Further Down the VM Spiral
Detection of full and partial emulation for IA-32 virtual machines

Danny Quist dannyquist@gmail.com
Val Smith mvalsmith@gmail.com

Offensive Computing
http://www.offensivecomputing.net/

Presented at Defcon 14
Las Vegas Nevada, USA

August 4, 2006

Introduction

Generic detection of virtual machines is possible using a variety of methods. As we outlined in our previous paper, there are several methods that are available using special pseudo-privileged operations running in ring-3, or user mode. There were several problems with this detection mechanism. First, in a fully emulated environment it was possible to trick the local descriptor table (LDT) [1, 2] detection method into thinking it was on raw hardware. Second, on multiprocessor systems, some of the existing methods are inaccurate. We have developed further methods of detecting the presence of a virtual environment. We will also present an amalgamated approach to determining signatures for various virtual environments.

Obfuscation and Avoidance

Modern malware is getting more difficult to reverse engineer. There are many different methods for the malware writer to prevent analysis of a particular piece of code. The three methods employed by malware writers that are discussed here are obfuscation, anti-debugging, and virtual machine detection.

Packing and Obfuscation

There are a variety of methods for obfuscating and/or compressing compiled binaries. The goal of these methods is to prevent analysis or reverse engineering and to provide a smaller disk footprint. Obscuring things like strings, the imports address table (IAT) and function calls make understanding a binary much more difficult. Implementations of these methods range from using the well-known UPX packers to the more advanced engines like morphine, Shiva and Telock and several different pieces of malware even use custom decoders. These methods work by implementing an algorithm which runs over the original binary code and either compresses it, encodes it or both. A decoder stub is then usually included somewhere in the binary and the entry point is reset to the location of this stub. When the decoding routine completes it sends execution back to the Original Entry Point of the binary.

A variety of techniques are available for bypassing these methods. In some cases un-packers are available which restore the binary to a de-obfuscated state. In other cases pure static analysis of the binary is performed and custom decoder stubs can be written if the algorithm is two way (reversible). Finally in extreme cases of complex obfuscation run time analysis is required in order to de-obfuscate the binary using techniques like memory region dumping, emulation, etc. These scenarios have been well covered in other areas.

Debugger Detection Methods

One of the goals of many types of malware is to avoid detection, and avoid analysis by reverse engineering. There are several methods used by malware to detect the presence of a debugger as well as techniques for bypassing these methods.

The first method is employed by using the thread execution block (TEB) debugging flag. This is the same method used by the IsDebuggerPresent() windows API code. Sample C code to look for this is as follows:

```
if (IsDebuggerPresent())
{
	printf("Debugger Present\n");
}
else
{
	printf("No debugger present\n");
}
```

The disassembly for the IsDebuggerPresent() reveals the following:

```
_IsDebuggerPresent@0:
7C812E03  mov         eax,dword ptr fs:[00000018h]
7C812E09  mov         eax,dword ptr [eax+30h]
7C812E0C  movzx       eax,byte ptr [eax+2]
7C812E10  ret
```

The effect of the call is to first load the TEB into a register at location fs:[00000018h]. The debug bit is then moved to the eax register which returns the value to the underlying debugger call. To circumvent this sort of detection, the memory can be modified prior to runtime analysis. Below is example code for modifying this memory:

```
void nonDebugHijack(void)
{
	// set the TEB to non-debug mode
	__asm
	{
		mov eax, dword ptr fs:[00000018h];
		mov eax, dword ptr [eax+30h];
		mov ebx, eax;
		add ebx, 2;
		mov eax, 0xFFFF0000;
		mov [ebx], eax;
	}

}
```

This effectively reverses the value of this check and will cause the debugger code to continue on as if no debugging is present. The debugger will continue to function normally, even if this value is toggled. Setting an IDA breakpoint on the IsDebuggerPresent() API call is a good alternative as well. [8]

Another popular method of malware preventing debugger access to an executing piece of code is to stimulate exceptions in the code. On some debuggers, such as the stock configuration of IDA Pro, this will cause the execution to stop. Ilfak Gulinov has written a module to help with the circumvention of these tricks. [7]

A final example is that some malware take constant checksums of their own image. If an interrupt (int 3) or breakpoint is called the image changes and the checksum will fail. The malware can then take action to avoid further analysis.

Virtual Machine Detection

Virtual machine detection is another growing field of development for malware researchers. Analysts examine malware inside of virtualized environments in order to protect the host system from infection. Snap shots of different stages of the malware are frequently employed as well. Virtualization provides protection for the host but also contains some risk due to the potential for detection. Since the Intel IA32 and IA64 platforms were not initially designed for virtualization [3] implementing them involves handling many of the incompatible instructions. In order to accommodate virtualization, any of the non-hardware virtual machines must emulate or translate these instructions. There are several well-known methods for detecting this virtualization.

Basic Detection Methods

Often malware will employ very simple techniques for detecting vmware, such as:

- Searching the registry for vmware related strings
- Searching the services table for vmware strings
- Walking the process table and detecting vmware related processes

Native API Detection Methods

The first native API detection method is to use special IO port communication methods unique to the VMWare line of virtual machines. This can be demonstrated in the following code:

```
__try
{
      __asm
      {
            mov    eax, 'VMXh'
            mov    ebx, 0; // any value but not the MAGIC VALUE
            mov    ecx, 0xA // get VMWare version
            mov    edx, 'VX'  // port number

            in     eax, dx; // read port
            cmp    ebx, 'VMXh' // is it a reply from VMWare?
            jne      notVmware
            jmp    isVmware
            notVmware:
            mov rc, 0
            jmp done
            isVmware:
            mov   rc, eax // on return EAX returns the version
            done:
      }
}
__except(EXCEPTION_EXECUTE_HANDLER)
{
      rc = 0;
}
```

This illustrates the usage of the proprietary VMWare protocol. This method will consistently return correctly provided Vmware is present on the local machine. The protocol for VMWare has been reverse engineered. []A similar method exists for detecting Microsoft's Virtual PC:

```
__try
{
   __asm
   {
      mov  ebx, 0; // It will stay ZERO if VPC is running
      mov  eax, 1; // VPC function number

      // call VPC
      __emit 0Fh;
      __emit 3Fh;
      __emit 07h;
      __emit 0Bh;

      test ebx, ebx;
      setz [rc];
   }
}
__except( IsInsideVPC_exceptionFilter(GetExceptionInformation()) )
{
   rc = 0;
}
```

Generic Detection Methods

As was demonstrated by two other methods via redpill [4] and scoopy_doo [2], it is possible to detect the presence of a virtual machine generically. These are a simple low-cost technique which checks for a simple signature. In our previous paper we demonstrated that the accuracy of these methods is suspect on parallel machines unless using the Local Descriptor Table LDT method. [1]

Furthering Generic Detection

Problems with the Local Descriptor Table

After releasing our previous paper, it was pointed out to us that the virtual machine detection mechanism could be subverted in VMWare by disabling acceleration. This has the effect of turning the virtual machine from a partially emulated environment to a fully emulated instruction set. This has the effect of removing the possible signature method used by the local descriptor table method, as the LDT value then mimics actual hardware.

The IDT problems on SMP machines discussed in our previous paper [1] still manifest themselves using the fully emulated environment. Given the work of J.S. Robin [3] we set out to find other ways to overcome this problem. The other virtualization problems mentioned in the Usenix paper were a starting ground for this. Given the need for instruction level virtualization, the IDT, LDT, and GDT methods did not have consistent results. Without entering the kernel space, it did not appear possible to detect an emulated environment.

Testing Methodology

The Usenix virtualization problem [3] highlights the issues with implementing a virtual machine on the Intel architecture. We took the steps to narrow down a method to detect the fully virtualized (fully emulated or accelerated) instruction sets. There were three tests we performed on the software.

The first was to run nopill tool [1] on raw un-emulated hardware. This test was performed on Intel and AMD hardware, in configurations up to four processors. The next step was to test this using the VMWare tool with acceleration turned on. This method has VMWare emulate the Intel instructions which cause problems for virtualization. Virtual PC falls in the category of only allowing a partially emulated virtualization format. The last method was to test with VMWare with acceleration turned off. This causes VMWare to fully emulate all the instructions.

We tested each of these environments with all of the instructions mentioned in the Usenix overview and found that the machine status word (MSW) yielded useful results.

The Machine Status Word Test

Given the three environments we noticed the following outcomes. First, the IDT, GDT and LDT values follow the previously discussed results. The machine status word (MSW) turned out to be the key for detecting changes in the fully vs. partially emulated environment. There is no demonstrable change in the MSW on raw hardware vs. partially emulated hardware. When running on fully emulated hardware, the value of the MSW differs. This allows us to close the gap in detecting virtualization.

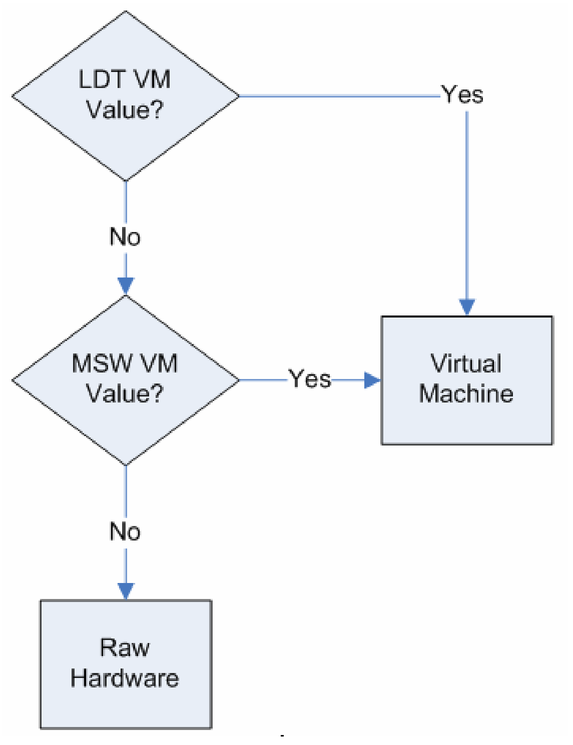

Figure 1: Decision flow for determining virtualization

Conclusion and Future Work

Using this method it is possible to fully detect without flaw the presence of a virtual machine. Given the low cost of this decision point, these are a vector for a piece of malicious software to determine its root operating environment. Some of these problems may not be able to be solved by the various virtualization products. We suspect that the MSW issue is one possible avenue that could be fixed. In practice, however, supporting malicious software research has not been a strong market for the virtualization vendors.

Since this work is all based upon detection at the user level (ring-3), there exists the possibility that many of the privileged operations could yield interesting results in kernel space (ring-0).

References

1. “Generically detecting the Presence of Virtualization”, D. Quist, V. Smith
http://www.offensivecomputing.net//files/active/0/vm.pdf
2. Scoopy_doo, T. Kline
http://www.trapkit.de/research/vmm/scoopydoo/index.html
3. “Analysis of the Intel Pentium’s Ability to Support a Secure Virtual Machine Monitor”, Proceedings of the 9th USENIX Security Symposium, J.S. Robin, C.E. Irvine
4. “Red Pill: or how to detect VMM using (almost) one CPU instruction” J. Rutkowska
http://www.invisiblethings.org/papers/redpill.html
5. Intel Developers Manual
http://www.intel.com/design/pentium4/manuals/index_new.htm
6. VMWare Product Description
http://www.vmware.com/products/ws/
7. Stealth Plugin, Ilfak Guilfanov
http://www.hexblog.com/2005/11/stealth_plugin_1.html
8. Simple Trick to Hide the IDA Debugger, Ilfak Guilfanov
http://www.hexblog.com/2005/11/simple_trick_to_hide_ida_debug.html
9. VMWare Backdoors, K. Kato
http://chitchat.at.infoseek.co.jp/vmware/backdoor.html
10. Detect if your program is running in a virtual machine, Lallous
http://www.codeproject.com/system/VmDetect.asp