Week 4 and 5: Packer, Anti-Debug, and Anti-Malware

> Author: Nguyen Anh Khoa
>
> Instructor: Dang Dinh Phuong

In these two weeks the task given are unpacking simple packers, and researching on some common anti debugging and anti virtualization.

> I also went on to find two CTF challenges that uses these techniques. But the report is unable to finish with this secion ontime. There will be modifications, additions, if possible.

Unpacking simple packer

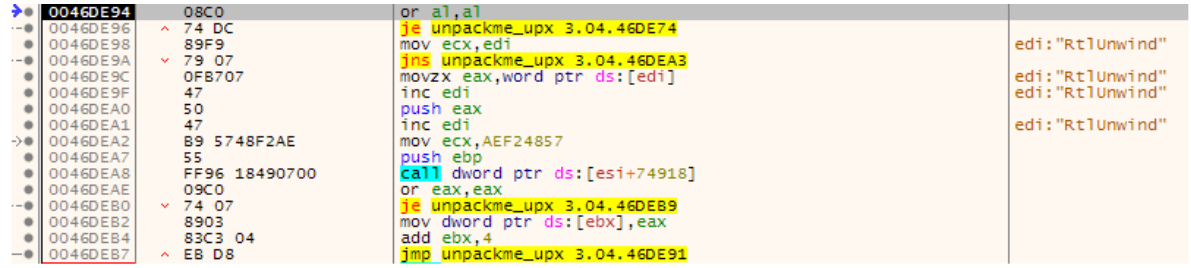
The packer samples are the three common UPX, PeCompact, and ASPack. The idea is simple, the packer will compress the content of the binary, usually `.text` and `.data`, and put a decompress function in the program. When being run, it will unpack the file dynamically, rewrite the packed part, or write out somewhere on memory, or write to a different file. The three packer given here will write the binary in the process memory.

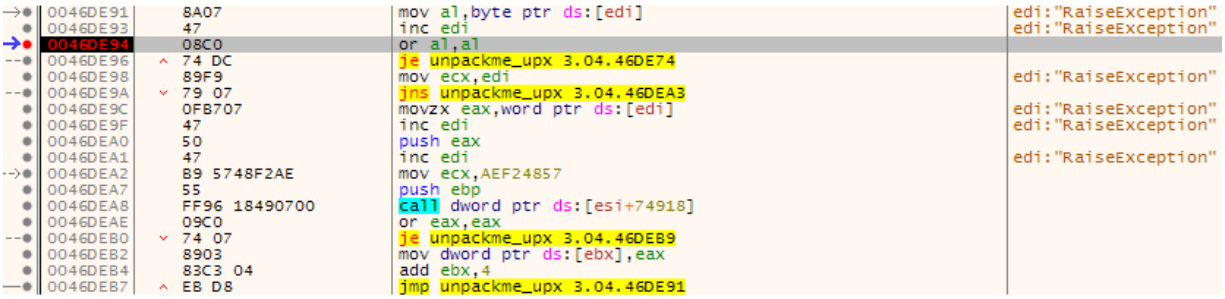
UPX

Disecting the section of a packed UPX executable, we can see that it does not have common section like `.text` nor `.data` but instead, `UPX0`, and `UPX1`. We can see that the section `UPX0` has a virtual size of 0x00044000 but the raw size is 0x00000000. This probaly the section where the code is extracted to. UPX is a simple packer that does not prevent debugging or anything, so unpacking should be easy.

Name	Virtual Size	Virtual Address	Raw Size	Raw Address
Byte[8]	Dword	Dword	Dword	Dword
UPX0	00044000	00001000	00000000	00000200
UPX1	00029000	00045000	00029000	00000200
.rsrc	00008000	0006E000	00007C00	00029200

On running the binary, we will detect code looping many times, in this one specific loop we can see that the arguments loaded to call are changed each time and the names are library names.

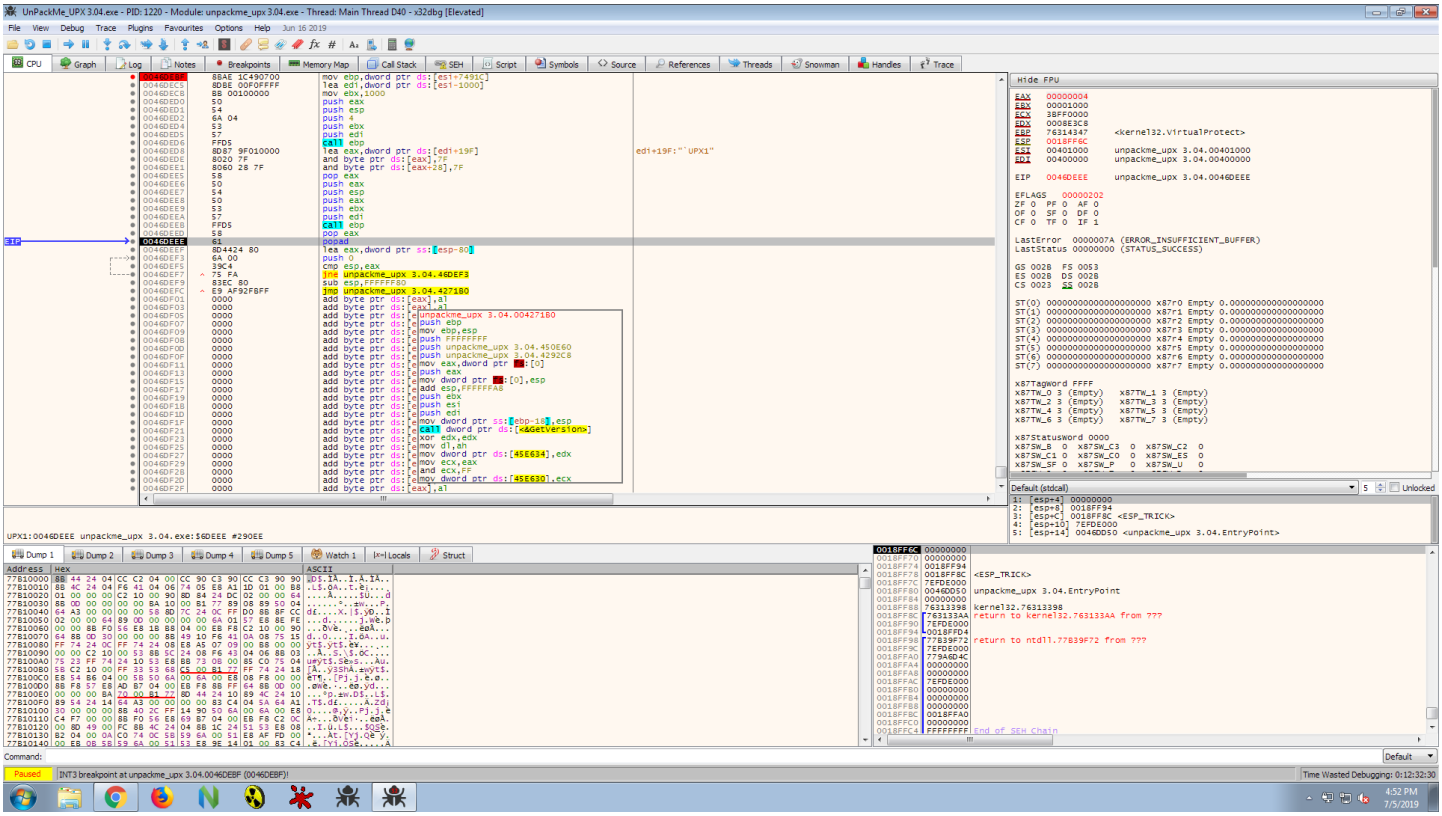




If we keep stepping, soon we will jump to where our real code is. At this time, we can extract the binary from memory which to be discussed later.

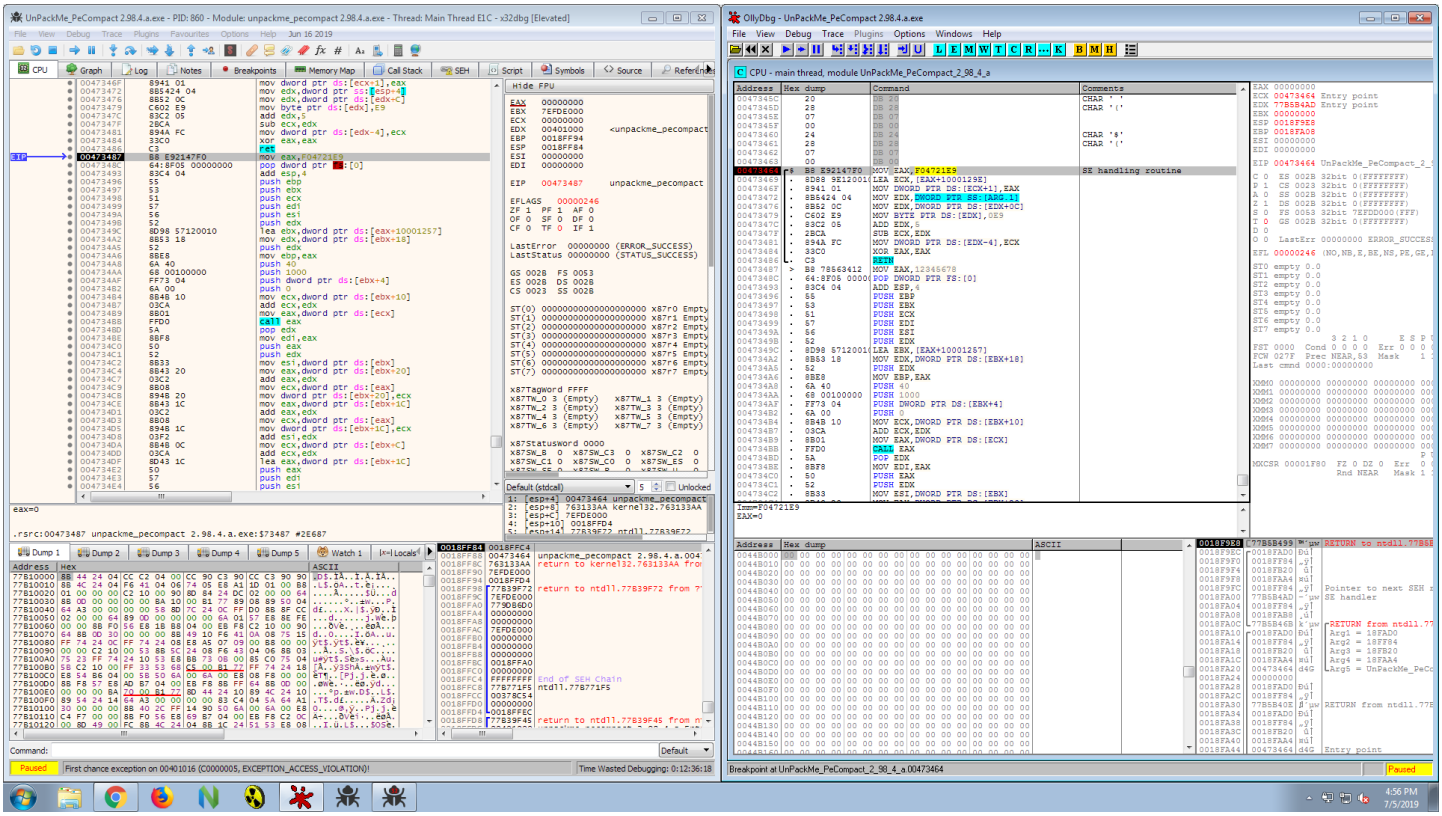
Another way to find the real code is to ultilize the `pushad` and `popad` instruction used by UPX. UPX will store all register value before decompressing and retrive those back after it is finished. UPX uses two special instructions for that purpose, we can search for `popad` to see when UPX is finished, and keep stepping.

Another way to find the real code is using the ESP trick. When `pushad` is called, the registers are push to the stack, and will be poped out on `popad`. The trick is to watch for the value of register if they are accessed. The order of register when push is `EAX, ECX, EDX, EBX, EBP, ESP, EBP, ESI, EDI`. We will monitor this address space to see if it get accessed, poped out. The moment the address for these is use is just right after `popad`. Stepping a few more and we hit our real code.



PeCompact

PeCompact does not use `pushad` and `popad`, but using Exception Handler to jump to the decompressing code. What special about Exception Handler is that when debugging the debugger will catch the exception. In the image below, the `eax` register is having the value `0x00000000` and the `EIP` is at `mov dword ptr ds:[eax], ecx`. The address at `0x00000000` is not accessible, so when this command is executed, an exception is thrown. When debugging, the debugger will catch the exception and we cannot go into the exception handler. I have tried to debug with Ollydbg but to no avail, while using x32dbg we can press `Shift+F7` to pass the exception to program. In Ollydbg, we can set break point on SEH but instructions are not correct and cannot debug to find the real code. In x32dbg we can find the real code just by stepping forward.



ASPack

ASPack flow is just like UPX, a `pushad` before and a `popad` after decompressing. Follow the trail and we can find real code.

Dumping the decompressed binary

When we find the real code, we can now dump the binary out. However, the dumped binary will not work correctly, because the Import Table are not located at the right offset. We have to fix the IAT table. The whole process can be done by using Scylla plugin of x32dbg.

Anti-debugging

Anti-debugging is a technique to somehow detect debuggers. The main goal is to prevent others people reversing the program. These technique often can be bypass if we have found where the call is made because it is often followed by a jump to exit code. Patching the jump to bypass the check.

```
call CheckDebugger
test eax, eax
jne ExitCode
```

However, these code can be hide somewhere along the run path, this is hard to find all of them. Often we will create a library hook to make the call to those API useless. In this paper, we will only deal with small binary with one execution path.

Windows API Anti-Debugging

Windows provides programmers ways to detect the debugger from a public API. The API are:

```
BOOL IsDebuggerPresent();
BOOL CheckRemoteDebuggerPresent(HANDLE hProcess, PBOOL pDebuggerPresent);
```

Some others API are not mentioned in the documentation of Windows, or not importabled. These API can be called by specify the address offset from the library.

```
__kernel_entry NTSTATUS NtQueryInformationProcess(
  IN HANDLE           ProcessHandle,
  IN PROCESSINFOCLASS ProcessInformationClass,
  OUT PVOID           ProcessInformation,
```

```
  IN ULONG            ProcessInformationLength,
  OUT PULONG          ReturnLength
);
```

The most common API is `NtQueryInformationProcess`, this API was not documented in the official Windows documentation, but later included but we cannot use. If we tried to directly call `NtQueryInformationProcess`, it will throw an error of `unknown import` when compiling. We can still call the function by write a function pointer for it and call, or call it in assembly.

```
int main(int argc, char** argv) {
  // code
  HANDLE process = GetCurrentProcess();
  DWORD isDebuggerPresent = 0;

  PVOID NtQueryInformationProcess = (PVOID)GetProcAddress(GetModuleHandleA("ntdll.dll"), "NtQueryInformationProcess");
  if (NtQueryInformationProcess == NULL) {
    printf("Failed\n");
    return 1;
  }
  __asm {
    push ecx
    push eax

    mov ecx, NtQueryInformationProcess
    lea eax, [isDebuggerPresent]
    push 0                ; NULL
    push 4                ; DWORD size
    push eax
    push 7                ; DEBUG PORT
    push process
    call ecx              ; NtQueryInformationProcess(process, 7, &isDebuggerPresent, 4, NULL)

    cmp isDebuggerPresent, 0
    pop eax
    pop ecx
    jne debuggerfound
  }
  // good code
  return 0;
debuggerfound:
  // bad code
}
```

THE PEB

One of the most common way to detect if the program is run in a debugger is to query the information from the PEB structure of the process. This structure is located inside of the TEB structure stores information about the threads. TEB is always at the `fs` segment of the process, looking at the offset, PEB is at 0x30 of TEB. The assembly for getting the PEB is:

```
lea eax, dword ptr fs:[30]
```

A few use case of PEB to detect debugger are:

- PEB[0x2], BeingDebugged
- PEB[0x18], ProcessHeap
- PEB[0x68], NtGlobalFlag

`BeingDebugged` is check when using legacy API `IsDebuggerPresent()`. `ProcessHeap` is a structure to heap information, two member `ForceFlags` and `Flags` are used to detect debugger. These values should be zero on a clean run. `NtGlobalFlag` is a flag `DWORD?`, where each bit in the word represent the status of something, here, if being debugged, the values of `NtGlobalFlag` should be `0x70`, where `FLG_HEAP_ENABLE_TAIL_CHECK` 0x10, `FLG_HEAP_ENABLE_FREE_CHECK` 0x20, `FLG_HEAP_VALIDATE_PARAMETERS` 0x40.

A straight way to detect this is possibly the load `fs:[30]`. But sometimes checks aren't perform right away, the address to PEB could be store for later use.

```
PEB* peb;

__asm {
  lea eax, dword ptr fs:[30]
  mov peb, eax
}

// do something

__asm {
  mov eax, peb
  mov ecx, [eax + 2]
  test ecx, ecx
}
```

Runtime checks

These checks are performed on runtime, either check running time or checksum. Running time check to see if a debugger slow the process are not. Checksum to detect if changes or patches has been applied to the application or not. When debug, software breakpoint often change the instruction to `0xCC` and trigger an interruption to stop execution, then replace back the correct command to run. Changed code will not pass through checksum, but we can always skip or patch the check right after.

Exception to detect debugger

This technique will trigger an exception, if handle wrongly, we cannot debug the program. There are three common type of exception based anti debug. The `0xCC` int3 interrupt, int 2d interrupt, standard exception handler (SEH) or vectored exception handler (VEH) interruption handling. As stated above, `0xCC` is often used by debugger to stop the execution, and then replace the command. But if the command is `0xCC`, the program run into an exception to stop, the debugger stop and replace the command, and loop. But both Ollydbg and x32dbg has a way to skip int3 command, or we can change `eip`. In the case of int 2d, it could cause a byte jump (called byte scission), if we put some code to modify something in this one byte, we can check if there is a debugger attached. Exception instruction stop the debugger, but when running outside of debugger, the process will crash, because it hit an interruption. To prevent this from happening, using int3 and int 2d anti debugging techniques often use Standard Exception Handler or Vectored Exception Handler to resolve the exception on clean run. Mostly we see SEH rather than VEH, because VEH is a new Exception Handler, while malware often write to be used in older system, thus we often see SEH.

We could only cover the basics of SEH only. SEH is a function that resolve when an exception occurs, it should be like this:

```
 EXCEPTION_DISPOSITION
 __cdecl _except_handler(
  struct _EXCEPTION_RECORD *ExceptionRecord,
  void * EstablisherFrame,
  struct _CONTEXT *ContextRecord,
  void * DispatcherContext
);
```

Where `_EXCEPTION_RECORD` is a structure that contains the error reporting to the kernel:

```
typedef struct _EXCEPTION_RECORD {
  DWORD ExceptionCode;
  DWORD ExceptionFlags;
  struct _EXCEPTION_RECORD *ExceptionRecord;
  PVOID ExceptionAddress;
  DWORD NumberParameters;
  DWORD ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
}  EXCEPTION_RECORD;
```

And we can have the context when the exception is thrown

```
typedef struct _CONTEXT
{
  DWORD ContextFlags;
  DWORD Dr0;
  DWORD Dr1;
  DWORD Dr2;
```

```
  DWORD Dr3;
  DWORD Dr6;
  DWORD Dr7;
  FLOATING_SAVE_AREA FloatSave;
  DWORD SegGs;
  DWORD SegFs;
  DWORD SegEs;
  DWORD SegDs;
  DWORD Edi;
  DWORD Esi;
  DWORD Ebx;
  DWORD Edx;
  DWORD Ecx;
  DWORD Eax;
  DWORD Ebp;
  DWORD Eip;
  DWORD SegCs;
  DWORD EFlags;
  DWORD Esp;
  DWORD SegSs;
} CONTEXT;
```

Manipulating registers when throwing an exception and check if the value is correct is also an anti debugging technique.

SEH exception are set using `fs:[0]`, the routine is as follows:

```
push ExceptionHandlerFunction
push dword ptr fs:[0]
mov dword ptr fs:[0], esp
; normal code
; unset exception handler if necessary
```

In both Ollydbg and x32dbg, we can see the SEH list, when we detect a new SEH function, put a breakpoint to prevent the program running past the exception handler.

Others

While the above are common, but the list of anti debugging goes on. It could check for process running, using one byte skip of debugger `rep stosb` / `rep movs`. When dealing with the executable, we can first static analyze the code and patching out anti debug one if we can, or else, debug and detect on the spot. Anti debugging could be self-modifying code, or encrypted code, obfuscated code, these are advanced techniques and often are resolved on runtime.

Anti Virtualization

Besides anti debugging, anti virtualization is very important for malware writer. Reverse engineers do not run untrusted executable on his own machine, but using a separate environment, either through virtualization or emulation. The concept of two are different, in this paper we discuss only virtualization, the term describe the action of running a Virtual Machine on a host machine. Programs providing virtualization are VMWare, VirtualBox, Virtual PC (old Windows?) to name a few. These application provide us a separate environment by simulating the hardware, and many other stuff. Malware will use these characteristic to detect whether the process is running inside a virtual machine.

Most anti virtualization techniques targeting on finding VMWare, due to VMWare popularity in the industry. The simplest of all is to use instruction to query the machine information. Then to reading files, devices created by Virtual Machine. Registry values specific to virtual machine too. There also an address validation of Interupt Descriptor Table (IDT), Local Descriptor Table (LDT), and Global Descriptor Table (GDT).

One can use CPUID instruction with `eax = 1` to query the information of the machine, if the 31st bit of `ecx` is 1 then VM is detected. CPUID with `eax = 40000000` will return the string of the vendor name to `ecx, edx`. This method can be patched by edit the config file of the VM, `.vmx`.

```
cpuid.1.ecx="0---:----:----:----:----:----:----:----"
cpuid.40000000.ecx="0000:0000:0000:0000:0000:0000:0000:0000"
cpuid.40000000.edx="0000:0000:0000:0000:0000:0000:0000:0000"
```

Virtualization Program often have files created in machine, in VMWare the common one are `vmmouse.sys` and `vmhgfs.sys` located in `system32` of Windows. Some other files are:

- C:\windows\System32\Drivers\Vmmouse.sys
- C:\windows\System32\Drivers\vm3dgl.dll
- C:\windows\System32\Drivers\vmdum.dll
- C:\windows\System32\Drivers\vm3dver.dll
- C:\windows\System32\Drivers\vmtray.dll
- C:\windows\System32\Drivers\VMToolsHook.dll
- C:\windows\System32\Drivers\vmmousever.dll
- C:\windows\System32\Drivers\vmhgfs.dll
- C:\windows\System32\Drivers\vmGuestLib.dll
- C:\windows\System32\Drivers\VmGuestLibJava.dll
- C:\windows\System32\Driversvmhgfs.dll
- C:\windows\System32\Drivers\VBoxGuest.sys
- C:\windows\System32\Drivers\VBoxSF.sys
- C:\windows\System32\Drivers\VBoxVideo.sys
- C:\windows\System32\vboxdisp.dll
- C:\windows\System32\vboxogl.dll
- C:\windows\System32\vboxoglarrayspu.dll
- C:\windows\System32\vboxoglcrutil.dll
- C:\windows\System32\vboxoglerrorspu.dll
- C:\windows\System32\vboxoglfeedbackspu.dll
- C:\windows\System32\vboxoglpackspu.dll
- C:\windows\System32\vboxoglpassthroughspu.dll
- C:\windows\System32\vboxservice.exe
- C:\windows\System32\vboxtray.exe
- C:\windows\System32\VBoxControl.exe

> In company Virtual Machine, there was no `vmmouse.sys` nor `vmhgfs.sys`

Or some services running background

- Vmtoolsd.exe
- Vmwaretrat.exe
- Vmwareuser.exe
- Vmacthlp.exe
- vboxservice.exe
- vboxtray.exe

Or some registry keys

- HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0 (Identifier) (VBOX)
- HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0 (Identifier) (QEMU)
- HARDWARE\Description\System (SystemBiosVersion) (VBOX)
- HARDWARE\Description\System (SystemBiosVersion) (QEMU)
- HARDWARE\Description\System (VideoBiosVersion) (VIRTUALBOX)
- HARDWARE\Description\System (SystemBiosDate) (06/23/99)
- HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0 (Identifier) (VMWARE)
- HARDWARE\DEVICEMAP\Scsi\Scsi Port 1\Scsi Bus 0\Target Id 0\Logical Unit Id 0 (Identifier) (VMWARE)
- HARDWARE\DEVICEMAP\Scsi\Scsi Port 2\Scsi Bus 0\Target Id 0\Logical Unit Id 0 (Identifier) (VMWARE)
- SYSTEM\ControlSet001\Control\SystemInformation (SystemManufacturer) (VMWARE)
- SYSTEM\ControlSet001\Control\SystemInformation (SystemProductName) (VMWARE)
- HARDWARE\ACPI\DSDT\VBOX__ (VBOX)

- HARDWARE\ACPI\FADT\VBOX__ (VBOX)
- HARDWARE\ACPI\RSDT\VBOX__ (VBOX)
- SOFTWARE\Oracle\VirtualBox Guest Additions (VBOX)
- SYSTEM\ControlSet001\Services\VBoxGuest (VBOX)
- SYSTEM\ControlSet001\Services\VBoxMouse (VBOX)
- SYSTEM\ControlSet001\Services\VBoxService (VBOX)
- SYSTEM\ControlSet001\Services\VBoxSF (VBOX)
- SYSTEM\ControlSet001\Services\VBoxVideo (VBOX)
- SOFTWARE\VMware, Inc.\VMware Tools (VMWARE)
- SOFTWARE\Wine (WINE)
- SOFTWARE\Microsoft\Virtual Machine\Guest\Parameters (HYPER-V)

Lists taken from al-khaser and cyberbit.com.

When running inside VMWare, machine has to communicate with the host. A reverse engineer has found the port and publicly announced it, the port is called a backdoor I/O port. We can use `in` instruction to read from the port and expect the result.

```
mov eax, 0x564d5868   ; magic number
mov ebx, 0            ; function param
mov ecx, 0xA          ; function id
mov edx, 5658         ; VMWare I/O Port
in eax, dx            ; try to communicate
; cmp ...
```

The above code will call the function with id `0xA`, fetch VMWare version number, and return the result on `eax, ebx, ecx`. Some other command are also useful.

Another way is verifing OS Table addresses. These tables (IDT, LDT, GDT) are OS specific tables, because two machine is run on the same machine, VMWare map the address of those tables to another location. Through Joanna Rutkowska researches' the author has shown us that we can check relative to one address. This method's name is Red Pill, inspired by the movie Matrix.

Alfredo Andr´es Omella wrote in his paper a method to use STR instruction to detect virtualizaion. STR instruction stores segment of the task register, and the value returned are different when running on host to running in virtualization.

There are many virtualization detection techniques we only cover some common and simple one. Detecting Virtualization is very important to malware writer, and finding them are jobs for a malware analysis, because we should never run untrusted executable in a host machine.

Summary

Unpacking, Anti debugging, Anti Virtualization are three techniques that almost every malware use. Some techniques are easy, some are harder, and for most malware they are all tricky. Unpacking pure packer would not require much time, but unpacking a moded packer or a manualy crafted packer takes time. For unpacking ELF there's a popular talk by unixfreaxjp about `Unpacking the non-unpackable` in r2con2018. Researching about anti debugging and anti virtualization is a field in reverse enginnering also. More resources can be found on `fumalwareanalysis`. There is a blog by waliedassar about anti debugging and anti virtualization. The list of common anti debugging is published publicly online by Peter Ferrie `The "Ultimate" Anti-Debugging Reference`. The list makes no sense without code, al-khaser is a program written by LordNoteworthy to test those anti debug and anti virtualization. OALabs has many videos about unpacking, dumping binaries, rebuilding headers/PE, to detect anti debugging, and detect anti virtualization on realworld malware.