Supporting Material

In the supporting material for this paper, I will provide specific examples which illustrate how easy it is to tell whether or not a program is running in a default VM. Primarily, these examples look for certain red flags that indicate that a computer may be running in a VM. For these examples, we will be looking at VMware, but there are similar techniques for various other popular virtual machine emulators. (Note: All examples of source code are running under the assumption that <`windows.h`>, <`excpt.h`>, and <`stdio.h`> are imported, compiled under 32bit Windows, and if a VM is running, it's under VMware).

(Next page)

Red Pill

Here is one of the oldest techniques, all the way back from 2004. It is called the Red Pill and works by checking specific information about the Interrupt Descriptor Table. Namely, on VMware, the IDT's pointer typically starts with "0xff_____" but when running on Windows host, it is located lower than that at "0x80_____". So, to tell if you are in VMware, just check the first byte of this pointer:

```
void red_pill() {
  unsigned char idtr[6];
  unsigned long idt = 0;

  _asm sidt idtr //assembly instruction to input the IDT register val
  idt = *((unsigned long *)&idtr[2]);

  if ((idt >> 24) == 0xff) { //or 0xe8 for VirtualPC, or etc...
    printf("VMware Detected!\n");
  } else {
    printf("Native OS\n");
  }
}
```

Primarily, this is just a trick that you can pull if you know certain very specific knowledge about how VMware works. Many of the best techniques employed by malware work based on the same idea; gather telling information from obscure sources (namely, the *implementation* of the VM), and use that information to conditionally branch to either do nasty malware things or feign doing normal program instructions.

Variants of the Red Pill look at the locations of other tables in memory, such as the Global Descriptor Table (GDT) and Local Descriptor Table (LDT). More can be found about these other techniques in the presentation by Liston/Skoudis (in *References*).

VMDetect

Another fairly easy method to detect a VMware environment is a method called VMDetect. In short, all it does is ask the computer for the current version of VMware that's running. This turns out to be surprisingly easy! You don't even need VMware APIs, you just need to know who to ask, what to ask for, and how to ask it. If there is no VMware running (aka, if you're on a host computer), then an exception is raised, caught, and it is assumed that the program is not being run in a VMware VM. Here's the code:

```
void vm_detect() {
  unsigned a=0, b=0;
  __try {
    __asm {
    // Check for VMware version
      mov eax, 'VMXh'         // VMware magic value
      mov ecx, 0Ah      // "Get VMware version" command
      mov dx, 'VX'      // VMware I/O port to get the version from
      in eax, dx              // <-- HERE IS THE CHECK FOR VMware
      mov a, ebx              // assign a and b
      mov b, ecx
    }
  } __except (EXCEPTION_EXECUTE_HANDLER) {} // if we are outside of VMware, a
                                  // privilege error occurs and 'a' remains 0
  if (a == 'VMXh') { // equal to the VMware magic value?
    printf("VMware ");
      switch(b) { // VMware version stored by the 0Ah function call
         case 1:
            printf("Express"); break;
         case 2:
            printf("ESX"); break;
         case 3:
            printf("GSX"); break;
         case 4:
            printf("Workstation"); break;
          default:
            printf("(unknown version)");
        }
       printf(" detected!\n");
  }
  else printf("Native OS\n");
}
```

Again, it all boils down to knowing specific implementation details about VMware and abusing that to gain information about how to proceed.

Naturally, there exist ways to thwart this simple program with direct manual configuration of the Virtual Machine's environment settings (or using less common virtualization platforms). Similarly, there are more advanced VM detection methods to see through these settings changes. Again, it's a game of cat-and-mouse between the malware producers and the "good guys."

Evidently, basic VM detection is rather easy if you know what to ask for. According to Intel security researchers Barbosa and Branco, approximately 79.7% of malware caught in their sample of 12 million batch analysis uses similar techniques to those expressed here (abusing the `IN` command to read data from VMware or other virtualization engines). Because of this, companies are, for example able to use static analysis to try to find "suspicious" calls to `IN`, resulting in a fairly high proportion of catches. However, these instruction sequences may be obfuscated by packers, code obfuscation, and other nefarious techniques. But still, over 20% of VM detecting malware would be undetected with this check, and in few years the number could be down to 30-40% of malware using this technique for VM detection, just because malware producers are realizing that AV companies are starting to perfect detecting this strategy. In fact, this has been the recent trend; from their analogous sample in 2012, 99.43% of Anti-VM programs used the IN instruction to tell whether or not they were in a VM, and now it's down to 79.7%. Again, malware is always mutating and changing based on threats to its success. As more VM detection techniques become easier to implement and harder detect, creators will likely shift away from the ever more archaic methods pioneered by programs such as Red Pill and VMDetect, as open-sourced and illustrated by researchers like Tom Liston and Ed Skoudis of Intelguardians, for the good of the community.

References

Liston, Tom and Skoudis, Ed. "On the Cutting Edge: Thwarting Virtual Machine Detection." *Intelguardians Research*. <http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf>

Klein, Tobias. "Trapkit - ScoopyNG, The VMware Detection Tool." *NESO Security Labs*. <http://trapkit.de/tools/index.html>