Campus di Arcavacata
UNIVERSITÀ DELLA CALABRIA

Malware Analysis: tools and techniques of Reverse Engineering on malicious code

Elia Florio
Security Response Engineer
6th March 2006

Agenda

- Introduction
- Technical Background
  - Malware classification
  - Win32 Portable Executable Format
  - Assembly Language Basics
  - Windows API and calling convention
- Reverse Engineering
  - Methodology
  - Disassembler
  - Debugger
  - Network and Monitoring tools
  - Virtual Machines
- Common Problems
  - Executable Packers
  - Encryption
  - Anti-Debugging
  - Stealth Techniques (Rootkit)
  - Polymorphic Code
- Live Malware analysis Demo
- Questions

Agenda

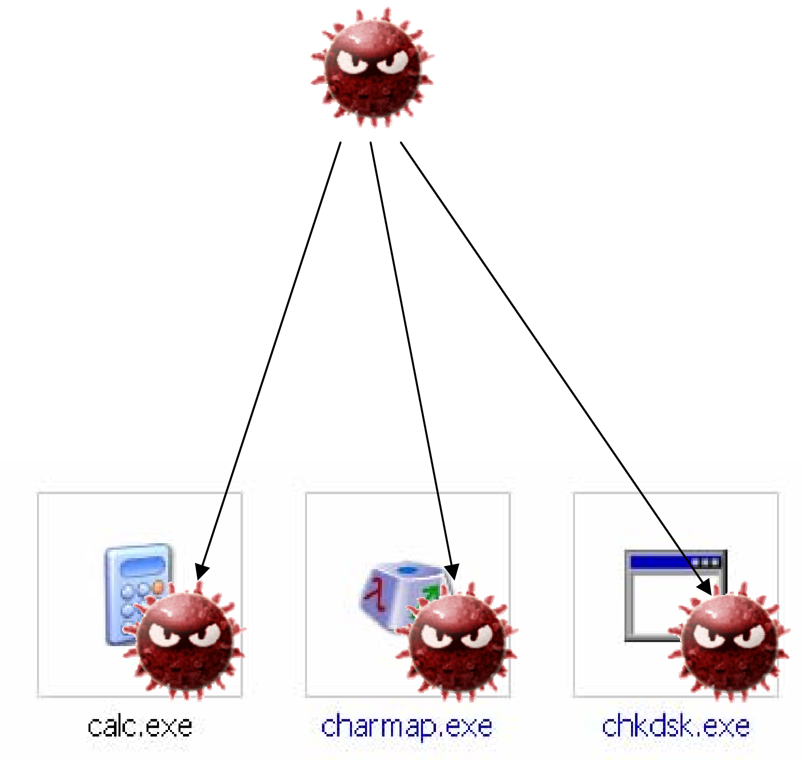
- Introduction
- Technical Background
  - Malware classification
  - Win32 Portable Executable Format
  - Assembly Language Basics
  - Windows API and calling convention
- Reverse Engineering
  - Methodology
  - Disassembler
  - Debugger
  - Network and Monitoring tools
  - Virtual Machines
- Common Problems
  - Executable Packers
  - Encryption
  - Anti-Debugging
  - Stealth Techniques (Rootkit)
  - Polymorphic Code
- Live Malware analysis Demo
- Questions

Malware Classification

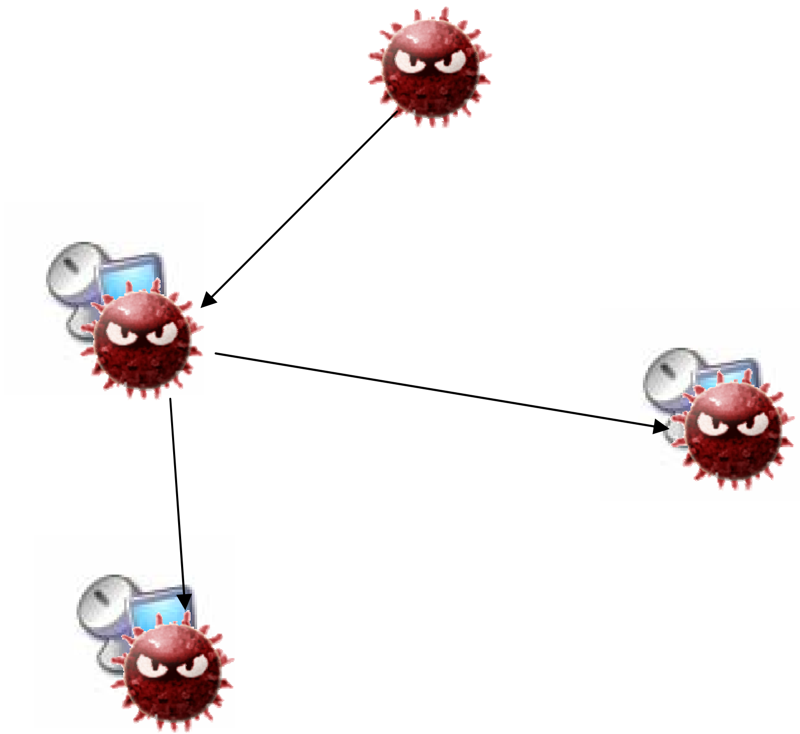
- Trojan Horse
  - Program that masquerades as useful program to execute malicious code once executed by the user.
- Backdoor
  - Malicious program that gives to the attacker the ability to control the compromised system bypassing normal authentication methods.
- Virus
  - Computer program that can self-replicate by making copies of itself or by inserting piece of its code into other “host” programs.
- Worm
  - Computer program that can self-replicate spreading from a computer to another computer using network resources.
- Rootkit
  - Stealth program able to hide its presence in the system by altering core components of the OS.
- Exploit
  - Piece of code that take advantage of a software bug/vulnerability to perform unwanted actions (eg. Privilege escalation, DoS, Code Execution)

Malware Classification: virus or worm?

- Virus



- Worm

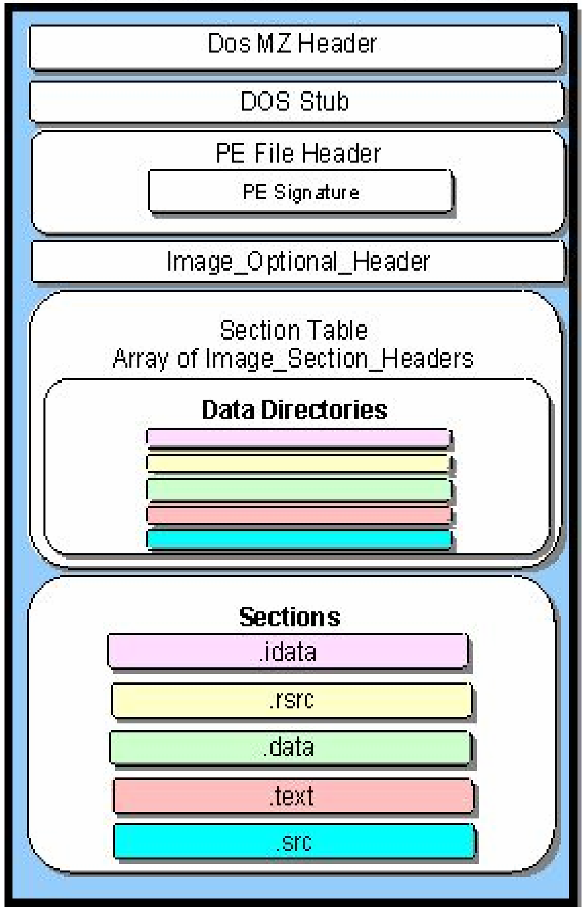


Technical Background

Malware Classification: what's not a "malware" ?

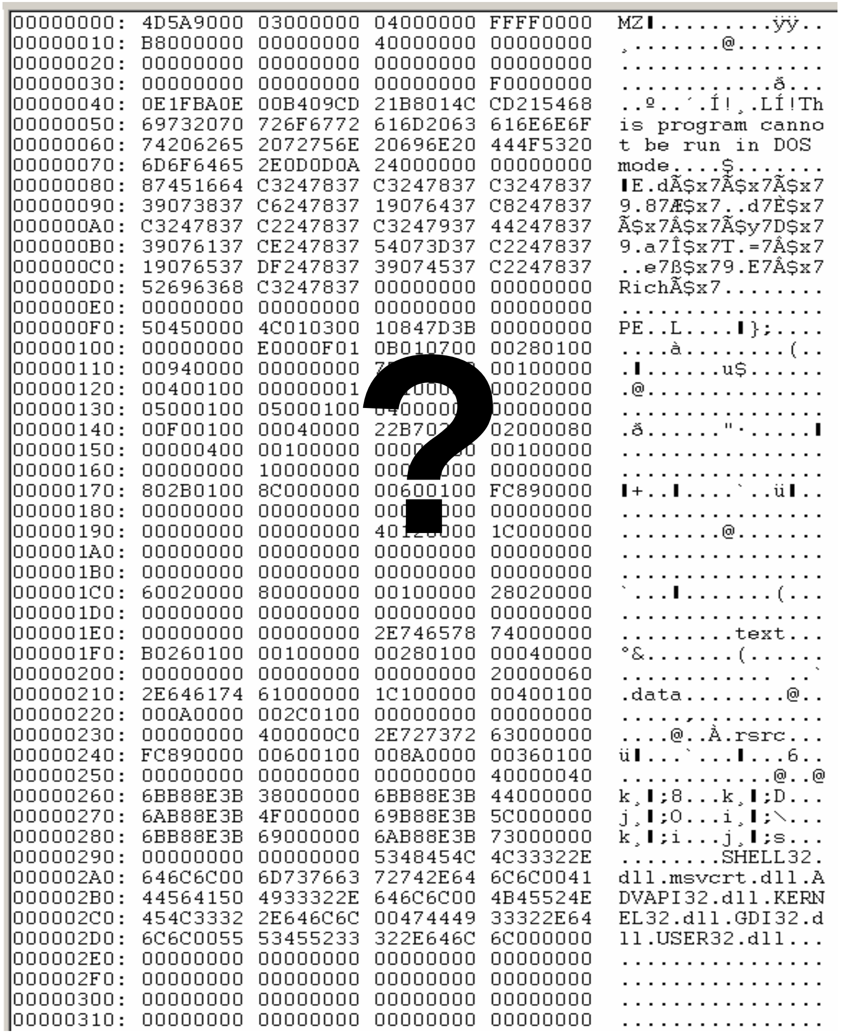
- Adware
  - Software that facilitates delivery of advertising content to the user through their own or another program's interface.
- Spyware
  - Programs that have the ability to gather, collect and distribute personal information, individual files and users data.
- Dialer
  - Programs that use a hijack/modify modem connection to dial out to a toll number or internet site, typically to accrue charges.
- Hacktool
  - Programs that can be used by an attacker for malicious purposes (eg. lower security settings, disable firewall, gain privilege, attack an host, perform a DoS)
- Remote Access
  - Programs that allow remote access to an host from a remote computer.
- Others (SecurityRisk)
  - Programs that do not meet the definition of any of the previous category but are a potential risk if installed.

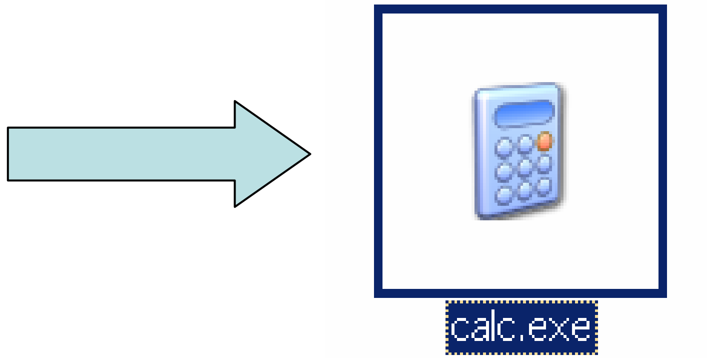
Win32 Portable Executable Format



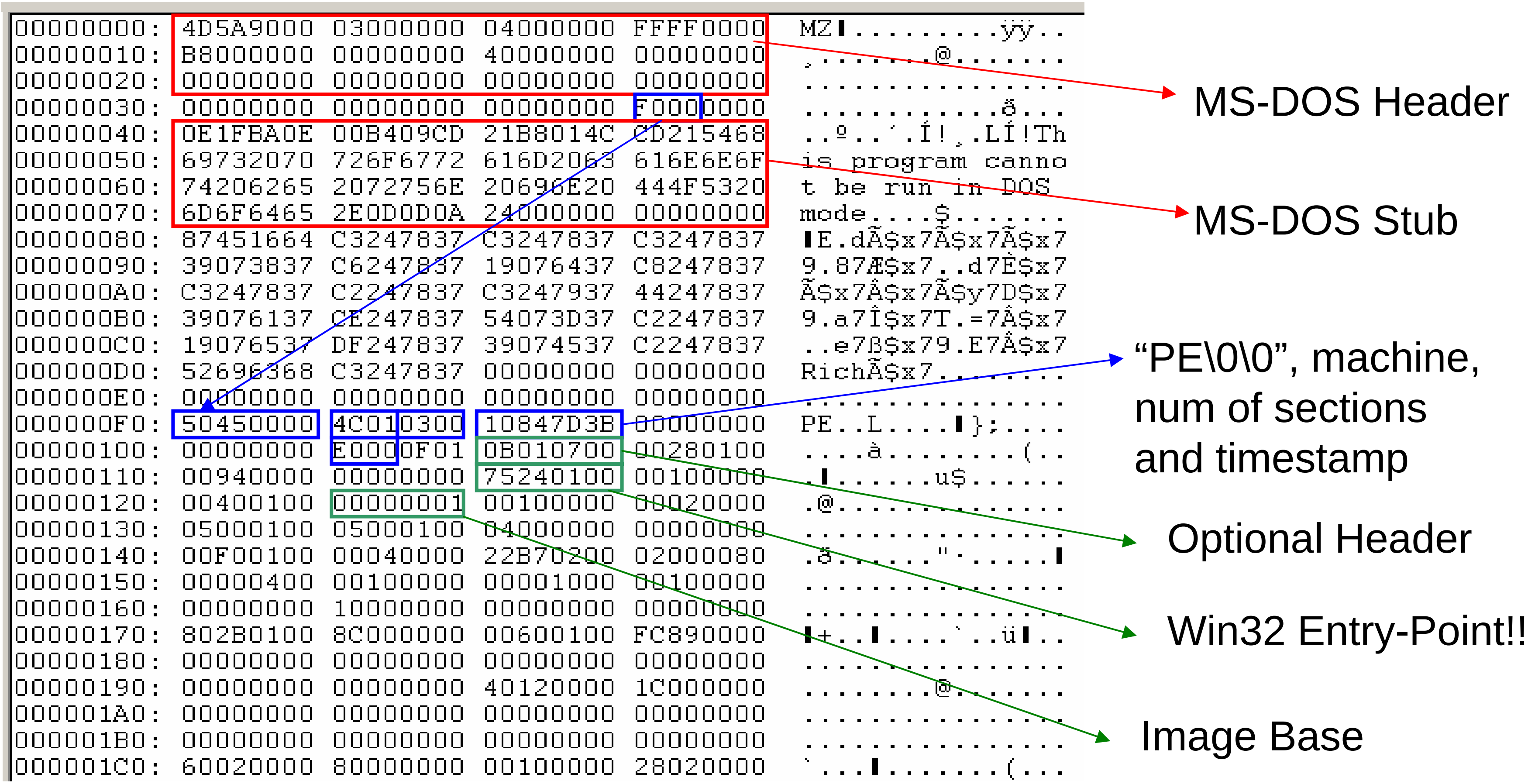
- …every programmer knows exactly how a .CPP file looks like, but who knows what's inside a compiled (executable) file?
- Portable Executable format (PE) was created by Microsoft in 1993 and first introduced in Windows NT 3.1
- PE format was essentially designed by Microsoft from "COFF" format of Unix System V (Common Object File Format Specification)
- PE format - with some improvements - is the official "win32" executable format of almost every Windows (NT, 9X, ME, 2000, XP and 2003)
- PE defines the internal data structure and the encapsulation of code objects inside executables (.EXE / .DLL / .SYS and many other types)
- It's a 32-bit file format created to replace the 16-bit NE of Windows 3.x and the old MS-DOS "MZ" format
- PE was designed to keep backward compatibility with old DOS application and contains a small MS-DOS stub
- PE supports x86 architecture but it's a format in evolution (PE+) and can support IA-64, PowerPC and ARM processors as well (Windows CE executable are still PE files).

Win32 Portable Executable Format



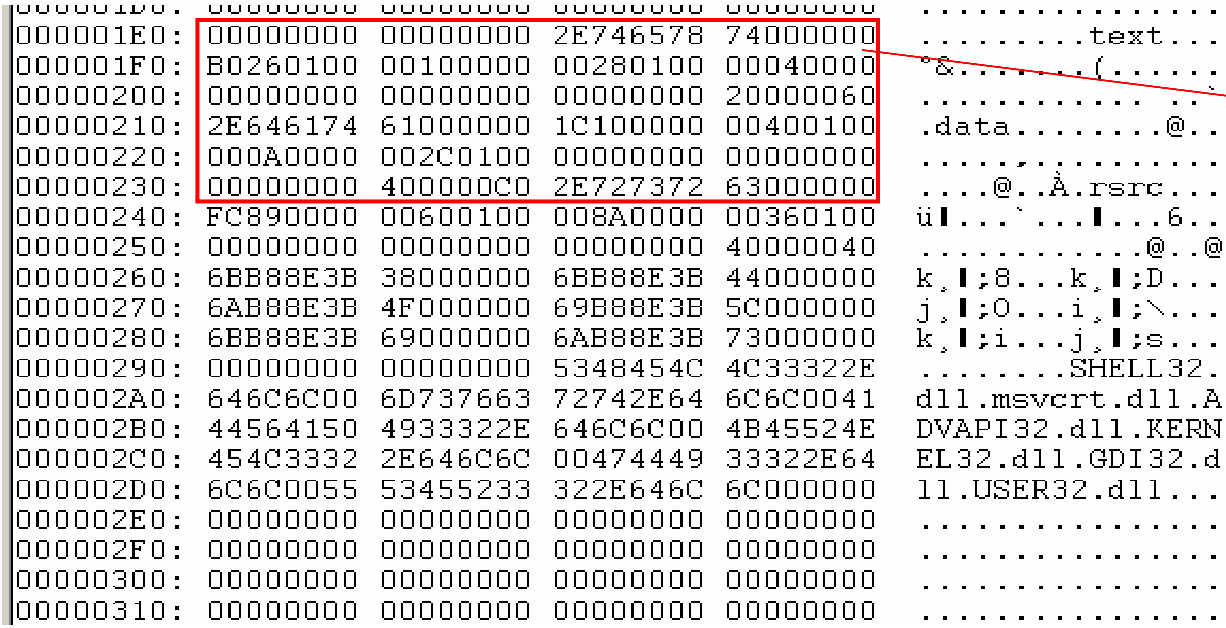


Win32 Portable Executable Format



```
00000000: 4D5A9000 03000000 04000000 FFFF0000  MZ▌.........ÿÿ..
00000010: B8000000 00000000 40000000 00000000  ¸.......@.......
00000020: 00000000 00000000 00000000 00000000  ................
00000030: 00000000 00000000 00000000 F0000000  ............ð...
00000040: 0E1FBA0E 00B409CD 21B8014C CD215468  ..º..´.Í!¸.LÍ!Th
00000050: 69732070 726F6772 616D2063 616E6E6F  is program canno
00000060: 74206265 2072756E 20696E20 444F5320  t be run in DOS 
00000070: 6D6F6465 2E0D0D0A 24000000 00000000  mode....$.......
00000080: 87451664 C3247837 C3247837 C3247837  ▌E.dÃ$x7Ã$x7Ã$x7
00000090: 39073837 C6247837 19076437 C8247837  9.87Æ$x7..d7È$x7
000000A0: C3247837 C2247837 C3247937 44247837  Ã$x7Â$x7Ã$y7D$x7
000000B0: 39076137 CE247837 54073D37 C2247837  9.a7Î$x7T.=7Â$x7
000000C0: 19076537 DF247837 39074537 C2247837  ..e7ß$x79.E7Â$x7
000000D0: 52696368 C3247837 00000000 00000000  RichÃ$x7........
000000E0: 00000000 00000000 00000000 00000000  ................
000000F0: 50450000 4C010300 10847D3B 00000000  PE..L....▌};....
00000100: 00000000 E0000F01 0B010700 00280100  ....à........(..
00000110: 00940000 00000000 75240100 00100000  .▌......u$......
00000120: 00400100 00000001 00100000 00020000  .@..............
00000130: 05000100 05000100 04000000 00000000  ................
00000140: 00F00100 00040000 22B70200 02000080  .ð......"·.....▌
00000150: 00000400 00100000 00001000 00100000  ................
00000160: 00000000 10000000 00000000 00000000  ................
00000170: 802B0100 8C000000 00600100 FC890000  ▌+..▌....`..ü▌..
00000180: 00000000 00000000 00000000 00000000  ................
00000190: 00000000 00000000 40120000 1C000000  ........@.......
000001A0: 00000000 00000000 00000000 00000000  ................
000001B0: 00000000 00000000 00000000 00000000  ................
000001C0: 60020000 80000000 00100000 28020000  `...▌.......(...
```

Win32 Portable Executable Format



```
000001E0: 00000000 00000000 2E746578 74000000   .........text...
000001F0: B0260100 00100000 00280100 00040000   °&.......(......
00000200: 00000000 00000000 00000000 20000060   ............ ..`
00000210: 2E646174 61000000 1C100000 00400100   .data........@..
00000220: 000A0000 002C0100 00000000 00000000   .....,..........
00000230: 00000000 400000C0 2E727372 63000000   ....@..À.rsrc...
00000240: FC890000 00600100 008A0000 00360100   ü....`.......6..
00000250: 00000000 00000000 00000000 40000040   ............@..@
00000260: 6BB88E3B 38000000 6BB88E3B 44000000   k¸.;8...k¸.;D...
00000270: 6AB88E3B 4F000000 69B88E3B 5C000000   j¸.;O...i¸.;\...
00000280: 6BB88E3B 69000000 6AB88E3B 73000000   k¸.;i...j¸.;s...
00000290: 00000000 00000000 5348454C 4C33322E   ........SHELL32.
000002A0: 646C6C00 6D737663 72742E64 6C6C0041   dll.msvcrt.dll.A
000002B0: 44564150 4933322E 646C6C00 4B45524E   DVAPI32.dll.KERN
000002C0: 454C3332 2E646C6C 00474449 33322E64   EL32.dll.GDI32.d
000002D0: 6C6C0055 53455233 322E646C 6C000000   ll.USER32.dll...
000002E0: 00000000 00000000 00000000 00000000   ................
000002F0: 00000000 00000000 00000000 00000000   ................
00000300: 00000000 00000000 00000000 00000000   ................
00000310: 00000000 00000000 00000000 00000000   ................
```

Sections Headers

....at the end of the file

+ Export Table and Import Table (DLLs required and APIs used)

+ Debug Information (.PDB)

+ File Properties (…right click to see!)

Technical Background

Win32 Portable Executable Format

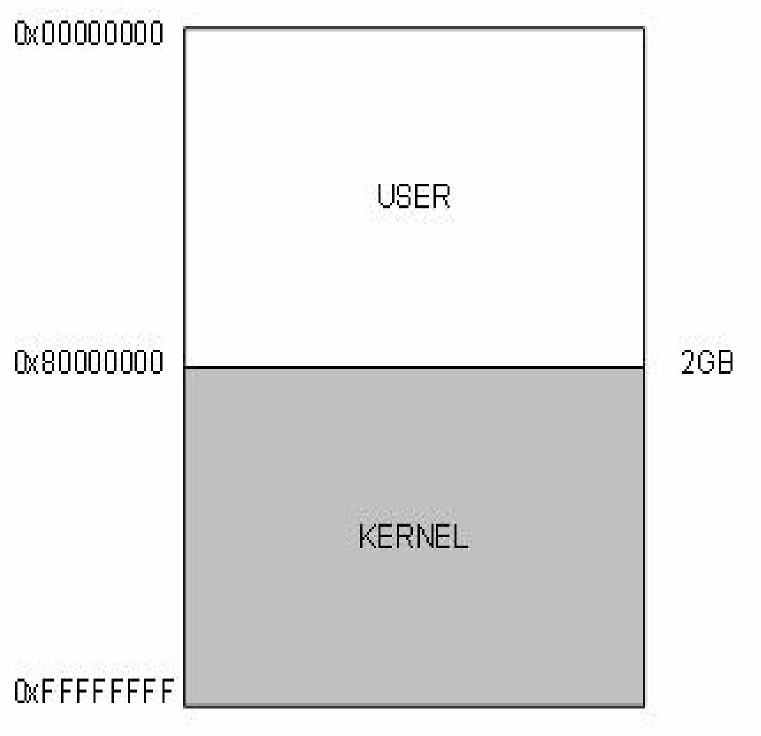
- Executable files on disk look different when loaded in memory. Basic concepts and definitions:

- FILE OFFSET
  - Index or position in the physical image of the file (stored on disk).
- IMAGE BASE
  - Preferred address when loaded into memory (must be a multiple of 64K). The default for EXE in Windows NT, 9X, 2000, XP is 0x00400000. The default for DLLs is 0x10000000.
- RELATIVE VIRTUAL ADDRESS (RVA)
  - RVA is always the address of an item *once loaded into memory* with the <u>base address of the image file subtracted from it</u>. The RVA of an item will almost always differ from its file offset.
- VIRTUAL ADDRESS (VA)
  - Same as RVA, except that the base address of the image file is not subtracted.
- PHYSICAL ADDRESS
  - Real (not virtual) address of data loaded into the physical memory of the machine (\Device\PhysicalMemory).

Technical Background

Win32 Portable Executable Format

- Win32 programs are executed in “Protected Mode”. Windows runs a process into a virtual space and reserves 4 GB memory area for it.
- Processes (in normal conditions) are not allowed to modify code or memory region of other processes.
- x86 CPU supports four different execution “rings”, but Windows uses only two of them (Ring-0 and Ring-3).
- User-Mode programs run usually in Ring-3 and they are not allowed to execute privileged instructions and change memory locations out of their memory space. When an user-mode program crashes, other processes are not affected.
- Kernel Drivers, Services and core system processes run in Ring-0 privileged mode. A software error in a Kernel driver program causes a BSOD (Blue Screen Of Death).



Win32 Portable Executable Format

- Useful tools and resources for PE:
  - Stud PE
  - http://www.softpedia.com/get/Programming/File-Editors/StudPE.shtml
  - Lord PE
  - http://www.softpedia.com/get/Programming/File-Editors/LordPE.shtml
  - ImpRec (Import Reconstructor)
  - ProcDump

- About PE format:
  - *Inside Windows: An In-Depth Look into the Win32 Portable Executable File Format:*
  - http://msdn.microsoft.com/msdnmag/issues/02/02/PE/default.aspx
  - *Microsoft Portable Executable and Common Object File Format Specification:*
  - http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx
  - *"PE File Structure" explained*
  - http://www.madchat.org/vxdevl/papers/winsys/pefile/pefile.htm

Technical Background

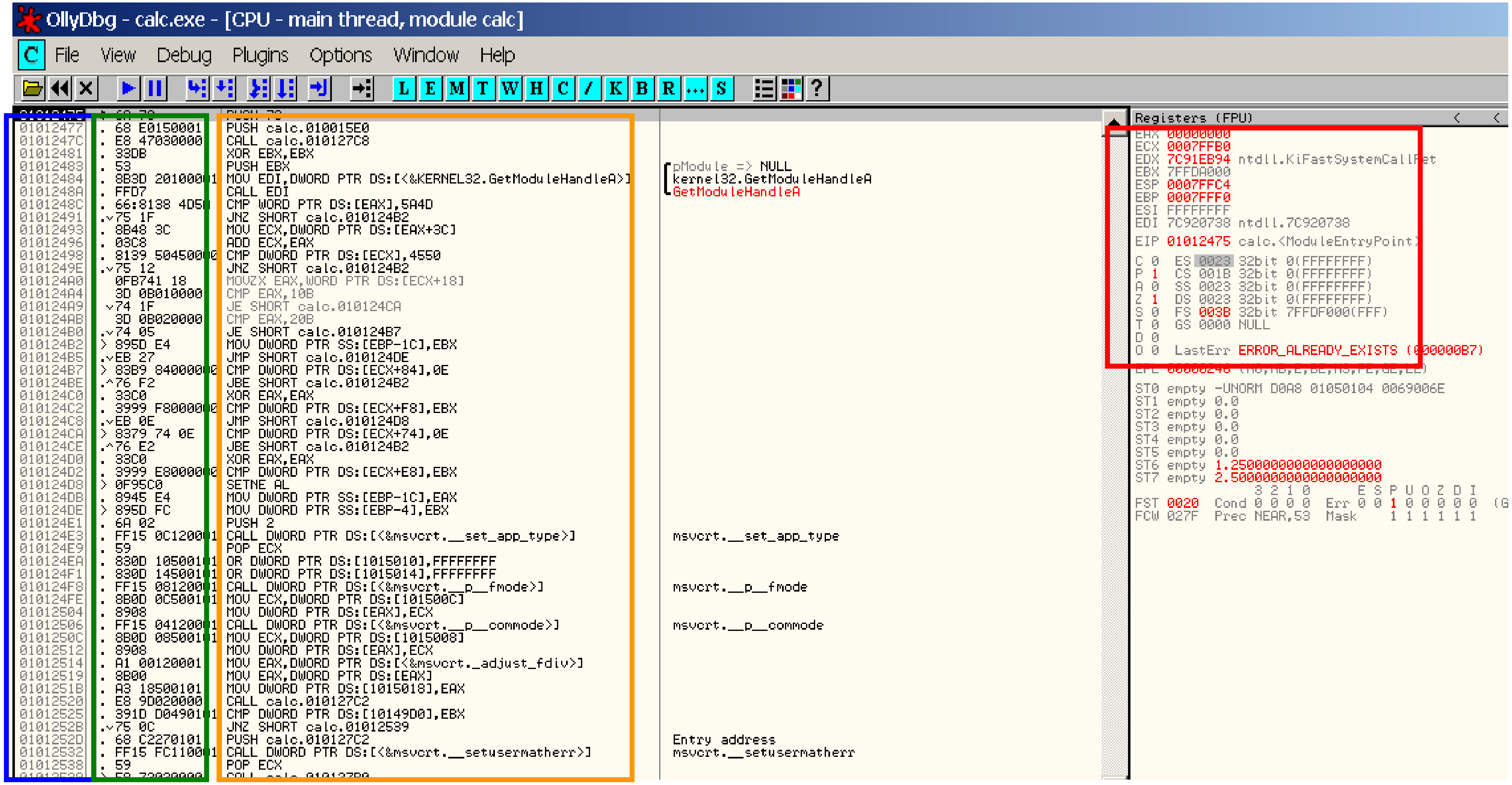
Assembly Language Basics

- Registers of x86 architecture:
  - EAX, EBX, ECX, EDX (accumulator, base, counter, data)
  - ESI, EDI (source, destination)
  - EBP, ESP (stack and base procedure call pointers)
  - EIP (current instruction pointer)
  - DS, ES, SS, FS, GS (segments)
  - CRx (control register, eg. CR3 contains PDB)
  - EFLAGS (ZF, SF, PF, CF, OF)
- Some common instructions (and their opcodes):
  - MOV (0x89, 0x8A,0x8B, …)
  - LEA (0x8D)
  - INC / DEC (0x40, 0x48, …)
  - CMP (0x3A, 0x3C, …)
  - JUMPS: JMP, JZ, JNZ, JE, JNE, JB, JA (0xE9, 0x74, 0x75, …)
  - ADD / SUB
  - AND / OR / XOR (0x83E0, 0x83F0, …)
  - PUSH / POP (0x50, 0x56, 0x53, …)
  - CALL (0xE8, 0xFF15) – RET (0xC3)

Technical Background

Assembly Language Basics

- …not only theory! (Pentium Instruction Set http://www.intel.com/design/pentium4/manuals/245471.htm)



Technical Background

Assembly Language Basics

- Operands and some typical memory addressing:
  - MOV EAX,EBX (register 32-bit)
  - MOV EAX, 0x12345678 (immediate 32-bit)
  - MOV AX, 0x1234 (immediate 16-bit)
  - MOV AL, 0x12 (immediate 8-bit)
  - MOV DWORD PTR [EBX], 0x12345678 (register direct 32-bit)
  - MOV WORD PTR [EBX], 0x1234 (register direct 16-bit)
  - MOV BYTE PTR [EBX], 0x12 (register direct 8-bit)
  - MOV WORD PTR [EBX], EAX (wrong size!)
  - MOV AH, BX (wrong size!)
  - MOV DWORD PTR [EAX], DWORD PTR [EBX] (not allowed!)
  - LEA EAX, DWORD PTR [0x00401000]
  - LEA EAX, DWORD PTR [EAX]
  - LEA EAX, DWORD PTR [EAX*2+EAX] (trick, multiply by 3 fast)
  - XOR EDX,EDX (trick to reset a register fast)
  - PUSH 1234
  - PUSH DWORD PTR [1234]
  - CALL 0x401000 or CALL DWORD PTR [0x401000] or CALL EAX

Technical Background

Windows API and calling convention

- Consider the following C++ program:
  - ```
    int myFunc(int a, int b, int c) {
      int r1,r2,r3;
      r1=a+b;
      r2=c*2;
      r3=r1+r2
      return r3;
    }
    void main() {
      myFunc(10,3,7);
    }
    ```
- How this code will be translated in Assembly language?

```
.text:0040102A                 push    7
.text:0040102C                 push    3
.text:0040102E                 push    0Ah
.text:00401030                 call    sub_401000
.text:00401035                 add     esp, 0Ch
```

- In C++ calling convention the parameters of a function are pushed into the stack from right to left, so the first parameter is always the last to be pushed (the stack works as LIFO).
```

Windows API and calling convention

- The called function saves uses EBP to address the stack and get the parameters from the caller.

```
.text:00401000 sub_401000      proc near                ;
.text:00401000
.text:00401000 var_C           = dword ptr -0Ch
.text:00401000 var_8           = dword ptr -8
.text:00401000 var_4           = dword ptr -4
.text:00401000 arg_0           = dword ptr  8
.text:00401000 arg_4           = dword ptr  0Ch
.text:00401000 arg_8           = dword ptr  10h
.text:00401000
.text:00401000                 push    ebp
.text:00401001                 mov     ebp, esp
.text:00401003                 sub     esp, 0Ch
.text:00401006                 mov     eax, [ebp+arg_0]
.text:00401009                 add     eax, [ebp+arg_4]
.text:0040100C                 mov     [ebp+var_4], eax
.text:0040100F                 mov     ecx, [ebp+arg_8]
.text:00401012                 shl     ecx, 1
.text:00401014                 mov     [ebp+var_8], ecx
.text:00401017                 mov     edx, [ebp+var_4]
.text:0040101A                 add     edx, [ebp+var_8]
.text:0040101D                 mov     [ebp+var_C], edx
.text:00401020                 mov     eax, [ebp+var_C]
.text:00401023                 mov     esp, ebp
.text:00401025                 pop     ebp
.text:00401026                 retn
.text:00401026 sub_401000      endp
```

prolog

epilog

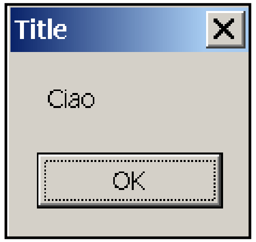
Technical Background

Windows API and calling convention

- *"The Microsoft Windows application programming interface (API) provides building blocks used by applications written for Windows … You can provide your application with a graphical user interface; display graphics and formatted text; and manage system objects such as memory, files, and processes"* – Microsoft MSDN

- API calling example:

```
#include<windows.h>
#pragma comment(lib, "user32")
void main() {
   MessageBox(0, "Ciao", "Title", 0);
}
```



- How this code will be executed by the CPU ?

```
00401003  . 6A 00          PUSH 0                                       Style = MB_OK|MB_APPLMODAL
00401005  . 68 30604000    PUSH test.00406030                           Title = "Title"
0040100A  . 68 38604000    PUSH test.00406038                           Text = "Ciao"
0040100F  . 6A 00          PUSH 0                                       hOwner = NULL
00401011  . FF15 9C504000  CALL DWORD PTR DS:[<&USER32.MessageBoxA>]    MessageBoxA
```

- The CALL lookups a DWORD value (a pointer) from the import table of PE file and redirects the code through the operating system libraries where the real function resides (eg. USER32.DLL, KERNEL32.DLL, GDI32.DLL, etc.).

Agenda

- Introduction
- Technical Background
  - Malware classification
  - Win32 Portable Executable Format
  - Assembly Language Basics
  - Windows API and calling convention
- Reverse Engineering
  - Methodology
  - Disassembler
  - Debugger
  - Network and Monitoring tools
  - Virtual Machines
- Common Problems
  - Executable Packers
  - Encryption
  - Anti-Debugging
  - Stealth Techniques (Rootkit)
  - Polymorphic Code
- Live Malware analysis Demo
- Questions

Reverse Engineering

Methodology

- Reverse Engineering
  - Reverse engineering is the process of creating an high-level description of a software to discern its rules by analyzing its functioning and its internal structure. White box and black box testing and analysis methods both attempt to understand the software, but they use very different approaches.
- White Box
  - White box analysis involves analyzing and understanding source code. Sometimes only binary code is available, but if you decompile a binary to get source code and then study the code, this can be considered a kind of white box analysis as well.
- Black Box
  - Black box analysis refers to analyzing a running program by probing it with various inputs. This kind of testing requires only a running program and does not make use of source code analysis of any kind.
- A mixed approach: Gray Box
  - Gray box analysis combines white box techniques with black box input testing. Gray box approaches usually require using several tools together. A good example of a simple gray box analysis is running a target program within a debugger and then supplying particular sets of inputs to the program.
- DMCA
  - In the United States, the Digital Millennium Copyright Act exempts from the circumvention ban some acts of reverse engineering aimed at interoperability of file formats and protocols (17 USC 1201(f)), but judges in key cases have ignored this law, since it is acceptable to circumvent restrictions for use, but not for access.

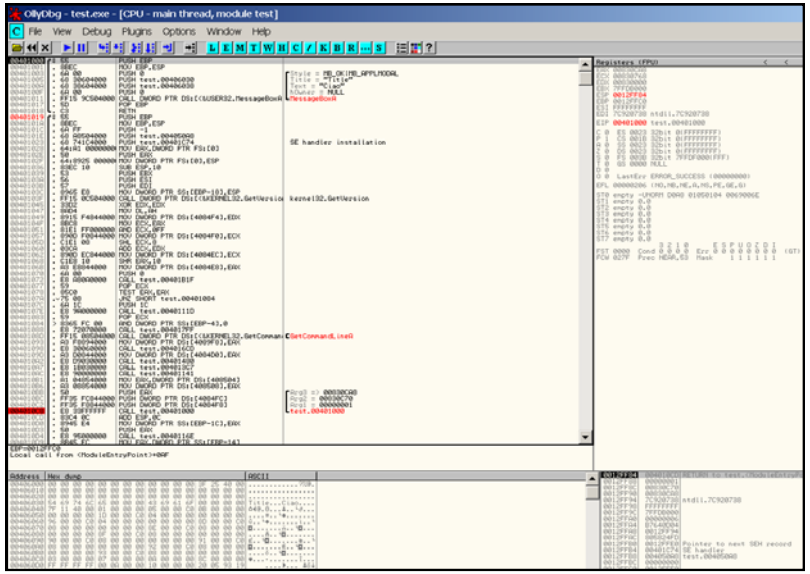
Disassembler

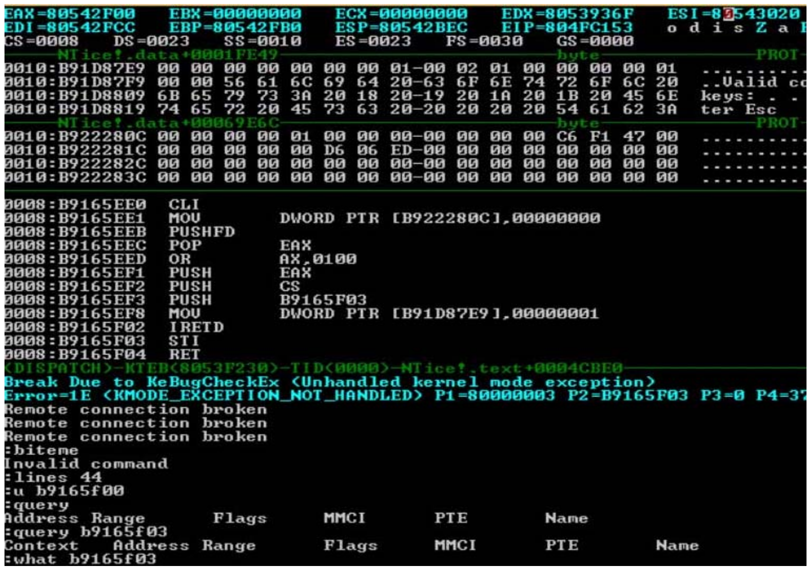
- A disassembler is a computer program which translates machine language into assembly language, performing the inverse operation to that of an assembler. A dissassembler differs from a decompiler, which targets a high level language rather than assembly language (eg. Java).
- IDA (Interactive DisAssembler) is the most famous disassembler (www.datarescue.com)
- 6A 00 68 30 60 40 00 68 38 60 40 00 6A 00..... (opcodes)

```
.text:00401000                         _main           proc near               ; CODE XR
.text:00401000
.text:00401000                         argc            = dword ptr  8
.text:00401000                         argv            = dword ptr  0Ch
.text:00401000                         envp            = dword ptr  10h
.text:00401000
.text:00401000 55                                      push    ebp
.text:00401001 8B EC                                   mov     ebp, esp
.text:00401003 6A 00                                   push    0               ; uType
.text:00401005 68 30 60 40 00                          push    offset Caption  ; "Title"
.text:0040100A 68 38 60 40 00                          push    offset Text     ; "Ciao"
.text:0040100F 6A 00                                   push    0               ; hWnd
.text:00401011 FF 15 9C 50 40 00                       call    ds:MessageBoxA
.text:00401017 5D                                      pop     ebp
.text:00401018 C3                                      retn
.text:00401018                         _main           endp
```

Debugger

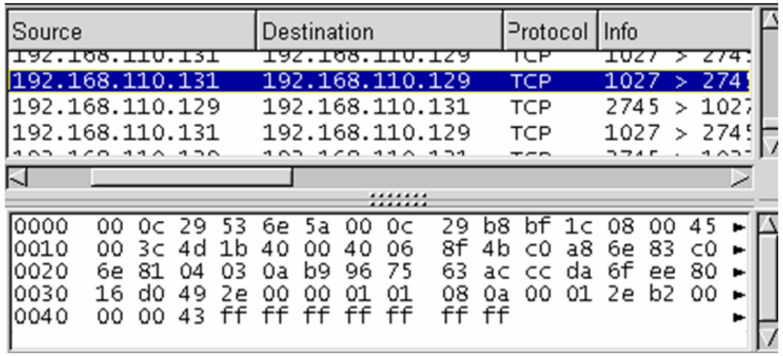
- A debugger is a computer program that is used to analyze, test (and sometimes optimize) other programs. The code to be examined is executed step-by-step and is possible to control the execution when some specific conditions occurs (breakpoint).
- Notable debugging programs are OllyDbg (user-mode debugger, http://www.ollydbg.de) and SoftICE (kernel-mode debugger, http://www.compuware.com).
- Microsoft distributes a free kernel debugger for Windows. WinDbg is downloadable from: http://www.microsoft.com/whdc/devtools/debugging/debugstart.mspx.



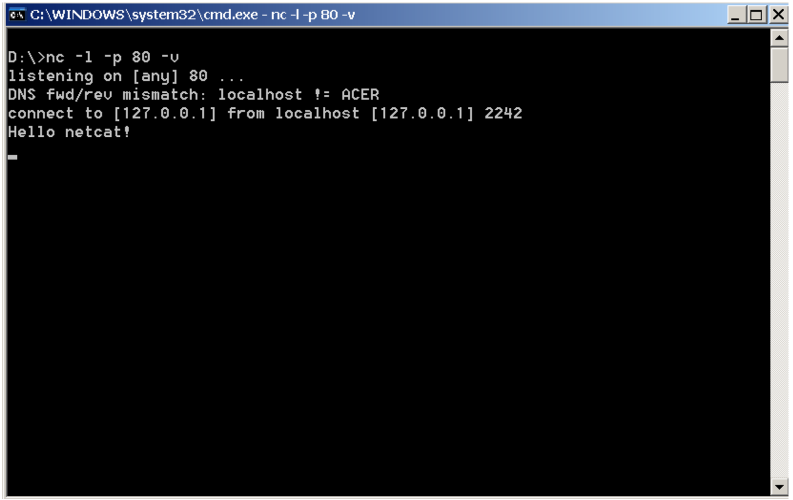


Network and Monitoring Tools

- Sniffer and Protocol Analyzer
  (eg. Ethereal, www.ethereal.com)



- Netcat, the TCP/IP “swiss army knife” available since 1996



- Fake-Server Daemons (httpd, smtpd, ircd)
- IDS (Intrusion Detection System, eg. Snort)
- Vulnerability Scanner
- Monitoring programs for Registry, Files, Disk, API calls (check Mark Russinovich tools at http://www.sysinternals.com)

Virtual Machines

- Virtual Machines are powerful OS emulators that can run as “guest” of a real operating system sharing its resources (memory, disks, network, etc.). For example, a Virtual Machine can run a Linux environment inside a Windows box.
  - VMWare (http://www.vmware.com/)
  - Virtual PC (http://www.microsoft.com/windows/virtualpc/default.mspx)
- VM are used to build Honeypots, simulated environments where is possible to test “live” malware by running them on virtual OS.
- Is the virtual “cage” safe enough? Many security researchers are studying VM environments to find a way to escape from the sand-box, but at the moment there’s still no exploit available.
- However there are several methods and piece of code that can detect if a program is running inside a VM or not. Many recent malwares use this approach to change the behavior during execution.
- Common methods used to detect commercial VM:
  - Hardware / Registry / Process fingerprinting
  - I/O backdoor for VMWare (MOV ECX, 0A / MOV EAX, “VMXh” / MOV DX, “VX” / IN EAX, DX)
  - Invalid Instruction processing for Virtual PC (http://www.codeproject.com/system/VmDetect.asp)
  - “Red Pill” for VMWare (http://invisiblethings.org/papers/redpill.html, SIDT anomaly)

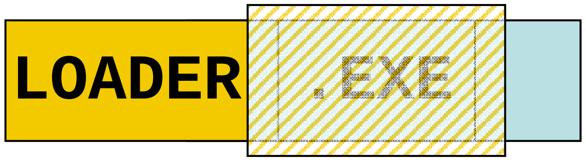
Agenda

- Introduction
- Technical Background
  - Malware classification
  - Win32 Portable Executable Format
  - Assembly Language Basics
  - Windows API and calling convention
- Reverse Engineering
  - Methodology
  - Disassembler
  - Debugger
  - Network and Monitoring tools
  - Virtual Machines
- Common Problems
  - Executable Packers
  - Encryption
  - Anti-Debugging
  - Stealth Techniques (Rootkit)
  - Polymorphic Code
- Live Malware analysis Demo
- Questions

Common Problems

Executable Packers

- Packers are programs that can compress a PE file on disk adding a loader stub to the executable. Once executed, the loader will decompress the original executable in memory and rebuild the PE structure so that the OS will run it without problems.
- Some special packers may also add an encryption layer over the compressed data (making them unreadable to hex editors) and may create a special loader/decrypter stub, which uses anti-debugging to avoid reverse engineering of the code.



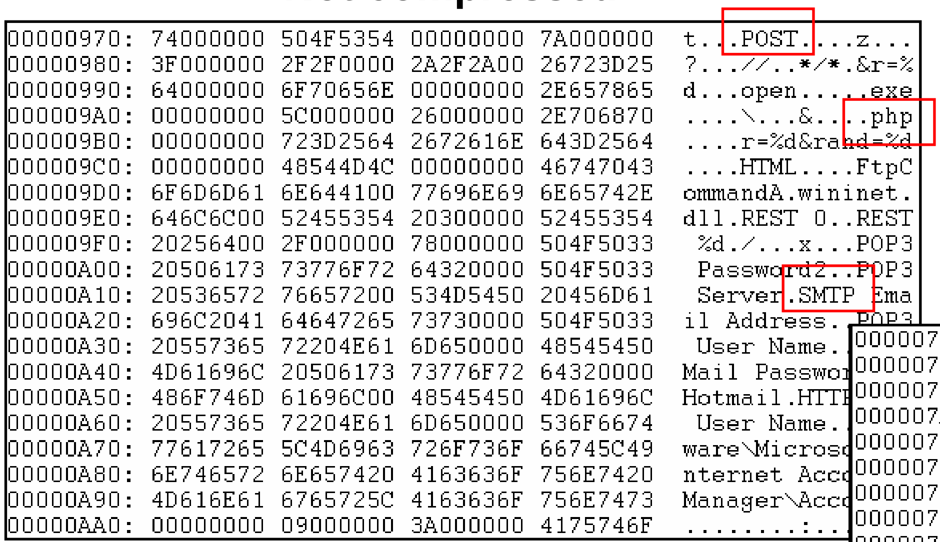
- At the moment there are more than 50 families of packers (…but if we consider custom-made packers, they are much more!)
- Some of the most common packers:
  - UPX, Petite, PolyEne, NsPack, PeCompact, Armadillo, Morphine, ASPack, D.B.P.E., Obsidium
- AV Scan Engines include special code to detect packers or eventually are able to unpack the file and search for virus patterns. Generic unpacking is realized by emulation.

Common Problems

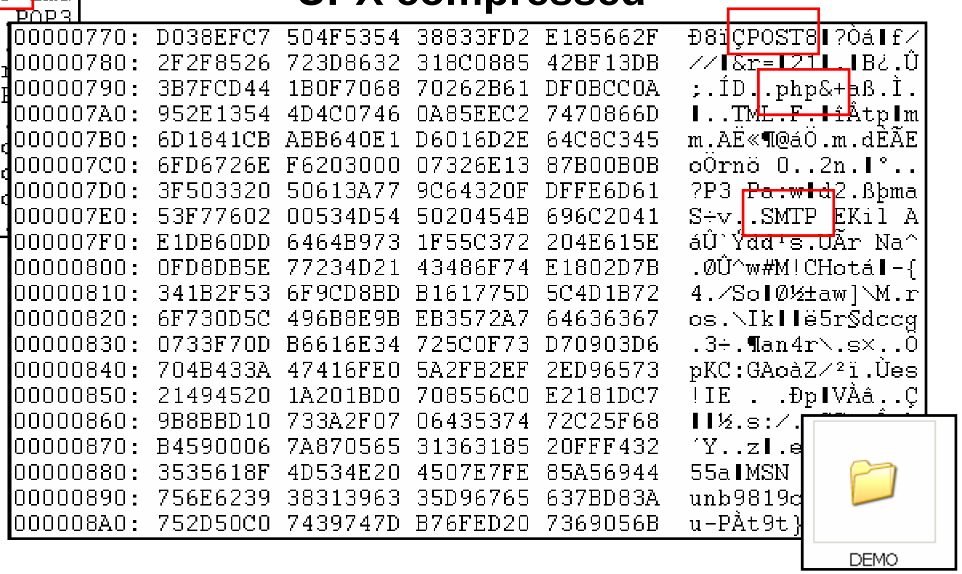
Executable Packers

- UPX compression example:

Not compressed



UPX compressed



Common Problems

Encryption

- Malwares protect their code from static string analysis using encryption algorithms with variable keys.
- Code encryption is used since MS-DOS virus!
- The classic encryption function is XOR (simmetric property):

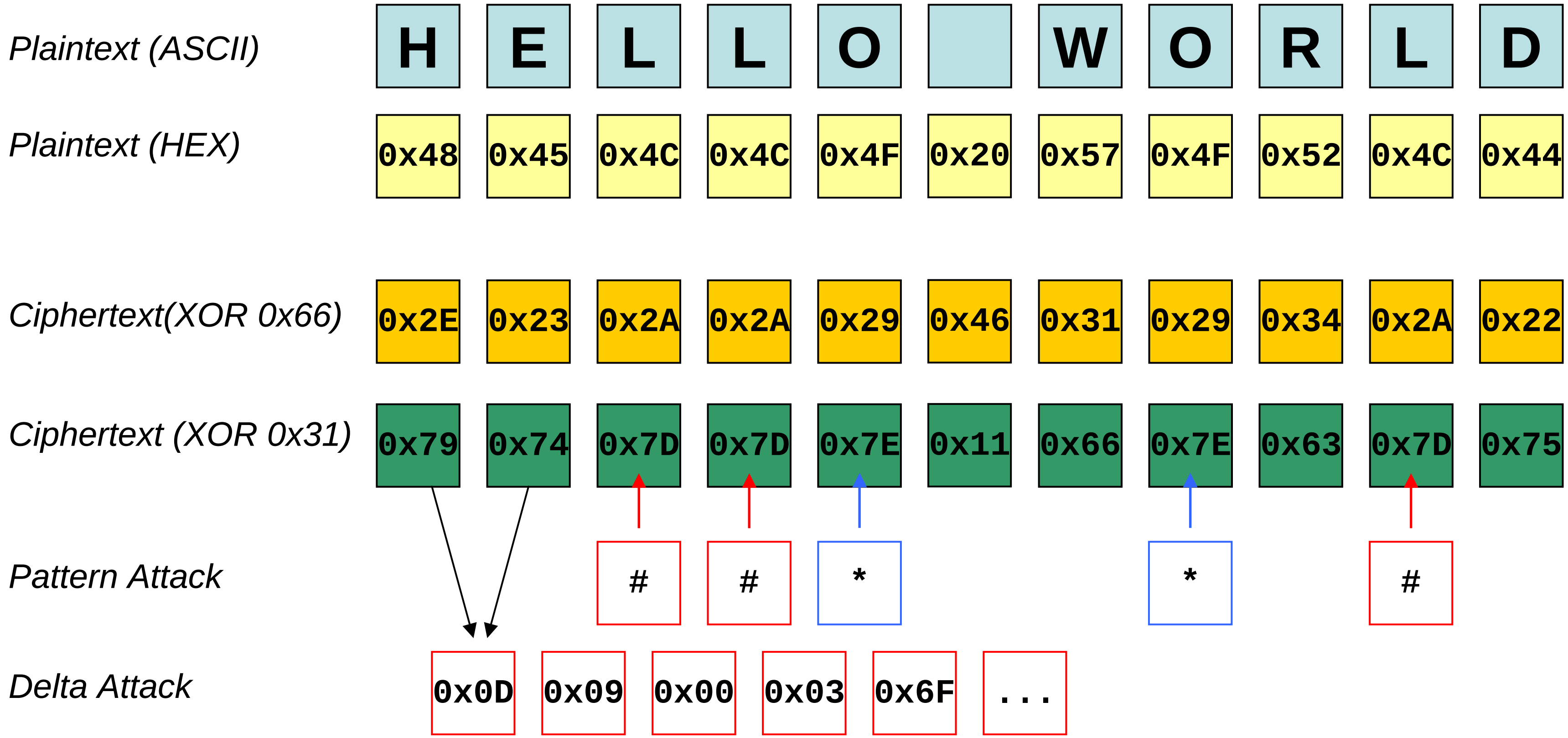
```
  MOV ESI, offset _EncryptedBuffer
  MOV ECX, 0x1000
  MOV DL, 0x5A
encrypting:
  MOV BL, BYTE PTR [ESI]
  XOR BL, DL
  MOV BYTE PTR [ESI], BL
  LOOP encrypting
```

- More complex encryption algorithms use ADD / SUB / ROR / ROL / NOT instructions and they can involve the counter in the key to reduce crypto-analysis attacks success.
- AV scanners can detect the most common encryption loops and are able emulate the generic encryption function to get the original bytes back.
- In some cases is possible to detect a malware by analyzing the encrypted data without decrypting the code. This attack (X-RAY) exploits statistical property of encrypted data and analyze well-known regions of the PE file (known-plaintext attack).

Common Problems

Encryption

- Encryption Example (XOR, 1-byte key, fixed key):



Common Problems

Anti-Debugging

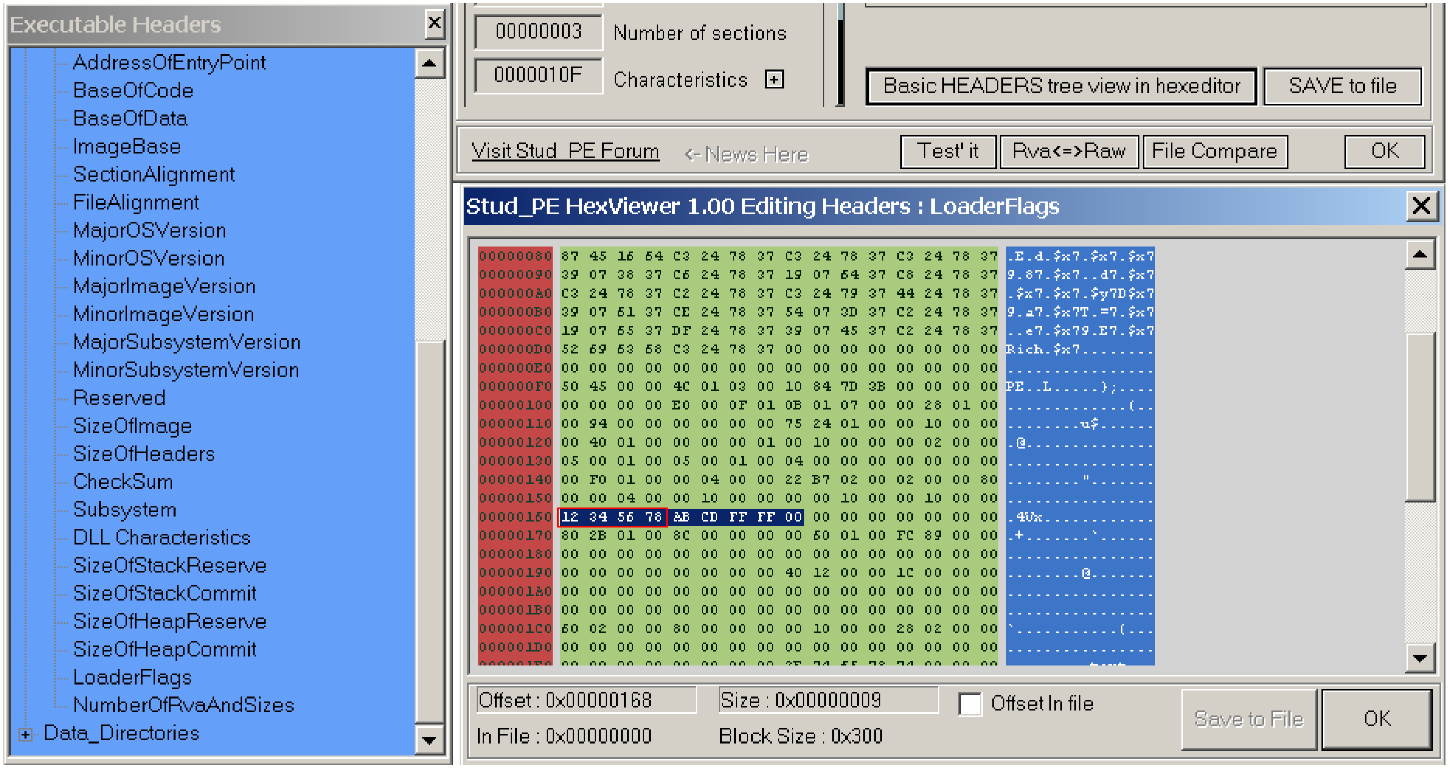
- Anti-Debugging techniques are routine and piece of code used to detect if a program is debugged or not. This techniques can be passive (only detection of the debugger) or active (crashing/attacking the debugger).

- The most common anti-debug techniques are:
  - Malformed PE Header
  - Timing Attacks
  - Check for breakpoint (INT 3)
  - “IsDebuggerPresent” API
  - “CreateFile” API attack (for SoftICE)
  - “FindWindow” API attack (for OllyDbg)
  - SEH (Structured Exception Handler)
  - TLS (Thread Local Storage)
  - INT 1 / INT 3 hooking

Common Problems

Anti-Debugging

- Malformed PE Header example:
  - Patching some fields of the PE header (eg. LoaderFlags and NumberOfRvaAndSizes) with random values is possible to crash OllyDbg when the debugger attempts to run the executable file.



Common Problems

Anti-Debugging

- Timing Attacks
  - When a program is being debugged, it runs in “step-by-step” mode. So, the execution flow is usually slower compared to normal running due to the tracing activity, the presence of breakpoints, the debugger delay, etc.
  - Timing Attacks can detect debuggers by checking the time difference in two different locations of the code.
  - Timing Attacks may use “**GetTickCount**” API or the “**RDTSC**” assembly instruction (ReaD Time Stamp Counter), which get the number of cycles executed by CPU. Comparing the time difference with a specific delta value, the program will take a different execution branch detecting the debugger.

```
#include <windows.h>
void main() {
   _asm {
                call dword ptr [GetTickCount]
                mov ebx, eax

                mov ecx, 0x5000
                fakeLoop:
                dec ecx
                loop fakeLoop

                call dword ptr [GetTickCount]
                sub eax, ebx
                cmp eax, 500             ; 1/2 sec.
                jbe notdebugged

                mov eax,1

                notdebugged:
                mov eax,0

   }
}
```

Common Problems

Anti-Debugging

- Check for breakpoint (INT 3):
  - Debuggers use INT 1 and INT 3 to debug a program step-by-step. INT 3 (opcode = 0xCC) is used to set breakpoints: when the interrupt is triggered, the execution control is returned from the debugged program to the debugger. Checking the code for presence of INT 3 will reveal a debugger in action!

```
#include <windows.h>
void main() {
  _asm {
        mov esi, dword ptr [GetTickCount]
        mov dl, byte ptr [esi]
        cmp dl, 0xCC ; INT3 opcode
        jne notdebugged
        call dword ptr [ExitProcess]

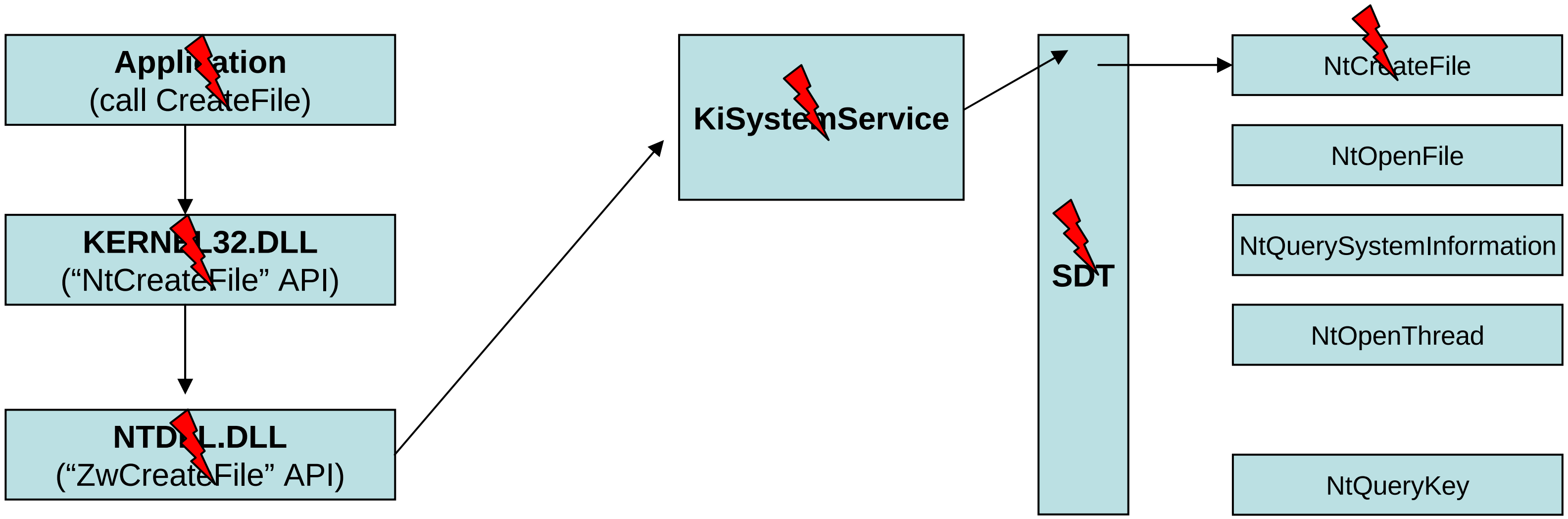
        notdebugged:
        call dword ptr [GetTickCount]
  }
}
```

- "CreateFile" attack for SoftICE:
  -
```
HANDLE hFile=CreateFile( "\\\\.\\NTICE",
                        GENERIC_READ | GENERIC_WRITE,
                        FILE_SHARE_READ | FILE_SHARE_WRITE,
                        NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
NULL);
```

Common Problems

Stealth Techniques (Rootkit)

- Rootkits are stealth programs that hide their presence (files, ports, processes, registry keys) in a compromised system by patching critical area and components of the operating system. Rootkits are able to "subvert" OS by patching API code to return false information or by altering kernel data regions where kernel information are stored.
- Rootkit works in user-mode (Ring-0) or kernel-mode (Ring-3).
- Calling APIs, the big picture:



Common Problems

Stealth Techniques (Rootkit)

- Rootkits techniques:
  - User-mode:
    - IAT patching
    - DLL injection
  - Kernel-mode:
    - IDT (Interrupt Descriptor Table) hooking
    - SDT (System Service Descriptor Table) hooking
    - Native Kernel API hooking
    - DKOM (Direct Kernel Object Manipulation)
- Kernel mode Rootkits need to work in Ring-0 and usually are implemented as System Device Drivers (.SYS files). Alternatively they can patch Kernel by writing directly into "\Device\PhysicalMemory" object.
- Some in-famous rootkits:
  - Vanquish, FU, Hacker Defender, Shadow Walker, Apropos.C, Suckit, eEye BootRoot

Common Problems

Stealth Techniques (Rootkit)

- Resources for Rootkit studying:
  - “Rootkits, subverting Windows Kernel” – Greg Hoglund and Jamie Butler (book)
  - “Windows rootkits of 2005” - http://www.securityfocus.com/infocus/1850
  - Rootkit discussion about code, ideas, new techniques:
  - http://www.rootkit.com
  - J. Rutkowska, developer of SVV and Flister
  - http://www.invisiblethings.org
  - Windows System Call Table (NT/2000/XP/2003) by Metasploit
  - http://www.metasploit.com/users/opcode/syscalls.html
  - Rootkit Revealer by Mark Russinovich
  - http://www.sysinternals.com/Utilities/RootkitRevealer.html

Polymorphic Code

- Polymorphic generators come from old DOS viruses, when many virus writes started to develop complex polymorphic engines (Dark Avenger developed one the first mutation engine called “MtE” in 1992).
- A polymorphic engine is a routine that can generate completely different samples of the same piece of code using different opcodes and without changing the semantic of the original program.

Original (3*4 + 4) program

```
0040103A      B8 03000000     MOV EAX,3
0040103F      C1E0 02         SHL EAX,2
00401042      BB 04000000     MOV EBX,4
00401047      03C3            ADD EAX,EBX
```

Equivalent “junk-instructions” version

```
0040104A      53              PUSH EBX
0040104B      5B              POP EBX
0040104C      B8 03000000     MOV EAX,3
00401051      75 00           JNZ SHORT 00401053
00401053      48              DEC EAX
00401054      40              INC EAX
00401055      C1E0 02         SHL EAX,2
00401058      90              NOP
00401059      90              NOP
0040105A      BB 03000000     MOV EBX,4
0040105F      8BD2            MOV EDX,EDX
00401061      74 00           JE SHORT 00401063
00401063      9B              WAIT
00401064      03C3            ADD EAX,EBX
00401066      F7D3            NOT EBX
00401068      F7D3            NOT EBX
```

Common Problems

Polymorphic Code

- Another example: self-modifying code and meta-morphic code (…powerful of semantic!):

```
0040103A    B8 03000000    MOV EAX,3
0040103F    C1E0 02        SHL EAX,2
00401042    BB 04000000    MOV EBX,4
00401047    03C3           ADD EAX,EBX
```

Self-modifying code

```
01020C90  E8 00000000      CALL 01020C95
01020C95  5E               POP ESI
01020C96  66:C746 07 B803  MOV WORD PTR DS:[ESI+7],03B8
01020C9C  33C0             XOR EAX,EAX
01020C9E  0000             ADD BYTE PTR DS:[EAX],AL
01020CA0  00C1             ADD CL,AL
       2  E0 02            LOOPDNE SHORT 01020CA6
       4  BB 04000000      MOV EBX,4
       9  03C3             ADD EAX,EBX
```

Meta-morphic code

```
0040103A  B8 02000000  MOV EAX,2
0040103F  40           INC EAX
00401040  B9 02000000  MOV ECX,2
00401045  D3E0         SHL EAX,CL
00401047  33DB         XOR EBX,EBX
00401049  83C3 05      ADD EBX,5
0040104C  4B           DEC EBX
0040104D  03C3         ADD EAX,EBX
```