D 0A 00	Hello, world!...
00003010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00003020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00003030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00003040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00003050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00003060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00003070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00003080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....

PE/COFF tools...

- Dependency Walker
- PEiD
- PE.explorer
- PETools
- ProcDump32
- LordPE
- PEdump
- PEview
- Periscope
- FileAlyzer
- 7zip can dump PE/COFF sections to files (.data, .text etc.)
- Perl (ch6 WFA)
  - Pedmp.pl
  - Fvi.pl (resources)



Section Viewer

Name	V. Offset	V. Size	R. Offset	R. Size	Flags
.text	00001000	00001000	00000400	00000200	C0000040
.rdata	00002000	00001000	00000600	00000200	C0000040
.data	00003000	00001000	00000800	00000200	C0000040
	00004000	00003000	00000A00	00002200	C0000040

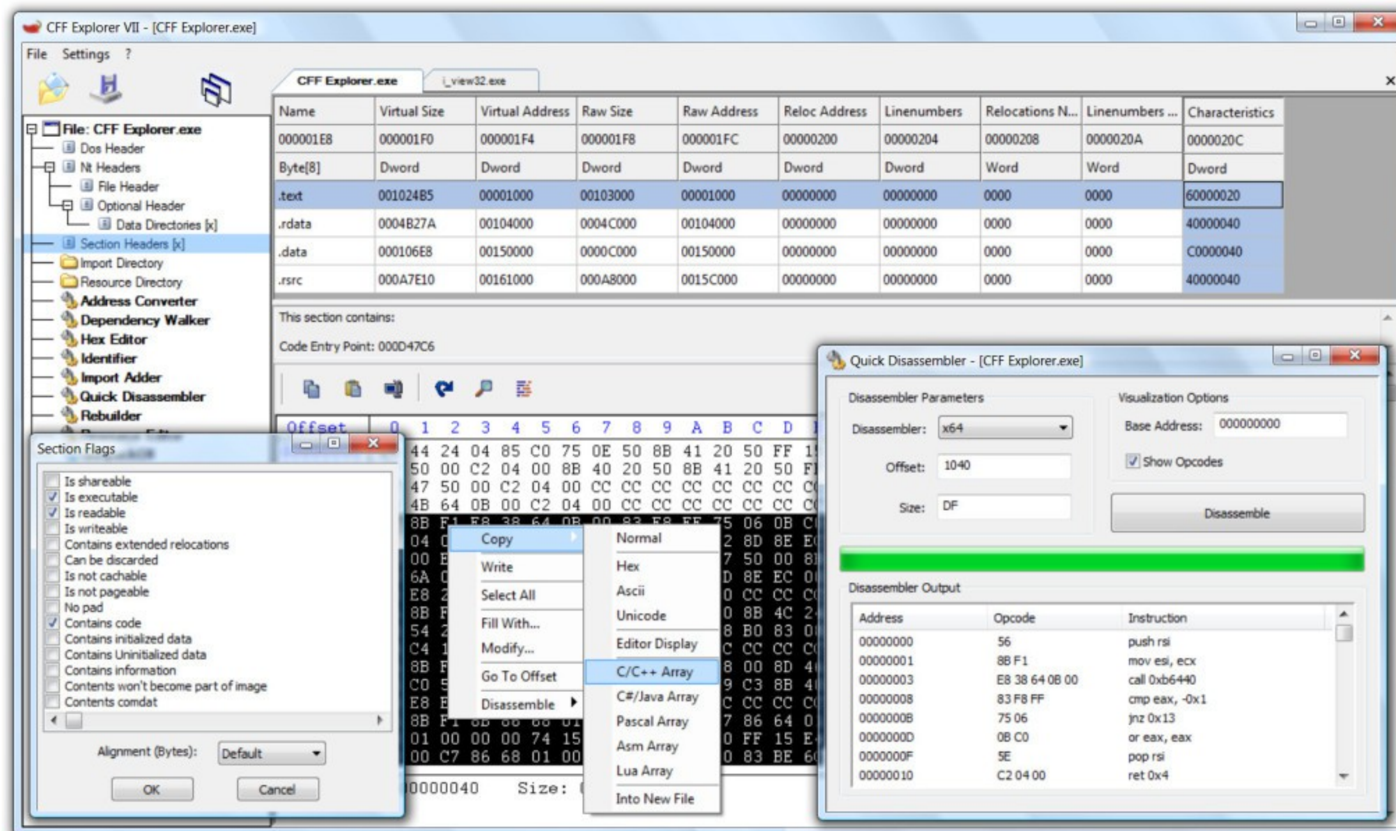
Close

Address of entry point (EP) should be located in .text or .code

CFF Explorer

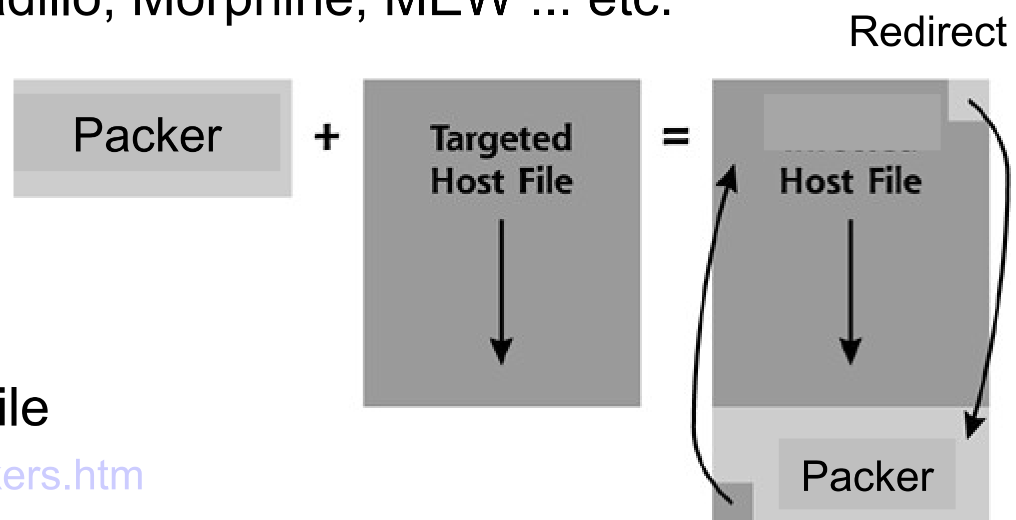
A freeware suite of tools. The PE editor has full support for PE32/64. Special fields description and modification (.NET supported), utilities, rebuilder, hex editor, import adder, signature scanner, signature manager, extension support, scripting, disassembler, dependency walker etc. The suite is available for x86, x64 and Itanium.

http://www.ntcore.com/exsuite.php



Executable (PE/COFF) obfuscation

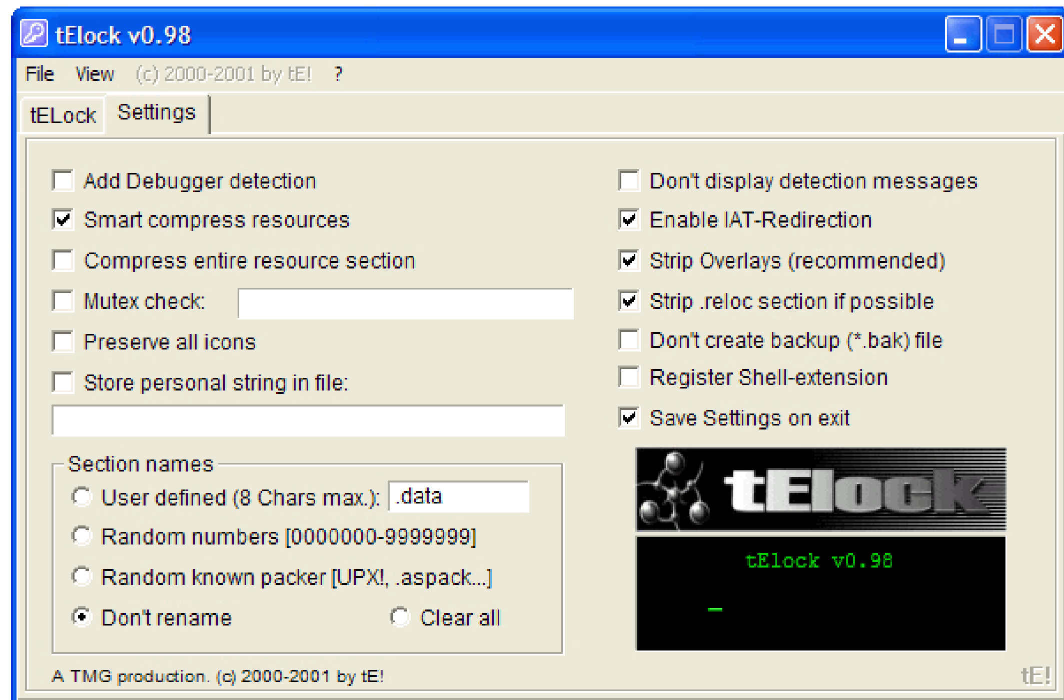
- Binders
  - Bind two applications into one, mainly used for trojans
- Packers or compressors
  - Compress the binarys sections to make it smaller and harder to detect and analyse
  - Works much like a virus appending an application and when unpacked in memory the entry point is reset to original
  - ASPack, UPX, FSG, Armadillo, Morphine, MEW ... etc.
  - Scan for section names indicating a packer
  - Special tools is neded to unpack the binary, then dump and rebuild it Image dump != MS .dmp file



http://www.woodmann.com/crackz/Packers.htm

Executable (PE/COFF) obfuscation

- Cryptors
  - As packers but with encrypted sections usually with anti-disassembly and anti-debugging techniques, also
  - Rebuilding the import address tables at runtime
- Example: tElock
- Crackers Kit v2.0/3.2
  - Large packages with tools as:
  - Packers
  - Unpackers
  - Rebuilding
  - Analysis
  - Patchers
  - Google for it ...



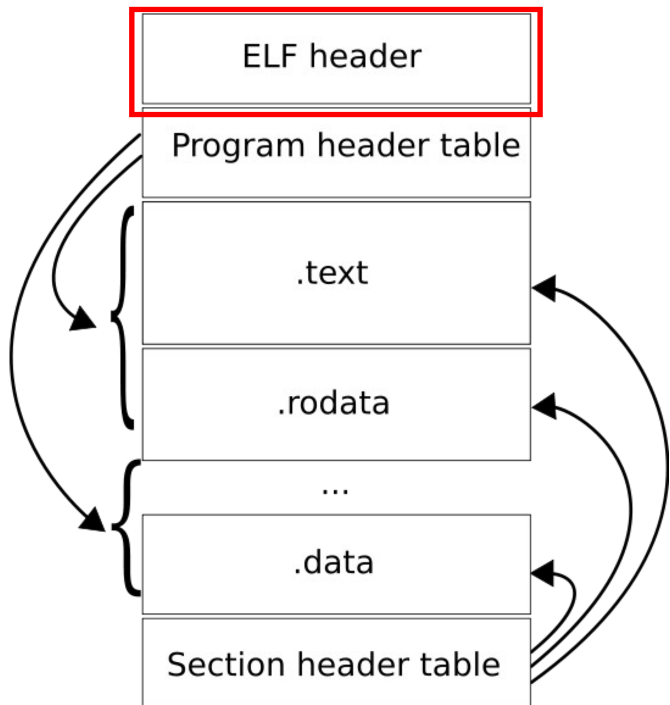
ELF (Executable and Linking Format)

- ELF header

Tells us basic info and where everything is located in the file

Can be read directly from the first e_ehsize (default: 52) bytes of the file

Fields of interest: e_entry, e_phoff, e_shoff, and the sizes given. e_entry specifies the location of _start, e_phoff shows us where the array of program headers lies in relation to the start of the executable, and e_shoff shows us the same for the section headers




```
/* ELF File Header */
typedef struct
{
  unsigned char e_ident[EI_NIDENT];     /* Magic number and other info */
  Elf32_Half    e_type;                 /* Object file type */
  Elf32_Half    e_machine;              /* Architecture */
  Elf32_Word    e_version;              /* Object file version */
  Elf32_Addr    e_entry;                /* Entry point virtual address */
  Elf32_Off     e_phoff;                /* Program header table file offset */
  Elf32_Off     e_shoff;                /* Section header table file offset */
  Elf32_Word    e_flags;                /* Processor-specific flags */
  Elf32_Half    e_ehsize;               /* ELF header size in bytes */
  Elf32_Half    e_phentsize;            /* Program header table entry size */
  Elf32_Half    e_phnum;                /* Program header table entry count */
  Elf32_Half    e_shentsize;            /* Section header table entry size */
  Elf32_Half    e_shnum;                /* Section header table entry count */
  Elf32_Half    e_shstrndx;             /* Section header string table index */
} Elf32_Ehdr;
```

http://en.wikipedia.org/wiki/Executable_and_Linkable_Format

ELF (Executable and Linking Format)

- ELF Program segment headers
  - Describe the **segments** of the program used at run-time
  - In a typical ELF executable usually end-to-end, forming an array of structs
  - The interesting fields in this structure are p_offset, p_filesz, and p_memsz
- ELF Section headers
  - Describe various named **sections** of the binary as a file
  - Each section has an entry in the section headers array
- HT Editor (http://hte.sourceforge.net/)
  - Examine and modify everything in an ELF file (PE files also), disassemble etc.

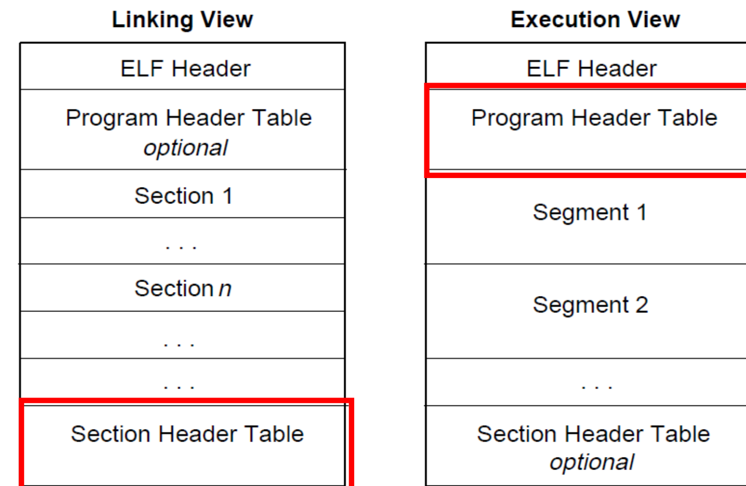
```
/* Program segment header */
typedef struct
{
  Elf32_Word   p_type;     /* Segment type */
  Elf32_Off    p_offset;   /* Segment file offset */
  Elf32_Addr   p_vaddr;    /* Segment virtual address */
  Elf32_Addr   p_paddr;    /* Segment physical address */
  Elf32_Word   p_filesz;   /* Segment size in file */
  Elf32_Word   p_memsz;    /* Segment size in memory */
  Elf32_Word   p_flags;    /* Segment flags */
  Elf32_Word   p_align;    /* Segment alignment */
} Elf32_Phdr;
```

```
/* Section header */
typedef struct
{
  Elf32_Word   sh_name;       /* Section name (string tbl index) */
  Elf32_Word   sh_type;       /* Section type */
  Elf32_Word   sh_flags;      /* Section flags */
  Elf32_Addr   sh_addr;       /* Section virtual addr at execution */
  Elf32_Off    sh_offset;     /* Section file offset */
  Elf32_Word   sh_size;       /* Section size in bytes */
  Elf32_Word   sh_link;       /* Link to another section */
  Elf32_Word   sh_info;       /* Additional section information */
  Elf32_Word   sh_addralign;  /* Section alignment */
  Elf32_Word   sh_entsize;    /* Entry size if section holds table */
} Elf32_Shdr;
```

ELF Object File Format

Some of the sections (from elf.pdf)

.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 and .data1 These sections hold initialized data that contribute to the program's memory image.

.debug This section holds information for symbolic debugging. The contents are unspecified. All section names with the prefix .debug are reserved for future use.

.dynamic This section holds dynamic linking information

.hash This section holds a symbol hash table.

.line This section holds line number information for symbolic debugging, which describes the correspondence between the source program and the machine code. The contents are unspecified.

.rodata These sections hold read-only data that typically contribute to a .rodata1 non-writable segment in the process image.

.shstrtab This section holds section names.

.strtab This section holds strings, most commonly the strings that represent the names associated with symbol table entries.

.symtab This section holds a symbol table, as "Symbol Table"

.text This section holds the "text," or executable instructions, of a program.

Sweetscape 010 editor - ELF template

010 Editor - C:\data\HDA\Digitalbrott_och_eSäkerhet\forensics_2\labs_mm\challenge

File Edit Search View Scripts Templates Tools Window Help

Edit As: Hex ELFTemplate.bt

Workspace

- Open Files
  - C:\data\HDA\...\labs_mm\challenge
- Favorite Files
- Recent Files
  - C:\Users\hjo\...\ELFTemplate.bt
- Bookmarked Files

Files Explorer

Startup challenge

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000h:	7F	45	4C	46	01	01	01	00	00	00	00	00	00	00	2D	00	.ELF.........-.
0010h:	02	00	03	00	01	00	00	00	C0	83	04	08	34	00	00	00	........Àƒ..4...
0020h:	18	22	00	00	00	00	00	00	34	00	20	00	06	00	28	00	."......4. ...(.
0030h:	22	00	1F	00	06	00	00	00	34	00	00	00	34	80	04	08	"...........4...4€..
0040h:	34	80	04	08	C0	00	00	00	C0	00	00	00	05	00	00	00	4€..À...À.......
0050h:	04	00	00	00	03	00	00	00	F4	00	00	00	F4	80	04	08	........ô...ô€..
0060h:	F4	80	04	08	13	00	00	00	13	00	00	00	04	00	00	00	ô€..............
0070h:	01	00	00	00	01	00	00	00	00	00	00	00	00	80	04	08	.............€..
0080h:	00	80	04	08	04	06	00	00	04	06	00	00	05	00	00	00	.€..............
0090h:	00	10	00	00	01	00	00	00	04	06	00	00	04	96	04	08	.............–..
00A0h:	04	96	04	08	18	01	00	00	28	01	00	00	06	00	00	00	.–......(.......
00B0h:	00	10	00	00	02	00	00	00	14	06	00	00	14	96	04	08	.............–..

Template Results - ELFTemplate.bt

Name	Value	Start	Size	Color	
struct FILE file		0h	F4h	Fg:	Bg:
struct ELF_HEADER elf_header		0h	34h	Fg:	Bg:
struct e_ident_t e_ident		0h	10h	Fg:	Bg:
enum e_type32_e e_type	ET_EXEC...	10h	2h	Fg:	Bg:
enum e_machine32_e e_machine	EM_386 (3)	12h	2h	Fg:	Bg:
enum e_version32_e e_version	EV_CUR...	14h	4h	Fg:	Bg:
Elf32_Addr e_entry_START_ADDRESS	134513600	18h	4h	Fg:	Bg:
Elf32_Off e_phoff_PROGRAM_HEADER_OFFSET_IN_FILE	52	1Ch	4h	Fg:	Bg:
Elf32_Off e_shoff_SECTION_HEADER_OFFSET_IN_FILE	8728	20h	4h	Fg:	Bg:
Elf32_Word e_flags	0	24h	4h	Fg:	Bg:
Elf32_Half e_ehsize_ELF_HEADER_SIZE	52	28h	2h	Fg:	Bg:
Elf32_Half e_phentsize_PROGRAM_HEADER_ENTRY_SIZE_IN_FILE	32	2Ah	2h	Fg:	Bg:
Elf32_Half e_phnum_NUMBER_OF_PROGRAM_HEADER_ENTRIES	6	2Ch	2h	Fg:	Bg:
Elf32_Half e_shentzise_SECTION_HEADER_ENTRY_SIZE	40	2Eh	2h	Fg:	Bg:
Elf32_Half e_shnum_NUMBER_OF_SECTION_HEADER_ENTRIES	34	30h	2h	Fg:	Bg:
Elf32_Half e_shtrndx_STRING_TABLE_INDEX	31	32h	2h	Fg:	Bg:
struct PROGRAM_HEADER_TABLE program_header_table		34h	C0h	Fg:	Bg:
struct program_table_entry32_t program_table_element[6]		34h	C0h	Fg:	Bg:
struct program_table_entry32_t program_table_element[0]		34h	20h	Fg:	Bg:
struct program_table_entry32_t program_table_element[1]		54h	20h	Fg:	Bg:
struct program_table_entry32_t program_table_element[2]		74h	20h	Fg:	Bg:
struct program_table_entry32_t program_table_element[3]		94h	20h	Fg:	Bg:
struct program_table_entry32_t program_table_element[4]		B4h	20h	Fg:	Bg:
struct program_table_entry32_t program_table_element[5]		D4h	20h	Fg:	Bg:

Inspector

Type	Value
Signed Byte	127
Unsigned Byte	127
Signed Short	17791
Unsigned Short	17791
Signed Int	1179403647
Unsigned Int	1179403647
Signed Int64	282579962709375
Unsigned Int64	282579962709375
Float	13073.37
Double	1.39613051777803e-309
String	ELF
Unicode	
DOSDATE	11/31/2014
DOSTIME	08:43:62
FILETIME	11/24/1601 01:26:36
OLETIME	
time_t	05/17/2007 12:07:27

Auto Variables Bookmarks

Selected: 52 [34h] bytes (Range: 0 [0h] to 51 [33h]) Start: 0 [0h] Sel: 52 [34h] Size: 12002 ANSI LIT W OVR

References if not given in presentation

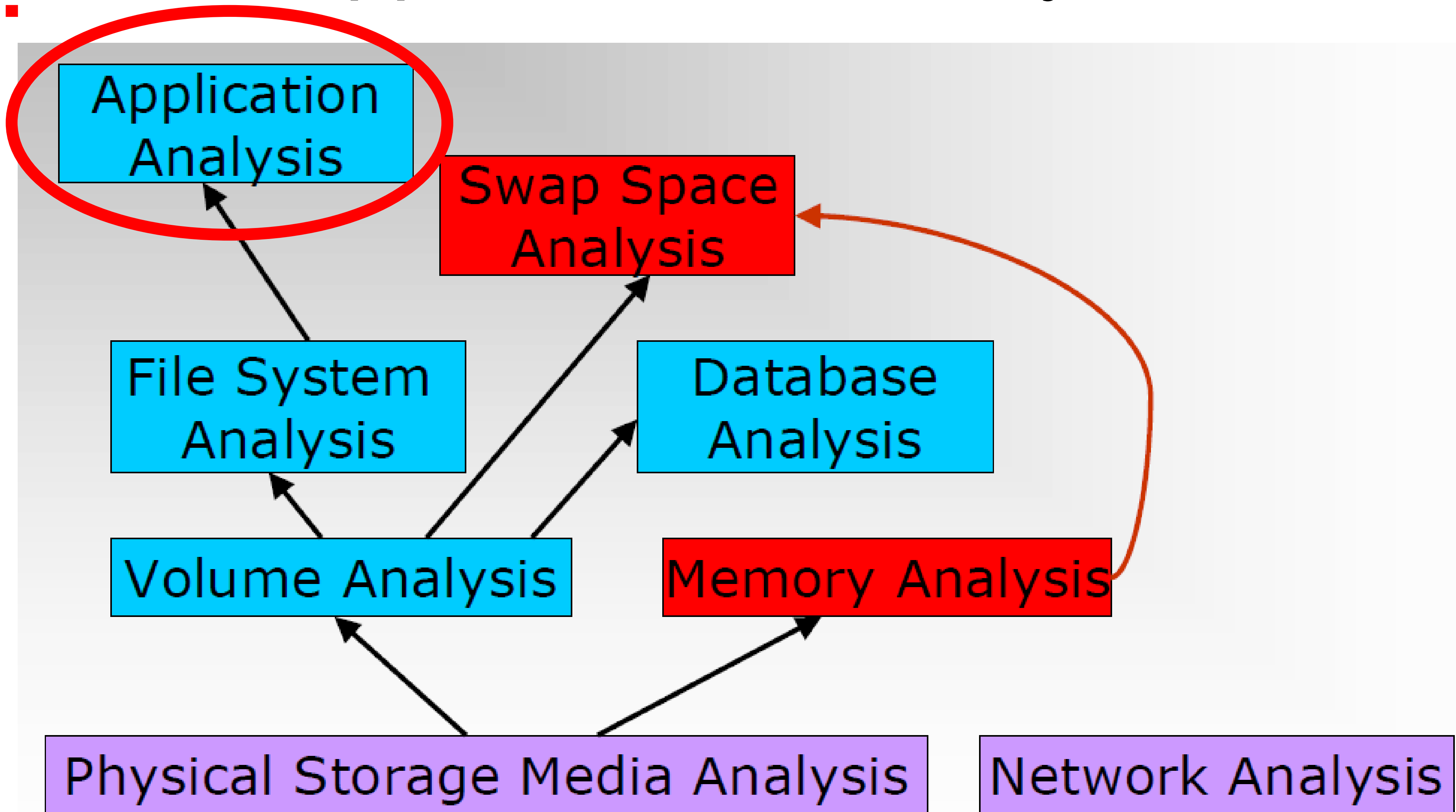
- Import Mechanisms and Intermodular Calls
  - http://www.woodmann.com/yates/documents/30.html
- Understanding the Import Address Table
  - http://sandsprite.com/CodeStuff/Understanding_imports.html
- Introduction to Reverse Engineering Software
  - http://www.acm.uiuc.edu/sigmil/RevEng/
- X86/Win32 Reverse Engineering Cheat-Sheet
  - http://www.rnicrosoft.net/
- x86 processor information
  - http://www.sandpile.org/
- Moving to Windows Vista x64 - x64 ASM, PE64, etc.
  - http://www.codeproject.com/KB/vista/vista_x64.aspx

End!

and

Backups

Application/File analysis



Programs in memory I

- When processes are loaded into memory by the OS loader, they are basically broken into many small sections. There are six main sections that we are concerned with:

- **.text or .code Section**

The .text section basically corresponds to the .text portion of the binary executable file. It contains the machine instructions to get the task done. This section is marked as read-only and will cause a segmentation fault if written to. The size is fixed at runtime when the process is first loaded.

- **.data Section**

The .data section is used to store global initialized variables such as:

int a = 0;

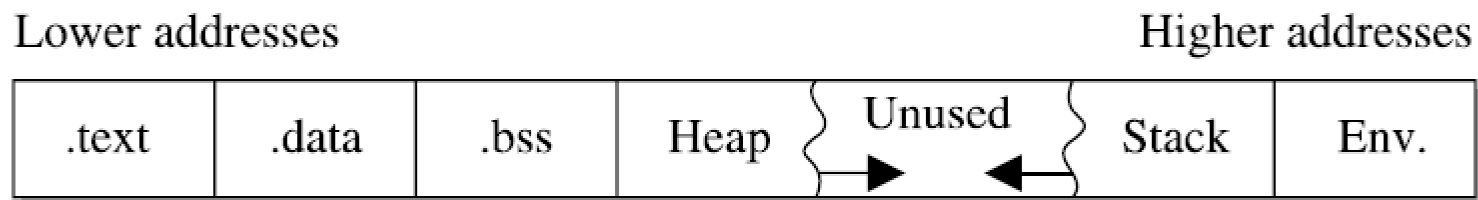
The size of this section is fixed at runtime.

- **.bss Section**

The below stack section (.bss) is used to store global non-initialized variables such as:

int a;

The size of this section is fixed at runtime.



Programs in memory II

- **Heap Section**

The heap section is used to store dynamically allocated variables and grows from the lower-addressed memory to the higher-addressed memory. The allocation of memory is controlled through the **malloc()** and **free()** functions. Example:

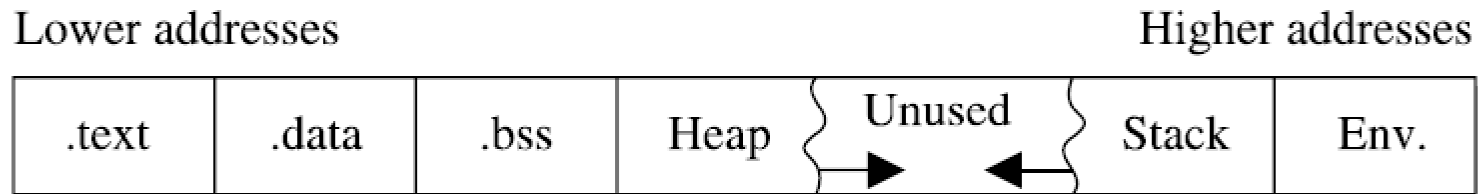
```
int i = malloc(sizeof (int));     //dynamically allocates an integer
```

- **Stack Section**

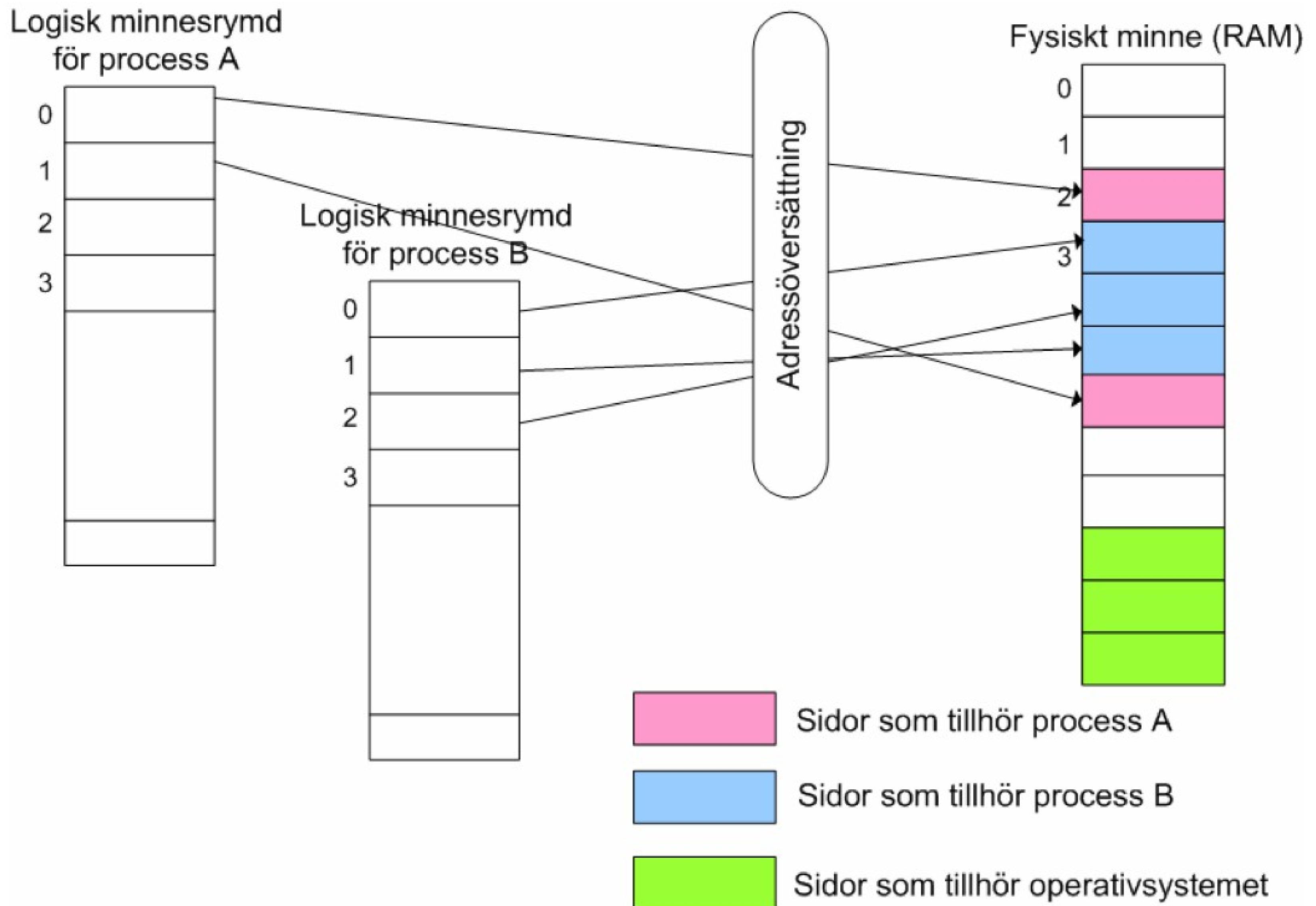
The stack section is used to keep track of function calls (recursively) and grows from the higher-addressed memory to the lower addressed memory on most systems. As we will see, the fact that the stack grows in this manner allows the subject of buffer overflows to exist. Local variables exist in the stack section.

- **Environment/Arguments Section**

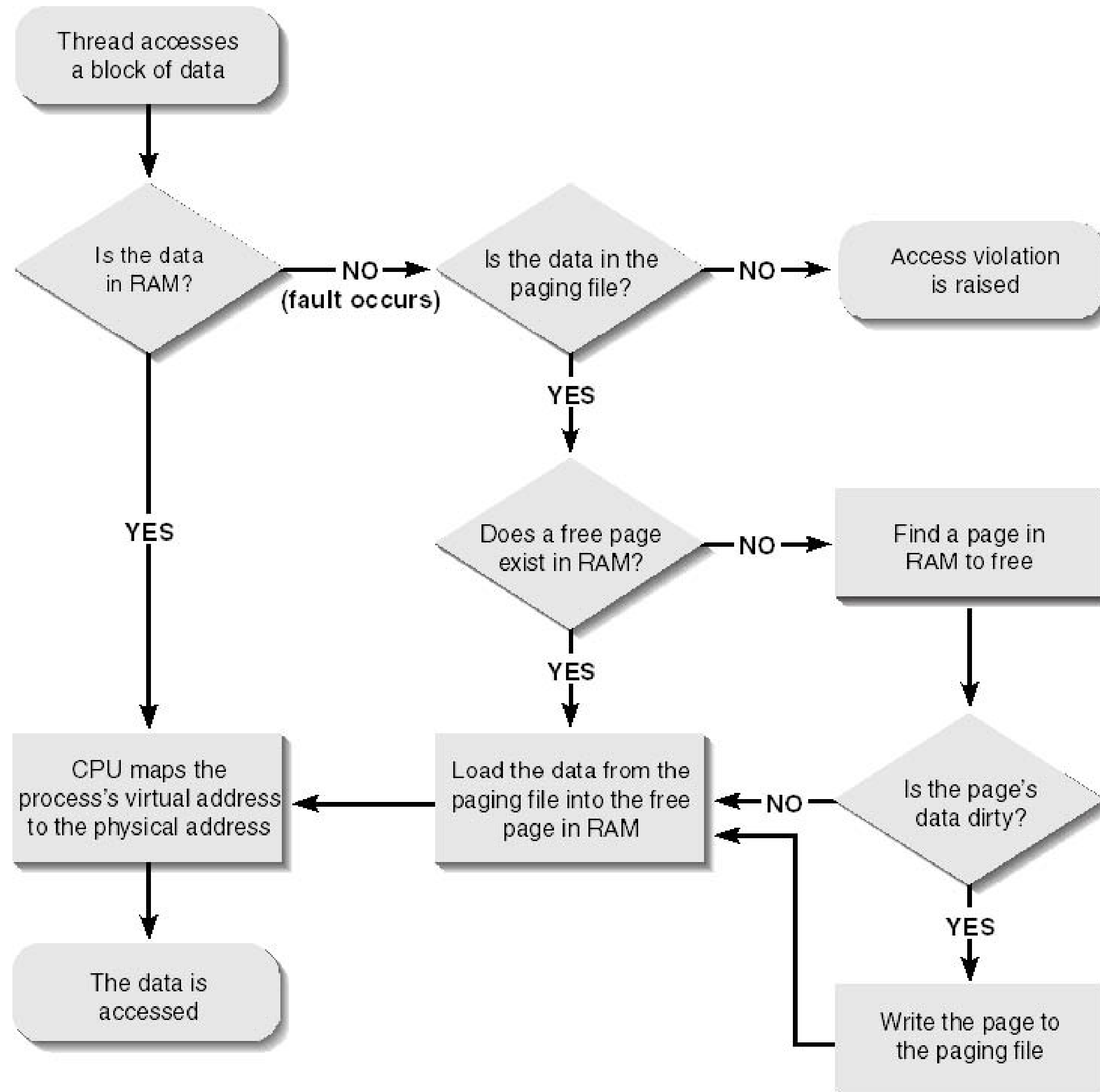
The environment/arguments section is used to store a copy of system-level variables that may be required by the process during runtime. For example, among other things, the path, shell name, and hostname are made available to the running process. This section is writable, allowing its use in format string and buffer overflow exploits. Additionally, the command-line arguments are stored in this area.



Memory management 1



Memory management 2



Translating a virtual address to a physical storage address

Link Libraries and OS relocation 1

- A dynamic link library (or shared library) takes the idea of an ordinary library (also called a statically linked library) one step further
- A dynamic/shared link library is a lot like a program, but instead of being run by the user to do one thing it has a lot of functions "exported" so that other programs can call them
  - This list, called the export table, gives the address inside the DLL file of each of the functions which the DLL allows other programs to access
  - The calling executable have a list of imports or imported functions from every DLL file it uses
- When Windows loads your program it creates a whole new "address space" for the program
- When your program contains the instruction "read memory from address 0x40A0F0 (or something like that) the computer hardware actually looks up in a table to figure out where in physical memory that location is
  - The address 0x40A0F0 in another program would mean a completely different part of the physical memory of the computer

Link Libraries and OS relocation 2

- Programs, when they are loaded, are "mapped" into address space. This process basically copies the code and static data of your program from the executable file into a certain part of address space, for example, a block of space starting at address 0x400000
  - The same thing happens when you load a DLL
- A DLL, or a program for that matter, tells the operating system what address it would prefer to be mapped into
  - Although the same address means different things to different programs, within a single program an address can only be used once
- If two DLLs wants to be mapped to the same address the OS first check if the DLL is relocateable
- If so it performs the necessary relocations
- The relocateable DLL contains information so that the OS can change/adjust all those internal function addresses in the DLL