MALWARES: INTRODUCTION TO FEW ANTI-FORENSICS AND UNPACKING TECHNIQUES

BSIDES SAO PAULO 2017

By Alexandre Borges
PROFILE AND TOC

TOC:

• Introduction
• Anti-forensics
• Unpacking
  • Example 1
  • Example 2
  • Example 3
  • Example 4

• Malware and Security Researcher. Consultant, Instructor and Speaker on Malware Analysis, Memory Analysis, Digital Forensics, Rootkits and Software Exploitation.
• Instructor at Oracle, (ISC)2 and EC-Council. Ex-instructor at Symantec.
• Member of the CHFI Advisory Board in EC-Council.
• Reviewer member of the The Journal of Digital Forensics, Security and Law.
• Referee on Digital Investigation: The International Journal of Digital Forensics & Incident Response
• Author of “Oracle Solaris Advanced Administration book”

ALEXANDRE BORGES – MALWARE AND SECURITY RESEARCHER – TWITTER: @ALE_SP_BRAZIL
WARNING !!!

- Please, pay attention in the following considerations:
  - It is NOT ALLOWED to take pictures of the slides.
  - It is NOT ALLOWED to record the lecture.
  - It is NOT ALLOWED to film the lecture.

- Please, respect the speaker and his material.
INTRODUCTION
INTRODUCTION

• This is an INTRODUCTORY lecture. Obviously, we won’t enter into details about kernel malwares (not even close). However, it could help you give few steps forward!

• Please, you should an important thing: tools usually help you, but they do not provide the solution (again, not even close)

• During a case, I’ve heard the following statement from a professional: “It is relatively "straightforward" to detect malware allocating pages in nonpaged memory because they always uses the ExAllocatePoolWithTag( ) function. Therefore, it is possible to trace the allocations through these tags.”
INTRODUCTION

• Yes, I almost have agreed with him. However, it is not true for all situations because of two reasons:

  • first, because the tags may be faked (they do not have to follow what is determined in the pooltags.txt file and Windows does not check an existence of a tag against the file).

  • Second, I’ve already worked on malwares (running as a driver) that demand a big chunk of the allocation on non-pagable memory. Nevertheless, as using ExAllocatePoolWithTag() and ExAllocatePoolWithTagPriority() for this case would cause serious memory fragmentation, so the malware was using besides MmAllocatePagesForMdl() function that prevents excessive memory fragmentation and allows us to allocate big chunks on non-pageable memory. Finally, there is the issue of Kernel Big Page Pools too. Welcome to the real world!
ANTI-FORENSICS
INTRODUCTION

- There are many anti-forensics techniques for trying to hinder the malware analysis, but the most known are from the following areas:
  - Anti-Debugging
  - Anti-Disassembly
  - Miscellaneous
  - Anti-VMs
  - .NET
  - Obfuscation
  - Packers
ANTI-DEBUGGING

- Unfortunately, most professionals use plugins as StrongOD, PhantOM, HideOD and OllyAdvanced for evading anti-debugging techniques, but miss the chance of learning interesting techniques used by malwares, which most time can be modified for bypassing famous debugger as OllyDbg and Immunity.
Many times, when $\text{ZwQueryInformationProcess}$ is used with other functions, the $\text{ProcessInformation}$ is set to -1 when the process is being debugged. This option hides this behavior.
ANTI-DEBUGGING

• Fake breakpoints
  • Some unpackers set up an exception handler for managing problems during the flow execution and use GUARD_PAGES, which is a page protection modifier, to raise an exception. Thus, this exception will be managed by the previously exception handler (and it is the correct path).
  
  • If the malware is being analyzed by a debugger (OllyDbg), so this debugger believes that hit a breakpoint (what is fake because we didn’t set up it) and stop its execution.
  
  • Thus, we should force to raise the exception handler, so an alternative would be use a known trick as changing the next RETN (usually, it will take us to the wrong path) to INT3 (Binary → Edit: replace C3 by CC)
ANTI-DEBUGGING

Please, it is important to know that:

- **Exceptions →** events triggered in the context of the application (Access Violation, Divide by Zero, and so on)

- **Events →** they are NOT errors because they are triggered by OS to notify debuggers about **a new activity such as a process being created (or terminated), a module being loaded (unloaded) and so on.**
ANTI-DEBUGGING

0: kd> sx

ct - Create thread - ignore
et - Exit thread - ignore
cpr - Create process - ignore
epr - Exit process - ignore
ld - Load module - output
ud - Unload module - ignore
ser - System error - ignore
ibp - Initial breakpoint - ignore
iml - Initial module load - ignore
....
ANTI-DEBUGGING

0: kd> sx

.......  

  av - Access violation - break - not handled  
  asrt - Assertion failure - break - not handled  
  aph - Application hang - break - not handled  
  bpe - Break instruction exception - break  
  bpec - Break instruction exception continue - handled  
  .....  
  dm - Data misaligned - break - not handled  
  dbce - Debugger command exception - ignore - handled  
  gp - Guard page violation - break - not handled  
  ii - Illegal instruction - second-chance break - not handled  

Exceptions
ANTI-DEBUGGING

• The PEB (Process Environment Block) contains:

  • command line used for starting the process
  • full path of an executable:
  • three doubly linked list containing the loaded DLLs for the process (useful for investigating hollowing 😊):

    • InMemoryOrderModuleList
    • InLoadOrderModuleList
    • InInitializationOrderList
ANTI-DEBUGGING (FIRST TECHNIQUE)

64 bits:

0: `kd> .expr /s c++`
Current expression evaluator: C++ - C++ source expressions
0: `kd> ? #FIELD_OFFSET(_PEB, NtGlobalFlag)`
Evaluate expression: `188 = 00000000`000000bc

32 bit:

`kd> .expr /s c++`
Current expression evaluator: C++ - C++ source expressions
`kd> ? #FIELD_OFFSET(_PEB, NtGlobalFlag)`
Evaluate expression: `104 = 00000068`
ANTI-DEBUGGING (FIRST TECHNIQUE)

64 bits

```
0: kd> dt _PEB

....
+0x0b0 UnicodeCaseTableData : Ptr64 Void
+0x0b8 NumberOfProcessors : Uint4B
+0x0bc NtGlobalFlag : Uint4B
+0x0c0 CriticalSectionTimeout : _LARGE_INTEGER
+0x0c8 HeapSegmentReserve : Uint8B
....
```
ANTI-DEBUGGING (FIRST TECHNIQUE)

• 32 bits

kd> dt _PEB
+0x060 UnicodeCaseTableData : Ptr32 Void
+0x064 NumberOfProcessors : Uint4B
+0x068 NtGlobalFlag : Uint4B
+0x070 CriticalSectionTimeout : _LARGE_INTEGER
+0x078 HeapSegmentReserve : Uint4B
+0x07c HeapSegmentCommit : Uint4B
ANTI-DEBUGGING (FIRST TECHNIQUE)

• Curiously, **NtGlobalFlag** value is not changed when the process is being debugged!

• Indeed, when a process is being debugged, it is composed by flags containing the following value:
  
  • **FLG_HEAP_ENABLE_TAIL_CHECK** (0x10)
  • **FLG_HEAP_ENABLE_FREE_CHECK** (0x20)
  • **FLG_HEAP_VALIDATE_PARAMETERS** (0x40)

• Therefore, while the process is being debugged, the **NtGlobalFlag** is equal to 70.

• An easy **anti-forensic technique** would be change this value back to zero before resuming the process within the debugger.
ANTI-DEBUGGING (FIRST TECHNIQUE)

0:000> `expr /s c++`
Current expression evaluator: C++ - C++ source expressions
0:000> `? @$peb`
Evaluate expression: 2147315712 = 7ffd7000
0:000> `? #FIELD_OFFSET(_PEB, NtGlobalFlag) + (char *)@$peb`
Evaluate expression: 2147315816 = 7ffd7068
0:000> `? *( (unsigned long *) 0x7ffd7068)`
Evaluate expression: 112 = 00000070 → Being Debugged!

0:000> `expr /s masm`
Current expression evaluator: MASM - Microsoft Assembler expressions
0:000> `? poi(0x7ffd7068)`
Evaluate expression: 112 = 00000070 → Being Debugged!
ANTI-DEBUGGING (FIRST TECHNIQUE)

• 64 bits:

```assembly
xor rax, rax ; rax is zeroed.
mov rbx, qword ptr gs:[0x60] ; remember that PEB is at GS:[0x60]
mov eax, dword ptr [rbx + 0xbc] ; 0xbc is the offset of NtGlobalFlags
and al, 70h ; 70 is the expected value when the debugger is present
cmp al, 70h
je deceive_the_analyst```

Alexandre Borges – Malware and Security Researcher

Twitter: @ALE_SP_BRAZIL
ANTI-DEBUGGING (SECOND TECHNIQUE)

- Another common technique used by malwares as anti-debugging defense is to call the `IsDebuggerPresent` function. It checks the `BeingDebugged` field inside the PEB structure, which reflects whether the malware is being or not debugged.

```
0:000> dt _PEB
    +0x000 InheritedAddressSpace : UChar
    +0x001 ReadImageFileExecOptions : UChar
    +0x002 BeingDebugged    : UChar
    +0x003 BitField         : Uchar

....
```
ANTI-DEBUGGING (SECOND TECHNIQUE)

- **32 bits**
  
  ```
  mov eax, fs:[30h] ;
  cmp b [eax+2], 0 ;
  jne deceive_analyst
  ```

- **64 bits**
  
  ```
  push 60h
  pop rsi
  gs:lodsq ;
  cmp b [rax+2], 0 ;
  jne deceive_analyst
  ```
ANTI-DEBUGGING (SECOND TECHNIQUE)

0:000> `.expr /s c++`
Current expression evaluator: C++ - C++ source expressions

0:000> `? #FIELD_OFFSET(_PEB,BeingDebugged) + (char *)@$peb`
Evaluate expression: 2147340290 = 7ffdd002

0:000> `? *( (unsigned short *) 0x7ffdd002)`
Evaluate expression: 1 = 00000001
ANTI-DEBUGGING (SECOND TECHNIQUE)

```
.text:00401000 sub_401000 proc near ; CODE XREF: sub_4019F0+34p
.text:00401000             push    ebp
.text:00401001             mov     ebp, esp
.text:00401003             call    IsDebuggerPresent
.text:00401008             pop     ebp
.text:00401009             retn
.text:00401009 sub_401000   endp
```

ALEXANDRE BORGES – MALWARE AND SECURITY RESEARCHER – TWITTER: @ALE_SP_BRAZIL
ANTI-DEBUGGING (SECOND TECHNIQUE)
ANTI-DEBUGGING (SECOND TECHNIQUE)
ANTI-DEBUGGING (THIRD TECHNIQUE)

• Other excellent anti-debugging technique used by malware are the ProcessHeap flags. It holds the fields Flags and ForceFlags, which are initialized in conjunction to NtGlobalFlag and their values are usually set to HEAP_GROWABLE (2) on all versions of Windows.

• Flags has the following offset within the heap:
  • Until Windows XP: 0x0C (32-bit) and 0x14 (64-bit)
  • Windows Vista and later: 0x40 (32-bit) and 0x70 (64-bit)

• ForceFlags has the following offset within the heap:
  • Until Windows XP: 0x10 (32-bit) and 0x18 (64-bit)
  • Windows Vista and later: 0x44 (32-bit) and 0x74 (64-bit)
ANTI-DEBUGGING (THIRD TECHNIQUE)

- **Windows 32-bit**: the Flags field during the debugging process:
  - HEAP_GROWABLE (2)
  - HEAP_TAIL_CHECKING_ENABLED (0x20)
  - HEAP_FREE_CHECKING_ENABLED (0x40)
  - HEAP_SKIP_VALIDATION_CHECKS (0x10000000)
  - HEAP_VALIDATE_PARAMETERS_ENABLED (0x40000000)

- **Locating heap on Windows 32-bit**:
  
  ```
  mov eax, fs:[30h] ; Remember that it is the PEB’s offset
  mov eax, [eax+18h] ; Here we get the process heap base
  ```
ANTI-DEBUGGING (THIRD TECHNIQUE)

• **Windows 64-bit:** the Flags field during the debugging process:
  • `HEAP_GROWABLE (2)`
  • `HEAP_TAIL_CHECKING_ENABLED (0x20)`
  • `HEAP_FREE_CHECKING_ENABLED (0x40)`
  • `HEAP_VALIDATE_PARAMETERS_ENABLED (0x40000000)`

• **Locating heap on Windows 64-bit:**
  
  ```
  push 60h ; Remember that it is the PEB’s offset at 64-bit systems
  pop rsi
  gs:lodsq ; getting the PEB
  mov eax, [rax+30h] ; Here we get the process heap base
  ```
ANTI-DEBUGGING (THIRD TECHNIQUE)

• **Windows 32-bit and 64-bit:** the *ForceFlags field* composition during the debugging process:
  
  - HEAP_TAIL_CHECKING_ENABLED (0x20)
  - HEAP_FREE_CHECKING_ENABLED (0x40)
  - HEAP_VALIDATE_PARAMETERS_ENABLED (0x40000000)
ANTI-DEBUGGING (THIRD TECHNIQUE)

• (32 bits) How many heaps has the process? (information usually given by GetProcessHeaps() )

```assembly
push 30h
pop esi
fs:lodsd ; it is getting the PEB
mov esi, [esi+eax+5ch];
lodsd
```
ANTI-DEBUGGING (THIRD TECHNIQUE)

- **(64 bits)** How many heaps has the process? (information usually given by GetProcessHeaps())

push 60h

pop rsi

gs:lodsq ; Here we get the PEB

mov esi, [rsi*2+rax+20h] ;

lodsd
ANTI-DEBUGGING (THIRD TECHNIQUE)

**(32 bits)** Detecting the debugger (using Flag field):

```assembly
mov eax, fs:[30h] ; get the PEB
mov eax, [eax+18h] ; get the process heap base
mov eax, [eax+0ch] ; Offset of heap flags
bswap eax

and al, 0efh ; excluding HEAP_SKIP_VALIDATION_CHECK flag.
cmp eax, 62000040h ; composition of all previously discussed flags
je deceive_analyst
```
ANTI-DEBUGGING (THIRD TECHNIQUE)

- **(64 bits)** Detecting the debugger (using Flag field):

  ```
  push 60h
  pop rsi
  gs:lodsq; it gets the PEB
  mov ebx, [rax+30h]; get process heap base
  cmp d [rbx+70h], 40000062h; composition of all previously discussed flags
  je deceive_analyst
  ```
ANTIDEBUGGING (THIRD TECHNIQUE)

• **(32 bits)** Detecting the debugger (using ForceFlag field):

```
mov eax, fs:[30h] ; it gets the PEB
mov eax, [eax+18h] ; it gets the process heap base
cmp [eax+10h], 40000060h ; composition of all previously discussed flags (ForceFlags)
je deceive_analyst
```
ANTI-DEBUGGING (THIRD TECHNIQUE)

• **(64 bits)** Detecting the debugger (using ForceFlag field):

  ```assembly
  push 60h
  pop rsi
  gs:lodsq ; it gets the PEB
  mov ebx, [rax+30h] ; it gets the process heap base
  cmp d [rbx+74h], 40000060h ; composition of all previously discussed flags (ForceFlags)
  je deceive_analyst
  ```
ANTI-DEBUGGING (THIRD TECHNIQUE)

• How to manually **bypass** the **ProcessHeap flag trick**?
  • C:\> `windbg –hd malware1.exe`

• This simple trick starts the WinDbg, but the **heap is in normal mode** (not debug mode)
ANTI-DEBUGGING (OTHER TRICKS)

• Packers could set their entry point to ZERO.

• Eventually, it could be one more problem because if the analyst to set a breakpoint at the entry point, so part of the magic number (MZ) will be overwritten (the letter M). Therefore, it will cause a boring “INVALID_IMAGE_FORMAT” exception.

• Thus, it is strongly recommended to check the entry point (EP) before starting the analysis by using common tools such as CFF Explorer, PE Bear, Radare, Pestudio and so on.
ANTI-DEBUGGING (OTHER TRICKS)

- Other possible techniques used by malware for detecting debuggers:

  - `OutputDebugString()` ➔ this function is used as indirect trick to send a string to a debugger’s output.

    ```
xor eax, eax
    push offset szBsides
    call OutputDebugString
    
cmp eax, 1 ; If a debugger is attached, so the returned value will be the address of the string. Otherwise, in a common scenario, it should be 1.
    
jne deceive_analyst
    ```
ANTI-DEBUGGING (OTHER TRICKS)

- **NtQueryInformationProcess**: Of course, it gets the information about a process. 😊

```c
NTSTATUS WINAPI NtQueryInformationProcess(
    _In_      HANDLE           ProcessHandle,  // a handle
    _In_      PROCESSINFOCLASS ProcessInformationClass,
    _Out_     PVOID            ProcessInformation,
    _In_      ULONG            ProcessInformationLength,
    _Out_opt_ PULONG           ReturnLength
);
```

- The key is the **second parameter**: If it is **0x7 (ProcessDebugPort)**, so the function tells you whether the process is being debugged (**ring 3**). If it is, **so a port number is returned**. Otherwise, a zero is returned.
ANTI-DEBUGGING (OTHER TRICKS)

```assembly
push 0  // A pointer to a variable in which the function returns the size of the requested information.
push 4  // size of the buffer pointed to by the ProcessInformation parameter
push offset buffer1  // A pointer to a buffer supplied by the calling application into which the function writes the requested information.
push 7  // Type: ProcessDebugPort. Other possible values would be 0, 26, 27, 29 and 75.
push -1
call NtQueryInformationProcess
test eax, eax jne fail
cmp fail, 0 jne deceive_analyst
```

// function returns an NTSTATUS success or error code.
ANTI-DEBUGGING (OTHER TRICKS)

• **TLS (Thread Local Storage)** ➔ Its function is to initialize data that is specific from a thread, so each thread has its own TLS. The “catch” is that the initialization happen before the main thread to start. Nevertheless, TLS callbacks are are also called when the thread is destroyed. Of course, this behavior could be disabled by calling DisableThreadLibraryCalls() or LdrDisableThreadCalloutsForDll() functions.

• **IMAGE_TLS_DIRECTORY**
  
  ```
  struct IMAGE_TLS_DIRECTORY {
    UINT32 StartAddressOfRawData;
    UINT32 EndAddressOfRawData;
    PUINT32 AddressOfIndex;
    PIMAGE_TLS_CALLBACK *AddressOfCallBacks;
    UINT32 SizeOfZeroFill;
    UINT32 Characteristics;
  } IMAGE_TLS_DIRECTORY, *PIMAGE_TLS_DIRECTORY;
  ```
ANTI-DEBUGGING (OTHER TRICKS)
ANTI-DEBUGGING (OTHER TRICKS)

- **TLS section** during a segment view:

  ![TLS section screenshot](image)

- **Control+E** shows all entry points:

  ![Entry points screenshot](image)
ANTI-DEBUGGING (OTHER TRICKS)
ANTI-DEBUGGING (OTHER TRICKS)

• Eventually, the code might be not so good for you. 😊
ANTI-DEBUGGING (OTHER TRICKS)

- We could configure the debugger for catching TLS. In Olly, go to Options → Debugging Option → Events and set System Breakpoint (obviously, WinDbg does not have this problem 😊).
ANTI-DEBUGGING (OTHER TRICKS)
ANTI-DEBUGGING (OTHER TRICKS)

- There is a common trick that is used very often by malwares: taking the difference of time between two execution points. If the time difference is bigger than a determined value, so there is a debugger attached on it.

- The trick catches the single-step process (F7). Moreover, there are different instructions and functions that could be used:
  - rdtsc instruction
  - GetTickCount
  - QueryPerformanceCounter

- For evading these tricks, set a breakpoint after the trick and try to reach this point through a step-over operation (F8). Afterwards, we can use step-in (F7) again for analyzing the code.

- Another option would be disarm the tricks by modifying the code.
ANTI-DEBUGGING (OTHER TRICKS)

- While debugging a malware, most time it is strongly recommended passing all exceptions to the program for preventing anti-debugging tricks, which could use exception handlers.
- Go to Options → Debugging Option → Exceptions
ANTI-DEBUGGING (OTHER TRICKS – INT 2D)
ANTI-DEBUGGING (OTHER TRICKS – INT 2D)

*** Immunity Debugger Python Shell v0.1 ***
ImmLib instantiated as 'imm' PyObject
READY.
>>>imm.setReg("EIP", 0x00413BD7);
0
>>>
HIDING FILES INTO REGISTRY

• An executable is usually:
  • **Splitted** into several parts
  • **Encoded** (based 64)
  • **Stored** in several Registry keys
  • And later, **recovered** for infection

• Important functions for performing this job:
  • **RegCreateKeyExA** (open a handle / create a new key)
  • **RegGetValue** (reads a key value)
  • **RegSetValueEx** (write a key value)
HOLLOWING + HIDING FILES INTO REGISTRY

- Instead of recovering the file from Registry into disk (easily detected by AV), it would be nice to forward it into a memory process by using Hollowing.

  - **Start** a fake process in suspended state
  - **Unmap** its memory
  - **Recovery** the executable file from memory into the unmaped process’s memory
  - **Start** the process execution
CODE INJECTION

• **Process Injection** → it forces a process to **execute an injected code** (VirtualAllocEx/WriteProcessMemory/CreateRemoteThread)

• **DLL Injection** → It is possible to **force a process to load a DLL** into its address space (LoadLibrary)

• **Direct Injection** → It’s possible to inject a code (shellcode) directly from the memory. (WriteProcessMemory / NtMapViewOfSection)

• **APC Injection** → It allows a program to **execute a code in a specific thread** by using a queue of APCs (without using CreateRemoteThread). Some of these APIs: KeInitializeAPC and KeInsertQueueAPC.
CODE INJECTION

• **Hook Injection** → Inject a DLL into a process by using functions such as `SetWindowsHookEx`.

• **Process Replacement** → in few words, we “empty” the content of a process in the memory and insert another content. The basic steps are:
  
  • pause de process (**Create Process with CREATE_SUSPENDED flag**)
  • Unmap the original sections (**ZwUnmapViewOfSection**)
  • Write the new payload (**VirtualAllocEx + WriteProcessMemory**)
  • Resume the paused process. (**ResumeThread**)
CODE INJECTION

• For example, a basic procedure would be:

• Get the NTDLL.DLL handle (GetModuleHandleA( ))
• Get the address of the ZwQueryInformationProcess function (GetProcAddress( ))
• Create a process in SUSPENDED Mode (CreateProcess( ))
• Query information about the process by using ZwQueryInformationProcess( )
• Find the PEB base address and, from there, get the ImageBaseAddress (PEB base address + 8) by using ReadProcessMemory( )
CODE INJECTION

- Read the PE header and check whether there is the MZ string (`ReadProcessMemory( )`)

- Get the **address of the EP (entry point)** from header

- Make the pages where is the **EP writeable** (`VirtualProtectEx( )`)

- **Write the code** (for example, shellcode) at EP (`WriteProcessMemory( )`)

- **Resume** the process (remember, it was in suspended state) by using `ResumeThread( )`. 
# Anti-Disassembly

In [3]: `dis(0x7c900050)`

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7c900050</td>
<td>MOV DL, 0x3</td>
</tr>
<tr>
<td>0x7c900052</td>
<td>JMP 0x7c90005c</td>
</tr>
<tr>
<td>0x7c900054</td>
<td>MOV DL, 0x4</td>
</tr>
<tr>
<td>0x7c900056</td>
<td>JMP 0x7c90005c</td>
</tr>
<tr>
<td>0x7c900058</td>
<td>MOV DL, 0x5</td>
</tr>
<tr>
<td>0x7c90005a</td>
<td>JMP 0x7c90005c</td>
</tr>
<tr>
<td>0x7c90005c</td>
<td>PUSH EDX</td>
</tr>
<tr>
<td>0x7c90005d</td>
<td>CALL 0x7c900066</td>
</tr>
<tr>
<td>0x7c900062</td>
<td>ADD [EAX+EAX+0x6922ff5a], DL</td>
</tr>
<tr>
<td>0x7c90006a</td>
<td>OUTS DX, BYTE [ESI]</td>
</tr>
</tbody>
</table>
ANTI-DISASSEMBLY

In [4]: dis(0x7c900066) ➞ see the previous slide

0x7c900066 5a
0x7c900067 ff22
0x7c900069 696e20444f5320      IMUL EBP, [ESI+0x20], 0x20534f44
0x7c900070 6d                INS DWORD [ES:EDI], DX
0x7c900071 6f                OUTS DX, DWORD [ESI]
ANTI-DISASSEMBLY (OBFUSCATION)

add eax, ecx →

sub eax, C3
add eax + ecx
add eax, C3

push edx
mov edx, 62
inc edx
dec edx
add edx, 61
add eax, edx
pop edx

push ebx
mov ebx, C3
sub eax, ebx
pop ebx

sub eax, A3
add eax, ecx
add eax, A3
push edx

mov edx, 62
inc edx
dec edx
add edx, 61
add eax, edx
pop edx

push ecx
mov ecx, 62
mov edx, ecx
pop ecx

inc edx
dec edx, 61
add eax, edx
pop edx
ANTI-VIRTUAL MACHINE

• Usually, **VMware environment leaves many artifacts** on the systems, so most current malwares look for these artifacts. If they find them, so malwares change your behavior.

  • **VMwareService.exe**
  • **VMwareTray.exe**
  • **VMwareUser.exe**
  • **VMware MAC addresses** *(00:0C:29, for example)*

• **Uninstalling the VMware Tools can be useful.**

• **`net start | findstr VMware`** → this is implemented in the code by malware by using functions such as `CreateToolhelp32Snapshot()` and `Process32Next()`, for example.
ANTI-VIRTUAL MACHINE

```
.text:100033C7  mov    eax, 564D5868h ; 'VMXh'
.text:100033CC  mov    ebx, 0
.text:100033D1  mov    ecx, 0Ah ; 'VX'
.text:100033D6  mov    edx, 5658h
.text:100033DB  in      eax, dx
.text:100033DC  cmp    ebx, 564D5868h ; 'VMXh'
.text:100033E2  setz   [ebp+var_1C]
.text:100033E6  pop     ebx
```

0xA ➔ action: get Vmware version type
UNPACKING MALWARES
UNPACKING MALWARES

• Obviously, most malwares are found as packed on the Internet. Furthermore, they use several methods for hindering the analyst’s job such as **wiping their PE headers**, **using SEH (structured error handling)**, and deploying many anti-debugging techniques (for example, **changing the PE header**).

• Nowadays, many packers are **tailored for evading defenses** and there is not a ready and available unpacker like the UPX.
  
  • During an initial malware analysis, while checking the malware by using **CFF Explorer or PE View**, pay attention on this hint: **Size of Raw Data = 0 and Virtual Size is not zero**. It is one possible indicator that something is packed, but it is not only one.
  
  • There are unpackers such as **UPX, PECompact, FSG, ASPack** and so on, which are a bit easier to unpack it

  • Packers such as **WinUPack, Ntkrnl and Themida** are a bit more complex to analyze.
UNPACKING MALWARES

• While an unpacking process, the great problem is discover the OEP because the packer’s job is to conceal the malicious payload inside an apparently harmless executable.

• During the malware execution, the malware unpacks (decrypt + uncompRESS + extract) the malicious payload.

• Usually (not always), the entire payload is “in clear text” on the memory. Therefore, it is possible to dump this payload after unpacking it. Of course, it is not always easy.

• There are tools such as PEID and DiE that help us in having a better understanding about what packers were used.
UNPACKING MALWARES

- Header
  - Imports
  - Exports
  - .text Section
  - .data Section
  - .rsrc Section

- Unpacking Stub
- Packed Original Code
UNPACKING MALWARES

- Encrypted Payload
- Decoder
- EP
UNPACKING MALWARES

- There are packers that provide different methods for jumping to the OEP. For example, few of them utilize SEH (Structure Exception Handler), which is referenced by FS register [FS:0] (it makes part of the Thread Information Block – TIB):

  kd> dt _TEB

t!_TEB

  +0x000 NtTib : _NT_TIB
  +0x01c EnvironmentPointer : Ptr32 Void
  +0x020 CLIENT_ID : _CLIENT_ID
  +0x028 ActiveRpcHandle : Ptr32 Void
  +0x02c ThreadLocalStoragePointer : Ptr32 Void
UNPACKING MALWARES

kd> dt _NT_TIB

nt!_NT_TIB

+0x000 ExceptionList : Ptr32 _EXCEPTION_REGISTRATION_RECORD
+0x004 StackBase : Ptr32 Void
+0x008 StackLimit : Ptr32 Void
+0x00c SubSystemTib : Ptr32 Void
+0x010 FiberData : Ptr32 Void
+0x010 Version : Uint4B
+0x014 ArbitraryUserPointer : Ptr32 Void
+0x018 Self : Ptr32 _NT_TIB
UNPACKING MALWARES

kd> **dt _EXCEPTION_REGISTRATION_RECORD**

nt!_EXCEPTION_REGISTRATION_RECORD

+0x000 Next : Ptr32 _EXCEPTION_REGISTRATION_RECORD
+0x004 Handler : Ptr32 _EXCEPTION_DISPOSITION

- In **OllyDbg**, it is possible to see the SEH chain in View → SEH chain.
- For passing the exception back to the program, press either **SHIFT+F7 or SHIFT+F8** or **SHIFT+F9**.
UNPACKING MALWARES

![Debugging options window]

- Ignore memory access violations in KERNEL32
- Ignore (pass to program) following exceptions:
  - INT3 breaks
  - Single-step break
  - Memory access violation
  - Integer division by 0
  - Invalid or privileged instruction
  - All FPU exceptions
- Ignore also following custom exceptions or ranges:
  - 00000000 .. FFFFFFFF

Add last exception
Add range
Delete selection

OK  Undo  Cancel
UNPACKING MALWARES

• OllyDbg debugger has **plugins** that could help to **unpack a malware**. Some of these plugins are **OllyScript, QuickUnpack and OllyDump**.

• Eventually, OllyDbg plugins are not able to manage to unpack sample that is **PE header was modified** such as **LoaderFlags and NumberOfRvaAndSizes** field. For **OllyDbg, this last field** is critical and should be set up to **0x10**.

• **OllyAdvanced plugin** is the best plugin **against this anti-unpack tricks** and many other **anti-debug tricks**.

• **Ransomware Chronicle**  ➔ http://privacy-pc.com/articles/ransomware-chronicle.html
Sometimes, malware can be **more complicated than the usual** and it has a very simple IAT containing only ONE dll and export only ONE funcion. In this case, what can we do?

```
C:\Users\AB\Desktop\Malwares\Educational_Malwares>dumpbin /imports malware5.ex_
Microsoft (R) COFF/PE Dumper Version 14.00.23026.0
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file malware5.ex_
File Type: EXECUTABLE IMAGE

Section contains the following imports:

- **KERNEL32.dll**
- 405000 Import Address Table
- 405030 Import Name Table
- 0 time date stamp
- 0 Index of first forwarder reference

22F **IsDebuggerPresent**

Summary

- 4000 .aborges
- 1000 .alexbr
- 1000 .data
- 1000 .rdata

C:\Users\AB\Desktop\Malwares\Educational_Malwares>
```
UNPACKING MALWARES

![Graph showing entropy values over offset (kB)]

- **Entropy (E)**
  - Y-axis: 0 to 1
  - Values fluctuate over the offset range.

- **Offset (kB)**
  - X-axis: 0 to 15
  - Values range from 0 to 1.

**Graph Details**
- The graph likely represents the entropy analysis of a malware sample named `malware5.exe`.
- Entropy values are observed to change significantly over different offsets, indicating dynamic behavior.

**Related Information**
- [HTTP://ALEXANDREBORGES.ORG](http://alexandreborges.org)
- ALEXANDRE BORGES – MALWARE AND SECURITY RESEARCHER – TWITTER: @ALE_SP_BRAZIL
UNPACKING MALWARES (FIRST EXAMPLE)

- Packed malwares try to hide all their API used during the operation by using the `LoadLibrary` and `GetProcAddress` for resolving all necessary APIs. In this specific case, the IAT does not provide any clue about how the malware works.
UNPACKING MALWARES (FIRST EXAMPLE)
UNPACKING MALWARES (FIRST EXAMPLE)
UNPACKING MALWARES (FIRST EXAMPLE)

C:\radare2-w32> rabin2.exe -K entropy -S
C:\Users\AB\Desktop\Malwares\IDA_Education_Samples\malware_3.exe

[Sections]
idx=00 vaddr=0x00401000 paddr=0x00001000 sz=111616 vsz=311296 perm=m-rw-tenentropy=07000000 name=.text
idx=01 vaddr=0x0044d000 paddr=0x0001c400 sz=3072 vsz=4096 perm=m-rw-tenentropy=03000000 name=.rsrcc

2 sections
UNPACKING MALWARES (FIRST EXAMPLE)

What is the best approach for understanding this simple malware? Probably, it would be:

- **Attach** a ring 3 debugger (**Olly** or **Immunity**). If the sample is 64-bit, so **x64dbg** is the best option (by far)
- Find the **LoadLibrary function** (**CTRL+G** and type **LoadLibrary**) then set a breakpoint (**F2**).
- **Run** the malware (**F9**)
- **Remove** the breakpoint (**F2 again**)
- **Return** to the calling subroutine, which is the user code calling the **LoadLibrary** (**ALT + F9**)
UNPACKING MALWARES (FIRST EXAMPLE)

- Eventually, it is necessary continuing “going to up” for finding the code responsible for calling the routing containing the `LoadLibrary`. Therefore:
  - search the a near `RETN` instruction (try to search in the next 30 instructions)
  - set a breakpoint (F2)
  - press F9 (run)
  - remove the breakpoint (F2)
  - press F7 for going to the first instruction from the code calling the procedure containing `LoadLibrary` call.
UNPACKING MALWARES (FIRST EXAMPLE)

• Try to find a **CALL/JMP EAX** instruction or similar. If you find it, verify if before this instruction there is a **POPAD** (used for restoring registers). If there is such instruction, so we are close to OEP.

• **At this point, all imports should be resolved.** (take care: eventually, there will be import redirections, which are able to mask the true API calling)

• As a little hint, try to perform a binary search (**CTRL+B**) for **POPAD** (0x61) instructions. Additionally, we could use **CTRL+S** for searching a sequence of commands (**POPAD, CALL EAX**). In both cases, repeat the operation (**CTRL+L**) until having success. Sometimes, it works. 😊

• Set a breakpoint (**F2**) and run the program (**F9**)
UNPACKING MALWARES (FIRST EXAMPLE)

• Once the debugger hit the breakpoint, remove it (F2) and press F7.

• Check if the finding seems a OEP. Usually, it should has a sequence PUSH EBP and MOV EBP, ESP.

• DUMP the process (Plugins → OllyDump → Dump Debugged Process)

• Dumped files can have their PE header modified and, eventually, they are not runable anymore. It’s possible to fix them, but they are already useful because we can perform static analysis on it. 😊
UNPACKING MALWARES (SECOND EXAMPLE)

- I will show how to detect and understand few details about another simple malware. During the analysis, **there are many steps to perform and many tools that could be used for accomplish the same task**, so I will try to show you some of them here.

- Malwares usually are packed to cause problems for analyzing, so **few checks of the executable are necessary** because we only can really understand the malware after it being unpacked.

- There are **dozens of methods for profiling a malware**, so we are going to see few them here.
UNPACKING MALWARES: (SECOND EXAMPLE)

root@kali:~# rhash2 -a md5,sha1,sha256 /malwares/cerber.ex_
/malwares/cerber.ex_: 0x00000000-0x00045d85 md5: 8f14fc416b687b506db2639a112ad519
/malwares/cerber.ex_: 0x00000000-0x00045d85 sha1: 4661c840df84592cd84d9aff4070f1add41e725f
/malwares/cerber.ex_: 0x00000000-0x00045d85 sha256: e9db14241a3c7be4a8c59431d57a8e62f5d6cb55182d5b9afb44d1de48f2faa3

root@kali:~# rabin2 -K entropy -S /malwares/cerber.ex_

[Sections]
idx=00 vaddr=0x00401000 paddr=0x00000400 sz=26112 vsz=28672 perm=m-r-x entropy=06000000 name=.text
idx=01 vaddr=0x00408000 paddr=0x00006a00 sz=5120 vsz=8192 perm=m-r-- entropy=05000000 name=.rdata
idx=02 vaddr=0x0040a000 paddr=0x00007e00 sz=5632 vsz=143360 perm=m-rw- entropy=03000000 name=.data
idx=03 vaddr=0x0042d000 paddr=0x00000000 sz=0 vsz=73728 perm=m-rw- entropy=00000000 name=.ndata
idx=04 vaddr=0x0043f000 paddr=0x00009400 sz=17920 vsz=20480 perm=m-r-- entropy=05000000 name=.rsrc
UNPACKING MALWARES: (SECOND EXAMPLE)

root@kali:~/# python pecheck.py /malwares/cerber.ex_ | more

PE check for '/malwares/cerber.ex_':

**Entropy: 7.832370** (Min=0.0, Max=8.0)

MD5 hash: 8f14fc416b687b506db2639a112ad519
SHA-1 hash: 4661c840df84592cd84d9aff4070f1add41e725f

......

.text entropy: 6.404728 (Min=0.0, Max=8.0)
.rdata entropy: 5.134399 (Min=0.0, Max=8.0)
.data entropy: 3.549134 (Min=0.0, Max=8.0)
.ndata entropy: 0.000000 (Min=0.0, Max=8.0)
.rscc entropy: 5.916411 (Min=0.0, Max=8.0)
UNPACKING MALWARES: (SECOND EXAMPLE)

...

Overlay:
Start offset: 0x000435e8
Size: 0x0000279e 9.9 KB 3.55%
MD5: c53cf385a814b149590094682c8a71fa
SHA-256: 692dea93f1e473440587873faf182842067d60556b59123896566cc6af2931ce
MAGIC: b1e3a0c4  ❮❯❯ ❮
PE file without overlay:
MD5: 2c40ac0c75d0c1c5769afa2da469b006
SHA-256: 7577b1054c2c9e4aac111fddf94b927ffc9ac2eaf1bc0b187f2ae71ed56c1bf0
UNPACKING MALWARES: (SECOND EXAMPLE)

root@kali:~# rabin2 -I /malwares/cerber.ex_
havecode true
pic false
canary false
nx false
crypto false
va true
bintype pe
class PE32
....
compiled Sun May 11 17:05:33 2014
...
overlay true
signed false
UNPACKING MALWARES: (SECOND EXAMPLE)
UNPACKING MALWARES: (SECOND EXAMPLE)
UNPACKING MALWARES: (SECOND EXAMPLE)
UNPACKING MALWARES: (SECOND EXAMPLE)
UNPACKING MALWARES: (SECOND EXAMPLE)

root@kali:~# peframe cerber.ex_

Short information

<table>
<thead>
<tr>
<th>File type</th>
<th>PE32 executable (GUI) Intel 80386, for MS Windows, Nullsoft Installer self-extracting archive</th>
</tr>
</thead>
<tbody>
<tr>
<td>File name</td>
<td>cerber.ex_</td>
</tr>
<tr>
<td>File size</td>
<td>286086</td>
</tr>
<tr>
<td>Hash MD5</td>
<td>8f14fc416b687b506db2639a112ad519</td>
</tr>
<tr>
<td>Compile time</td>
<td>2014-05-11 17:05:33</td>
</tr>
<tr>
<td>Sections</td>
<td>5 (1 suspicious)</td>
</tr>
<tr>
<td>Directories</td>
<td>import, resource</td>
</tr>
<tr>
<td>Detected</td>
<td>packer, mutex, antidbg</td>
</tr>
<tr>
<td>Import Hash</td>
<td>2b0b308f8b60d9352318c75e4bab2335</td>
</tr>
</tbody>
</table>
UNPACKING MALWARES: (SECOND EXAMPLE)

Paker info

Nullsoft PiMP Stub -> SFX

Resources info

RT_ICON  296  ( wwwwwwwwwwwwwxxfffoxfffoxfffoxfffox
RT_DIALOG  238  *MS Shell DlgP(x P gP( x LPlease w
RT_GROUP_ICON  104  (h 00 h
RT_VERSION  664  4VS_VERSION_INFOStringFileInfo00000
RT_MANIFEST  1002  <?xml version="1.0" encoding="UTF-8
UNPACKING MALWARES: (SECOND EXAMPLE)

Sections suspicious

------------------------------------------------------------
hash_md5               d41d8cd98f00b204e9800998ecf8427e
virtual_address    0x2d000
name                 .ndata
size_raw_data       0
suspicious           True
hash_sha1            da39a3ee5e6b4b0d3255bfef95601890afd80709
virtual_size        0x12000

.... <truncated output>....
UNPACKING MALWARES: (SECOND EXAMPLE)

C:\Documents and Settings\Alexandre\Desktop> java -jar PortexAnalyzer.jar -o CerberBackup\cerber_portanalyzer.txt -p CerberBackup\cerberiamage.jpg cerber.ex_PortEx Analyzer
UNPACKING MALWARES (SECOND EXAMPLE)
UNPACKING MALWARES (SECOND EXAMPLE)
UNPACKING MALWARES (SECOND EXAMPLE)

![Image of executable modules]

<table>
<thead>
<tr>
<th>Base</th>
<th>Size</th>
<th>Entry</th>
<th>Name</th>
<th>File version</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>003B0000</td>
<td>00021000</td>
<td>003B69F4</td>
<td>postulation</td>
<td>2, 7, 5, 5</td>
<td>C: \Users\AB\AppData\Roaming\postulation.dll</td>
</tr>
<tr>
<td>00400000</td>
<td>00044000</td>
<td>0040395D</td>
<td>cerber</td>
<td>1.48</td>
<td>C: \Users\AB\Desktop\cerber.exe</td>
</tr>
<tr>
<td>00450000</td>
<td>00044000</td>
<td>0045365D</td>
<td>cerber_1</td>
<td>1.48</td>
<td>C: \Users\AB\Desktop\cerber.exe</td>
</tr>
<tr>
<td>10000000</td>
<td>00006000</td>
<td>100027C7</td>
<td>System</td>
<td></td>
<td>C: \Users\AB\AppData\Local\Temp\seA525.tmp\System.dll</td>
</tr>
<tr>
<td>713B0000</td>
<td>00055000</td>
<td>713B11D0</td>
<td>shfolder</td>
<td>6.1.7600.16385</td>
<td>C: \Windows\System32\shfolder.dll</td>
</tr>
<tr>
<td>747B0000</td>
<td>00130000</td>
<td>747B10D8</td>
<td>dwmapi</td>
<td>6.1.7600.16385</td>
<td>C: \Windows\System32\dwmapi.dll</td>
</tr>
<tr>
<td>749C0000</td>
<td>00040000</td>
<td>749C22DD</td>
<td>uxttheme</td>
<td>6.1.7600.16385</td>
<td>C: \Windows\System32\uxtheme.dll</td>
</tr>
<tr>
<td>74A10000</td>
<td>00F50000</td>
<td>74A1009E</td>
<td>propsys</td>
<td>7.00.7600.16385</td>
<td>C: \Windows\System32\propsys.dll</td>
</tr>
<tr>
<td>74BD0000</td>
<td>00210000</td>
<td>74BD145E</td>
<td>ntmsra</td>
<td>6.1.7600.16385</td>
<td>C: \Windows\System32\ntmsra.dll</td>
</tr>
<tr>
<td>74C00000</td>
<td>0019E000</td>
<td>74CE6666</td>
<td>comctl32</td>
<td>6.10 (win7_rtl)</td>
<td>C: \Windows\wine\v86\Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.7601.17</td>
</tr>
<tr>
<td>752A0000</td>
<td>00099000</td>
<td>752A1220</td>
<td>version</td>
<td>6.1.7600.16385</td>
<td>C: \Windows\System32\version.dll</td>
</tr>
<tr>
<td>75C30000</td>
<td>004C0000</td>
<td>75C32C14</td>
<td>apphelp</td>
<td>6.1.7600.16385</td>
<td>C: \Windows\System32\apphelp.dll</td>
</tr>
<tr>
<td>75C50000</td>
<td>00008000</td>
<td>75C510E1</td>
<td>cryptbase</td>
<td>6.1.7600.16385</td>
<td>C: \Windows\System32\cryptbase.dll</td>
</tr>
<tr>
<td>75E40000</td>
<td>004A0000</td>
<td>75E47DEB</td>
<td>KernelBase</td>
<td>6.1.7600.16385</td>
<td>C: \Windows\System32\KernelBase.dll</td>
</tr>
<tr>
<td>75E90000</td>
<td>00120000</td>
<td>75E91441</td>
<td>devobj</td>
<td>6.1.7600.16385</td>
<td>C: \Windows\System32\devobj.dll</td>
</tr>
<tr>
<td>76000000</td>
<td>00070000</td>
<td>76005801</td>
<td>config32</td>
<td>6.1.7601.17514</td>
<td>C: \Windows\System32\config32.dll</td>
</tr>
<tr>
<td>76030000</td>
<td>00190000</td>
<td>76034975</td>
<td>sechost</td>
<td>6.1.7600.16385</td>
<td>C: \Windows\System32\sechost.dll</td>
</tr>
<tr>
<td>76050000</td>
<td>0000A000</td>
<td>7605136C</td>
<td>lpk</td>
<td>6.1.7600.16385</td>
<td>C: \Windows\System32\lpk.dll</td>
</tr>
<tr>
<td>76080000</td>
<td>00090000</td>
<td>76083985</td>
<td>user32</td>
<td>6.1.7601.17514</td>
<td>C: \Windows\System32\user32.dll</td>
</tr>
<tr>
<td>761A0000</td>
<td>0000C000</td>
<td>761A168B</td>
<td>msctf</td>
<td>6.1.7600.16385</td>
<td>C: \Windows\System32\msctf.dll</td>
</tr>
<tr>
<td>76270000</td>
<td>000D0000</td>
<td>762A3F07</td>
<td>usp10</td>
<td>1.062.7601.175</td>
<td>C: \Windows\System32\usp10.dll</td>
</tr>
<tr>
<td>763D0000</td>
<td>00C40000</td>
<td>76451601</td>
<td>shell32</td>
<td>6.1.7601.17514</td>
<td>C: \Windows\System32\shell32.dll</td>
</tr>
</tbody>
</table>
UNPACKING MALWARES (SECOND EXAMPLE)
UNPACKING MALWARES (SECOND EXAMPLE)
UNPACKING MALWARES (SECOND EXAMPLE)
UNPACKING MALWARES (SECOND EXAMPLE)

CFF Explorer VIII - _01E80000.mem_2

Module Name      Imports | OFTs    | TimeDateStamp | ForwarderChain | Name RVA  | FTs (DAT)
------------------|---------|---------------|----------------|-----------|---------
CRYPT32.dll      3        | 00113210| 0000000000    | 0000000000     | 0001358E  | 0011050  
SHLWAPL.dll      19       | 00131438| 0000000000    | 0000000000     | 00013652  | 001127C  
WS2_32.dll       10       | 00131430| 0000000000    | 0000000000     | 0001365E  | 0011338  
POWRFULL:.dll    1        | 00131418| 0000000000    | 0000000000     | 00013675  | 001125C  
urlmon.dll       1        | 00132570| 0000000000    | 0000000000     | 00012646  | 0011388  
CLEAN32.dll      1        | 00131430| 0000000000    | 0000000000     | 00014004  | 0011245  
ofe2.dll        7        | 00132350| 0000000000    | 0000000000     | 00014012  | 0011399  
KERNL32.dll      105      | 00132450| 0000000000    | 0000000000     | 0011050C  | 001109C  

OFTs     FTs (DAT) | Hint | Name
---------|------|------
Dword    Dword    Word      
0001358E 0001358E 007C    |      |      
0001433A 0001343A 00D8    |      |      
00001258 00013578 0063    | szAnsi          

CryptBinaryToStringA
CryptStringToBinaryA
CryptDecodeObjectEx
UNPACKING MALWARES (SECOND EXAMPLE)

This is the original and packed sample. Compare it against the previous slide.

This is the original and packed sample. Compare it against the previous slide.
UNPACKING MALWARES (SECOND EXAMPLE)
UNPACKING MALWARES (SECOND EXAMPLE)
UNPACKING MALWARES (SECOND EXAMPLE)
UNPACKING MALWARES (SECOND EXAMPLE)
UNPACKING MALWARES (SECOND EXAMPLE)
UNPACKING MALWARES (SECOND EXAMPLE)

root@kali:/tmp/trash# python /root/volatility26/vol.py --profile=Win7SP1x86 -f /samples/cerber.vmem malfind -p 3524 | grep -A8 Process
Vollatility Foundation Volatility Framework 2.6
Process: cerber.ex  Pid: 3524 Address: 0x24a0000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: CommitCharge: 1, MemCommit: 1, PrivateMemory: 1, Protection: 6

0x024a0000 55 8b ec 83 ec 83 83 e4 f8 83 c4 04 83 ec 10 c7 U.............
0x024a0010 45 f0 06 8f 01 00 c7 45 f4 4a 64 01 00 33 c0 89 E...........J...
0x024a0020 45 f8 89 45 fc 8b 45 f8 8b 55 f9 3b c2 73 2b 8b E..E..E..U..;s+.
0x024a0030 45 fc 8b 55 f4 3b c2 73 0d ff 45 fc 8b 45 fc 8b E..U.;s...E...E..

--
Process: cerber.ex  Pid: 3524 Address: 0x7ff70000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: CommitCharge: 1, MemCommit: 1, PrivateMemory: 1, Protection: 6

0x7ff70000 ff 12 68 83 7c 24 28 02 75 0c 8b 7c 24 2c 8b 4c ...|$(u.|$,.L
0x7ff70010 24 30 33 c0 f3 aa 61 c2 14 00 00 00 00 00 00 00 s03..a........
0x7ff70020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..................
0x7ff70030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..................

--
Process: cerber.ex  Pid: 3524 Address: 0x7ff60000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: CommitCharge: 1, MemCommit: 1, PrivateMemory: 1, Protection: 6

0x7ff60000 83 7c 24 08 11 75 03 c2 10 00 b8 4f 01 00 00 e9 |$.u........
0x7ff60010 89 66 53 f7 00 00 00 00 00 00 00 00 00 00 00 00 s5............
0x7ff60020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..................
0x7ff60030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..................

--
Process: cerber.ex  Pid: 3524 Address: 0x7ff90000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: CommitCharge: 1, MemCommit: 1, PrivateMemory: 1, Protection: 6

0x7ff90000 ff 12 83 7c 24 08 07 75 11 8b 44 24 0c c7 00 00 ...|$..u..D$. ....
0x7ff90010 00 00 00 33 c0 c2 14 00 eb 1d 83 7c 24 08 00 75 ...3......|$..u
0x7ff90020 13 8b 44 24 0c 69 8b 78 10 89 78 14 61 33 c0 c2 D$..x..x.a3..
0x7ff90030 14 00 eb 03 c2 14 00 00 00 00 00 00 00 00 00 00 00 00 .................

HTTP://ALEXANDREBORGES.ORG
UNPACKING MALWARES (SECOND EXAMPLE)
UNPACKING MALWARES (SECOND EXAMPLE)

root@kali:/tmp/trash# python /root/volatility26/vol.py --profile=Win7SP1x86 -f /samples/cerber.vmem malfind -p 3524 | grep -i process
Volatility Foundation Volatility Framework 2.6
Process: cerber.ex Pid: 3524 Address: 0x24a0000
Process: cerber.ex Pid: 3524 Address: 0x7ff70000
Process: cerber.ex Pid: 3524 Address: 0x7ff60000
Process: cerber.ex Pid: 3524 Address: 0x7ff90000
Process: cerber.ex Pid: 3524 Address: 0x7ff80000
Process: cerber.ex Pid: 3524 Address: 0x7ffa0000

root@kali:/tmp/trash# python /root/volatility26/vol.py --profile=Win7SP1x86 -f /samples/cerber.vmem malfind -p 3524 -W | grep -i process
Volatility Foundation Volatility Framework 2.6
Process: cerber.ex Pid: 3524 Address: 0x24a0000

root@kali:/tmp/trash# python /root/volatility26/vol.py --profile=Win7SP1x86 -f /samples/cerber.vmem vadinfo -p 3524 | grep VadS
Volatility Foundation Volatility Framework 2.6
VAD node @ 0x867d5990 Start 0x00280000 End 0x0037ffff Tag VadS
VAD node @ 0x86943d30 Start 0x00200000 End 0x0020ffff Tag VadS
VAD node @ 0x86bb8750 Start 0x00160000 End 0x00160fff Tag VadS
VAD node @ 0x867d1098 Start 0x00030000 End 0x0012ffff Tag VadS
VAD node @ 0x86a7b978 Start 0x00015000 End 0x000150ff Tag VadS
VAD node @ 0x86ab7c60 Start 0x00017000 End 0x000170ff Tag VadS
VAD node @ 0x854a73e8 Start 0x00024000 End 0x00027fff Tag VadS
VAD node @ 0x86ad1c8f Start 0x0003b000 End 0x0003bfff Tag VadS
VAD node @ 0x86aacc90 Start 0x0001b2000 End 0x002320ff Tag VadS
VAD node @ 0x86ac0a28 Start 0x00123000 End 0x00132ffff Tag VadS
VAD node @ 0x869ad308 Start 0x0016e000 End 0x00174ffff Tag VadS
VAD node @ 0x866b75c0 Start 0x0024b000 End 0x002551fff Tag VadS
VAD node @ 0x853c6868 Start 0x00236000 End 0x00249ffff Tag VadS
VAD node @ 0x86add998 Start 0x0024a000 End 0x0024a0fff Tag VadS
VAD node @ 0x868e044a Start 0x07ff70000 End 0x7ff70ffff Tag VadS
VAD node @ 0x867acd10 Start 0x7ff60000 End 0x7ff60fff Tag VadS
VAD node @ 0x8536d370 Start 0x7ff90000 End 0x7ff90fff Tag VadS
VAD node @ 0x867450b0 Start 0x7ff80000 End 0x7ff80fff Tag VadS
VAD node @ 0x869402c0 Start 0x7ffa0000 End 0x7ffa0fff Tag VadS
## UNPACKING MALWARES (SECOND EXAMPLE)

```
root@kali:~/tmp/scratch ~ root@kali:~/tmp/scratch ~ root@kali:~/tmp/scratch ~ root@kali:~/tmp/scratch ~

<table>
<thead>
<tr>
<th>Pid</th>
<th>Process</th>
<th>Start</th>
<th>End</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>3524</td>
<td>cerber.ex</td>
<td>0x75869ff</td>
<td>0x7589dff</td>
<td>/tmp/trash/cerber_ex.7e7a8e30.0x75829000-0x75869ff..dmp</td>
</tr>
<tr>
<td>3524</td>
<td>cerber.ex</td>
<td>0x75869ff</td>
<td>0x7589dff</td>
<td>/tmp/trash/cerber_ex.7e7a8e30.0x75829000-0x75869ff..dmp</td>
</tr>
<tr>
<td>3524</td>
<td>cerber.ex</td>
<td>0x75869ff</td>
<td>0x7589dff</td>
<td>/tmp/trash/cerber_ex.7e7a8e30.0x75829000-0x75869ff..dmp</td>
</tr>
<tr>
<td>3524</td>
<td>cerber.ex</td>
<td>0x75869ff</td>
<td>0x7589dff</td>
<td>/tmp/trash/cerber_ex.7e7a8e30.0x75829000-0x75869ff..dmp</td>
</tr>
<tr>
<td>3524</td>
<td>cerber.ex</td>
<td>0x75869ff</td>
<td>0x7589dff</td>
<td>/tmp/trash/cerber_ex.7e7a8e30.0x75829000-0x75869ff..dmp</td>
</tr>
<tr>
<td>3524</td>
<td>cerber.ex</td>
<td>0x75869ff</td>
<td>0x7589dff</td>
<td>/tmp/trash/cerber_ex.7e7a8e30.0x75829000-0x75869ff..dmp</td>
</tr>
</tbody>
</table>
```

ALEXANDRE BORGES – MALWARE AND SECURITY RESEARCHER – TWITTER: @ALE_SP_BRAZIL
UNPACKING MALWARES (SECOND EXAMPLE)

```
root@kali:/tmp/trash# grep -l Crypt * | xargs grep -l "This program cannot"
cerber.ex_.7e7a8030.0x024b0000-0x02551fff.dmp
cerber.ex_.7e7a8030.0x74350000-0x7438fff.dmp
cerber.ex_.7e7a8030.0x74390000-0x74484fff.dmp
cerber.ex_.7e7a8030.0x758a0000-0x7593fff.dmp
cerber.ex_.7e7a8030.0x75ff0000-0x76046fff.dmp
cerber.ex_.7e7a8030.0x761b0000-0x7630bff.dmp
cerber.ex_.7e7a8030.0x76310000-0x763e3fff.dmp
cerber.ex_.7e7a8030.0x76800000-0x77449fff.dmp
```

```
root@kali:/tmp/trash# grep -l CryptString * | xargs grep -l "This program cannot"
cerber.ex_.7e7a8030.0x024b0000-0x02551fff.dmp
```
UNPACKING MALWARES (SECOND EXAMPLE)
UNPACKING MALWARES (SECOND EXAMPLE)

```
/tmp/cerber.ex_.7e7a8030.0x024b0000-0x02551fff.dmp._edited - Bless

 cerber.ex_.7e7a8030.0x024b0000-0x02551fff.dmp._edited 

00000000 4D 5A 90 00 03 00 00 04 00 00 00 FF FF 00 00 B8
00000011 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000022 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000033 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000044 00 B4 09 CD 21 B8 01 4C CD 21 54 68 69 73 20 70 72
00000055 6F 67 72 61 6D 20 63 61 6E 6E 6F 74 20 6D 20 6F 0E 1F BA 0E
00000066 75 6E 20 69 66 20 44 4F 53 20 0D 0D 0D 0D 0D 0D
00000077 0A 24 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000088 41 F5 BC AA 41 F5 BC AA 82 FA B3 AA 42 F5 BC AA 82
00000099 FA E3 AA 40 F5 BC AA 82 FA E1 AA 43 F5 BC AA 48 8D
000000aa 3F AA 44 F5 BC AA 48 8D 2F AA 62 F5 BC AA 41 F5 BD
000000bb AA 2B F4 BC AA 5A 68 13 AA 02 F5 BC AA 5A 68 21 AA
000000cc 40 F5 BC AA 52 69 63 68 41 F5 BC AA 00 00 00 00 00

Signed 8 bit: 77
Unsigned 8 bit: 77
Signed 16 bit: 19802
Unsigned 16 bit: 19802
Signed 32 bit: 1297780736
Unsigned 32 bit: 1297780736
Float 32 bit: 2.291794E+08
Float 64 bit: 4.37087150980837E+64

Decimal: 077 090 144 000
Octal: 115 132 220 000
Binary: 01001101 01011010 10

ASCII Text: MZ?

Offset: 0x0 / 0xa1fcf
Selection: None

HTTP://ALEXANDREBORGES.ORG

117

ALEXANDRE BORGES – MALWARE AND SECURITY RESEARCHER – TWITTER: @ALE_SP_BRAZIL
UNPACKING MALWARES (SECOND EXAMPLE)

```
root@kali:/tmp# r2 cerber.ex_.7e7a8030.0x024b0000-0x0251fff.dmp_edited
   -- EXPLICIT CONTENT
[0x00402ff] -> aaa

[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[] [*] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn.* and sym.func.* functions (aan)
[0x00402ff] -> 1S

[Sections]
idx=00 vaddr=0x00401000 paddr=0x00000400 sz=64000 vsz=65536 perm=m-r-x name=.text
idx=01 vaddr=0x00411000 paddr=0x0000fe00 sz=13824 vsz=16384 perm=m-r-- name=.rdata
idx=02 vaddr=0x00415000 paddr=0x00013400 sz=132608 vsz=135168 perm=m-rw- name=.data
3 sections
```
UNPACKING MALWARES (SECOND EXAMPLE)

```
[0x00402ff] > il
[Linked libraries]
crypt32.dll
shlwapi.dll
ws2_32.dll
powrprof.dll
urlmon.dll
oleaut32.dll
ole32.dll
kernel32.dll
user32.dll
advapi32.dll
shell32.dll
gdi32.dll
netapi32.dll
ntdll.dll
version.dll
15 libraries
[0x00402ff] > ie
[Entry points]
vaddr=0x00402ff baddr=0x000023fb baddr=0x00400000 laddr=0x00000000 haddr=0x00000118 type=program
1 entry points
```
UNPACKING MALWARES (SECOND EXAMPLE)
UNPACKING MALWARES (SECOND EXAMPLE)
UNPACKING MALWARES (SECOND EXAMPLE)
UNPACKING MALWARES (THIRD EXAMPLE)

- Open the OllyDbg/Immunity/x64dbg and set a breakpoint for all VirtualAlloc( ) or GlobalAlloc( ) functions.
- One the breakpoint is hit, observe the allocated size for checking whether it is a reasonable space for containing an executable or DLL.
- If the allocated space is good enough, so proceed with the ALT-F9 to continue the execution for returning to the procedure that called the VirtualAlloc( ) or GlobalAlloc( ) functions.
- Right click on EAX (return of the function) and choose Follow in Dump. Probably, there will be a huge empty space.
- Continue the execution by pressing F8 (step-over) until something appears at dump area. If an executable appears, so dump it through this area or Modules windows. If nothing useful to appear there, so repeat the steps.
- If the content to delay to appear, try to use a hardware breakpoint (on write).
UNPACKING MALWARES (THIRD EXAMPLE)
UNPACKING MALWARES (THIRD EXAMPLE)
UNPACKING MALWARES (THIRD EXAMPLE)
UNPACKING MALWARES (THIRD EXAMPLE)
UNPACKING MALWARES (THIRD EXAMPLE)
UNPACKING MALWARES (THIRD EXAMPLE)
UNPACKING MALWARES (THIRD EXAMPLE)
UNPACKING MALWARES (THIRD EXAMPLE)
UNPACKING MALWARES (THIRD EXAMPLE)
UNPACKING MALWARES (THIRD EXAMPLE)
UNPACKING MALWARES (THIRD EXAMPLE)
UNPACKING MALWARES (THIRD EXAMPLE)

- Few comments:
  - Does this technique work in every single case? NOT!!!
  - For example, few cases we have that the executable makes an auto-overwrite!
  - In this cases, do:
    - set a hardware breakpoint at beginning of the code and run (F9)
    - set a breakpoint at RET instructions (F2) and run the code (F9)
    - Step in (F7) and check the code for a valid unpacked code.
    - Sometimes, set a breakpoint at CALL EAX instructions
UNPACKING MALWARES (FORTH EXAMPLE – WINDBG)

root@kali:/samples# rabin2 -e malware_windbg.ex_

[Entrypoints]

vaddr=0x00405130 paddr=0x00002530 baddr=0x00400000 laddr=0x00000000 haddr=0x000000f8

Microsoft (R) Windows Debugger Version 10.0.14321.1024 X86
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: C:\Users\AB\Desktop\Malwares\Educational_Malwares\malware_windbg.ex_

************* Symbol Path validation summary *************
Response Time (ms) Location Deferred Symbol search path is: SRV*C:\Symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00400000 0040b000 image00400000
ModLoad: 77420000 7755c000 ntdll.dll
ModLoad: 75a20000 75af4000 C:\Windows\system32\kernel32.dll
ModLoad: 75650000 7569a000 C:\Windows\system32\KERNELBASE.dll
ModLoad: 770c0000 770f5000 C:\Windows\system32\WS2_32.dll
ModLoad: 75d00000 75dac000 C:\Windows\system32\msvcrt.dll
ModLoad: 77230000 772d1000 C:\Windows\system32\RPCRT4.dll
ModLoad: 76020000 76026000 C:\Windows\system32\NSI.dll

(614.2d8): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=0012fb08 edx=774670f4 esi=ffffffffe edi=00000000
eip=774c05a6 esp=0012fb24 ebp=0012fb50 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 ef1=00000246
ntdll!IdrpDoDebuggerBreak+0x2c:
774c05a6 cc int 3
UNPACKING MALWARES (FORTH EXAMPLE – WINDBG)

0:000> bp $iment(0x00405130)
*** ERROR: Module load completed but symbols could not be loaded for image00400000
0:000> g
Breakpoint 0 hit
eax=75a73c33 ebx=7ffd8000 ecx=00000000 edx=00405130 esi=00000000 edi=00000000
eip=00405130 esp=0012ff8c ebp=0012ff94 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
image00400000+0x5130:
00405130 eb06        jmp    image00400000+0x5138 (00405138)
0:000> u eip
image00400000+0x5130:
00405130 eb06        jmp    image00400000+0x5138 (00405138)
00405132 6877150000   push   1577h
00405137 c3           ret
00405138 9c           pushfd
00405139 6d           pushad
0040513a e802000000   call   image00400000+0x5141 (00405141)
0040513f 33c0         xor    eax,eax
00405141 8bc4         mov    eax,esp
UNPACKING MALWARES (FORTH EXAMPLE – WINDBG)
UNPACKING MALWARES (FORTH EXAMPLE – WINDBG)

0:000> g
Breakpoint 1 hit
eax=75a73c33 ebx=7ffd8000 ecx=00000000 edx=00405130 esi=00000000 edi=00000000
eip=00407550 esp=0012ff8c ebp=0012ff94 iopl=0 nv up ei pl zr na pe nc
   cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
image00400000+0x7550:
00407550 50  push   eax
0:000> u eip
image00400000+0x7550:
00407550 50  push   eax
00407551 6877154000  push   offset image00400000+0x1577 (00401577)
00407556 c20400  ret   4
00407559 8bb55b974000  mov   esi,dword ptr image00400000+0x975b (0040975b)[ebp]
0040755f 0bf6  or   esi,esi
00407561 7418  je    image00400000+0x757b (0040757b)
00407563 8b95e6904000  mov   edx,dword ptr image00400000+0x90e6 (004090e6)[ebp]
00407569 03f2  add   esi,edx
UNPACKING MALWARES (FORTH EXAMPLE – WINDBG)

0:00 > P
eax=75a75c33 ebx=7ffd8000 ecx=00000000 edx=00405130 esi=00000000 edi=00000000
eip=00407551 esp=0012ff88 ebp=0012ff94 iopl=0 nv up ei pl zr na pe nc

cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
image04000000+0x7551:
 00407551 c5 71 15 40 00 push offset image04000000+0x1577 (00401577)

0:00 > P
eax=75a75c33 ebx=7ffd8000 ecx=00000000 edx=00405130 esi=00000000 edi=00000000
eip=00407556 esp=0012ff88 ebp=0012ff94 iopl=0 nv up ei pl zr na pe nc

cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
image04000000+0x7556:
 00407556 c2 20 40 ret 4

0:00 > u eip L15
image04000000+0x7556:
 00407556 c2 20 40 ret 4
 00407559 8bb55b974000 mov esi.dword ptr image04000000+0x975b (0040975b)[ebp]
 0040755f 0bf6 or esi.esi
 00407561 7418 je image04000000+0x757b (0040757b)
 00407563 8bb95e690400 mov edx.dword ptr image04000000+0x90e6 (004090e6)[ebp]
 00407569 03f2 add esi.esi
 0040756b e80f000000 call image04000000+0x757f (0040757f)
 00407570 720b jb image04000000+0x757d (0040757d)
 00407572 83c614 add esi.14h
 00407575 83e0c00 cmp dword ptr [esi+0Ch].0
 00407579 75f0 jne image04000000+0x756b (0040756b)
 0040757b f8 clc
 0040757c c3 ret
 0040757d f9 stc
 0040757e c3 ret
 0040757f c7 8531974000000000 mov dword ptr image04000000+0x9731 (00409731)[ebp],0
 00407589 8bb8 mov ecx.dword ptr [esi]
 0040758b 8bb7e10 mov edi.dword ptr [esi+10h]
 0040758e 8bc9 or ecx.esx
 00407590 7502 jne image04000000+0x7594 (00407594)
 00407592 8bb9 mov ecx.esx

0:00 > P
eax=75a75c33 ebx=7ffd8000 ecx=00000000 edx=00405130 esi=00000000 edi=00000000
eip=00401577 esp=0012ff8c ebp=0012ff94 iopl=0 nv up ei pl zr na pe nc

cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
image04000000+0x1577:
 00401577 55 push ebp

HTTP://ALEXANDREBORGES.ORG
UNPACKING MALWARES (FORTH EXAMPLE – WINDBG)

```
0:000> u eip l15
image=0b400000+0x1577:
00401577 55                push    ebp
00401578 8bec              mov      ebp,esp
0040157a 6aff              push    0FFFFFFFFh
0040157c 68c0404000         push    offset image00400000+0x40c0 (004040c0)
00401581 683c204000         push    offset image00400000+0x203c (0040203c)
00401586 64a100000000       mov      eax,dword ptr fs:[00000000h]
0040158c 50                push    eax
0040158d 648925000000000    mov      dword ptr fs:[0],esp
00401594 83ec10            sub      esp,10h
00401597 53                push    ebx
00401598 56                push    esi
00401599 57                push    edi
0040159a 8965e8            mov      dword ptr [ebp-18h],esp
0040159d ff1530404000       call     dword ptr [image00400000+0x4030 (00404030)]
004015a3 33d2              xor      edx,edx
004015a5 8ad4              mov      dl,ah
004015a7 8915d4524000       mov      dword ptr [image00400000+0x52d4 (004052d4)],edx
004015ad 8bc8              mov      ecx,eax
004015af 81e1ff000000       and      ecx,0FFh
004015b5 890dd0524000       mov      dword ptr [image00400000+0x52d0 (004052d0)],ecx
004015bb c1e108            shl      ecx,8
```
UNPACKING MALWARES (FORTH EXAMPLE – WINDBG)
UNPACKING MALWARES (FORTH EXAMPLE – WINDBG)
UNPACKING MALWARES (FORTH EXAMPLE – WINDBG)
UNPACKING MALWARES (FORTH EXAMPLE – WINDBG)
UNPACKING MALWARES (FORTH EXAMPLE – WINDBG)
UNPACKING MALWARES (FORTH EXAMPLE – WINDBG)

- IAT before unpacking
UNPACKING MALWARES (FORTH EXAMPLE – WINDBG)

- **Strings before unpacking**

  C:\Users\AB\Desktop\Malwares\Educational_Malwares> `strings -a malware5.ex_`

  .aborges
  .rdata
  @.data
  .alexb
  VW+
  ......
  IsDebuggerPresent
  KERNEL32.dll
  malware5.ex_
  ....

http://alexandreborges.org
UNPACKING MALWARES (FORTH EXAMPLE – WINDBG)

- IAT after unpacking
UNPACKING MALWARES (FORTH EXAMPLE – WINDBG)

Strings after unpacking
REMEMBER

We are always in CONTROL... 😊
THANK YOU FOR ATTENDING MY LECTURE!

LinkedIn: http://www.linkedin.com/in/aleborges

Twitter: @ale_sp_brazil

Blog: http://alexandreborges.org

E-mail: alexandreborges@blackstormsecurity.com

• Malware and Security Researcher. Consultant, Instructor and Speaker on Malware Analysis, Memory Analysis, Digital Forensics, Rootkits and Software Exploitation.
• Instructor at Oracle, (ISC)2 and EC-Council. Ex-instructor at Symantec.
• Member of the CHFI Advisory Board in EC-Council.
• Reviewer member of the The Journal of Digital Forensics, Security and Law
• Referee on Digital Investigation: The International Journal of Digital Forensics & Incident Response
• Author of “Oracle Solaris Advanced Administration book”