Editors: Richard Ford, rford@se.fit.edu
Michael Howard, mikehow@microsoft.com

How Not To Be Seen

Richard Ford and William H. Allen
Florida Institute of Technology

The concept of stealth—as it pertains to computers—shares a great deal with its real-world counterpart. As *Monty Python's Flying Circus* illustrated so brilliantly many years ago, stealth is all about "how not to be seen" (http://en.wikipedia.org/wiki/How_Not_To_Be_Seen).

In this column, we take a look at stealth from both a historical and a technological perspective. This is a hugely important topic, for if an unwanted computer program can't be seen, it can't be eliminated. In addition, software developers—especially security software developers—must have a solid understanding of what can be trusted in an environment—and what can't. When it comes to deception, stealth is the state of the art.

The need for stealth

Stealth techniques in software all relate to hiding information. A stealthy program, for example, could hide files on the hard drive, processes in the tasklist, or logins to a particular computer. As such, stealth techniques attempt to convince an observer that the world is some other way than it actually is.

The first faulty assumption we must do away with is the idea that being stealthy is always bad; in fact, there are several legitimate uses for stealthy programs on a computer. For example, if a vendor wishes to hide its software from prying eyes, stealth can be a useful technique. Both digital rights management (DRM) and security vendors wish to make the attacker's job as difficult as possible. Although using stealth to do this is clearly (at some level) security through obscurity, many would argue that it can play an important role in at least slowing down a reverse engineer. Several in the field have argued vehemently that understanding stealth techniques is a crucial step toward understanding an attacker's true capabilities. As such, creating stealth programs to understand what's possible might be a useful exercise.

Despite these claims of legitimacy, culturally, software stealth is primarily associated with computing's "dark side"—in particular, viruses, rootkits, and malicious code. For the remainder of this article, we refer to stealthy software as *malcode*, although this is purely a convenience: we acknowledge that stealthy software might not contain any malicious components.

Passive vs. active

Although stealthy computer software sounds terribly complex, in actuality some forms of limited stealth are very easy to implement. At the simplest level, marking a file as hidden in a FAT file system provides some level of (not very useful) stealth as it "hides" a file from normal view. Techniques like this are loosely called *passive* because they simply use the operating system's features to make a particular file's presence or process harder to spot. Other examples of passive stealth include co-locating a remote thread in a normal application, or using alternate data streams on NTFS file systems. In both cases, casually examining the environment is likely to leave the user with a false representation of its actual state.

When using passive techniques, malcode basically lets the underlying operating system or command shell operate exactly as designed. It does nothing to actively make the results of system calls misleading. As a result, it doesn't provide a great deal of stealth—anyone who understands how the operating system is designed to function won't be misled. Far more interesting is *active* stealth, in which the malcode actually modifies the way the computer functions to meet its own ends.

Before tackling this topic, it's worth revisiting the actual need for stealth in malicious software. Hiding is useful, but sometimes simply blending in might seem like an option—after all, what normal user can account for all files on his or her computer? One or two extra processes and files would be lost among the noise. Although this argument seems to hold water, it ignores why malcode uses stealth: in addition to hiding from the user, the malicious code hides from other software that's actively searching for it, either specifically or generically. Thus, although passive stealth, or even simply "hiding in plain sight" seems on the surface sufficient, it's anything but.

Active deception

Historically, active stealth isn't a particularly new idea. Even very early

Some hackers (both black and white hat) have already developed techniques that claim to circumvent Vista's protective mechanisms.

viruses could hide their presence on the disk. Back in the DOS days, such stealth was relatively easy to implement: a virus could simply hook an interrupt and handle requests that would reveal its presence. Figure 1 shows a boot-sector virus "hooking" INT 13h (the BIOS disk interrupt). Thus, when a program attempts to read the disk at a sector level, the virus surreptitiously modifies the call to return a "clean" disk view. Such deception isn't difficult to implement because the machine's host programs have fairly limited ways of accessing the underlying hardware, and the entire interrupt system explicitly supports interception of calls.

This idea can be extended dramatically to deceive at a more granular level. For simplicity's sake, we use an illustration from an old MS-DOS version; although the details of the technique are different for other operating systems, the basic principals apply.

Consider a MS-DOS-based virus that wished to hide changes made to file size, date, time stamps, and even file contents. In MS-DOS, most operating system services are provided by a single interrupt—INT 21h. When such an interrupt request is encountered (in the form of an INT 21h instruction), the CPU looks at the INT 21h vector stored in memory and passes control to the location to which this value points. If a program modifies this vector, control can be passed anywhere. To "stealth" almost any MS-DOS API, all the virus needs to do is redirect the INT 21h vector to point to its own code. It then effectively sits between the program requesting service and the underlying operating system.

Virus scanners have traditionally dealt with such stealth by checking the memory contents for viruses. If a virus is known, simply checking the memory for telltale interception signatures or virus code—at least under simple operating systems like DOS—can indicate its presence. Unfortunately, life gets complicated in more modern environments.

32-bit deception

As operating systems became more sophisticated, operating system vendors placed more emphasis on isolating one program's effects on another. Hence, Win32 helped bring the idea of different privileges to the typical computer user. The world was broken up into user-mode components—programs the user typically executes, which run on top of the host operating system—and kernel-mode programs that actually load into the operating system and have unrestricted access to its internal structures.

Simplistically, user-mode stealth is analogous to the DOS situation: user-mode programs can intercept system calls (typically, calls to the Win32 API) and modify their return codes before returning them to the calling process. Thus, a Windows virus can easily hide its presence from other user-mode programs. Although virus-specific detection techniques generally find known viruses in memory (and therefore render such stealth impotent), certain generic detection techniques can fall prey to stealth in such an environment. Running an integrity checker in user mode on an infected machine, for example, is a pointless exercise if the virus implements user-mode stealth.

Because user-mode interception is easy to implement, both malcode authors and antivirus vendors make full use of kernel-mode interception in which the stealth (and, hopefully, the detection thereof) is implemented in kernel-mode code. Although interception within the kernel can pose some technical problems for developers, correctly implemented kernel-mode stealth can be extremely effective—once malcode infiltrates the kernel, it's difficult to determine which calls have been tampered with.

Microsoft isn't oblivious to this; the recent launch of Windows OneCare and some of Vista's design are definitely attempts to protect the brand from malicious code. In particular, Vista's design attempts to make modifying the kernel difficult. Microsoft explains these features as a defense against changes that "violate the integrity of the kernel" and that lead to reliability, performance, and security issues (www.microsoft.com/whdc/driver/kernel/64bitpatch_FAQ.mspx). As such, 64-bit versions of Windows, including Vista, were developed to make it difficult for attackers to use kernel tricks to hide malicious software. Like all such software prophylactics, this approach isn't bulletproof—and isn't claimed to be. Some hackers (both black and white hat) have already developed techniques that claim to circumvent Vista's protective mechanisms.

For example, despite the introduction of device driver signing, it's still conceivable that attackers could execute rogue code in kernel mode. Similarly, a well-motivated attacker can make numerous modifications to the machine while the operating system is at rest by booting from a CD and mounting the stored operating system image. Thus, Vista isn't the end of the line for software stealth; introducing malcode just got significantly more difficult—not enough to make it impossible to see stealthy malcode, but difficult enough to raise the bar for attackers—at least for a while. The flipside of this, however,

is that it's also difficult for legitimate software to modify kernel operation. Thus, some third-party security developers object to Vista's protections; how this will develop is beyond this article's scope, but it does illustrate the double-edged nature of much of the technology and techniques we discuss here.

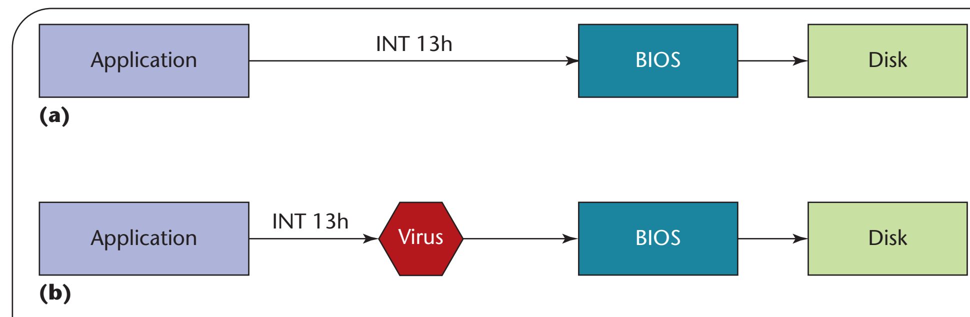

Figure 1. Interrupt interception schematic. (a) The normal functioning of an application. (b) The INT 13 vector is redirected to the virus code, giving the malcode control of the machine.

The future

One interesting development in the recent past is hypervisors—virtual machine monitors that virtualize the operating system—such as SubVirt[1] or BluePill.[2] In each case, a thin hypervisor is placed between the operating system and the underlying computer hardware. Hypervisors ultimately control of every aspect of the computer's operation.

Such hypervisors are likely to become increasingly common as CPUs begin to contain more support for machine virtualization. As convenient as virtualization technologies are for load balancing and data center simplification, they can also be turned squarely upon the user as "virtual rootkits." In such a scenario, an attacker places the entire operating system inside a virtual machine. Depending on the simulated environment's fidelity, it can be extremely difficult for software running in the virtual environment to determine the environment's real nature, as system calls must ultimately trust not the true state of the underlying hardware (which might be inaccessible from the virtual machine), but the state the hypervisor reports. Essentially, you might be able to determine the hypervisor's presence but not its intent.

In basic terms, the future for users looks pretty bleak, regardless of the operating system you use. Stealth techniques are continually evolving, and new technologies will make system-call interception easier on vulnerable end-user machines. High-end solutions are certainly possible in the corporate world (stealth has a hard time with booting the system into a known clean configuration, but how many home users will ever do that?), but unless they're made far more user-friendly, they'll go unused at home—where spyware and other malcode are a potential tunnel into corporate networks via virtual private networks.

Although most of this article's discussion has been Windows-centric, stealth techniques are in no way solely a Windows problem; they're a snare for Apple and Unix users, too. Attackers focus on Windows because of the technology's omnipresence, but the general techniques applied here are very catholic in their applicability.

Simple stealth techniques can be extremely effective when countering generic virus-detection techniques, but all fall prey to virus-specific detection because the stealth technique can be detected via signatures. However, new developments in stealth, such as virtualization, pose significant challenges to defenders; when every program call can potentially return modified data, it's difficult to determine anything about the environment with certainty. Fortunately, all isn't lost: in a future column we'll look at state-of-the-art anti-stealth technology, and see that defenders aren't without options. ◻

References

1. S.T. King et al., "SubVirt: Implementing Malware with Virtual Machines," *Proc. 2006 IEEE Symp. Security and Privacy*, IEEE CS Press, 2006, pp. 314–327.
2. J. Rutkowska, "Subverting Windows Vista Kernel for Fun and Profit," Black Hat Briefings presentation, Dec. 2006; http://blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf.

Richard Ford *is an associate professor at the Florida Institute of Technology. His research interests include malicious mobile code, spyware detection and prevention, and security metrics. Ford has a PhD in physics from the University of Oxford, England. He is a consulting editor for* Virus Bulletin *(www.virusbtn.com). Contact him at rford@fit.edu.*

William H. Allen *is an assistant professor of computer science at Florida Institute of Technology. His research interests include computer and network security, the modeling and simulation of network-based attacks, and computer forensics. Allen has a PhD in computer science from the University of Central Florida. He is a member of the IEEE, the ACM and Usenix. Contact him at wallen@fit.edu.*

For our readers:

Should "white hat" researchers write stealthy malware for test purposes? Is it okay to build a rootkit for research? We'd like to hear about some of your experiences with stealthy programs. Send your stories to Richard Ford at rford@se.fit.edu.