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

How Not to Be Seen II

The Defenders Fight Back

William H. Allen and Richard Ford
Florida Institute of Technology

In a previous column that ran earlier this year ("How Not to Be Seen," vol. 5, no. 1, 2007, pp. 67–69), we introduced readers to the varied technologies that stealthy software—that is, software that manipulates a computer in some way to avoid some aspect of its operation—uses. We broke stealth up into roughly three categories: passive, hooking, and hypervisor-based stealth. At the end of the column, we promised to take a whirlwind look at stealth detection; this installment fulfills that obligation.

It seems best to provide a brief refresher on the techniques covered—after all, 2007 has been an awfully long year for security researchers. Essentially, we can think of stealth techniques as either passive—simply using a hidden file to hide content from a casual viewer, for example—or active. For detection purposes, we typically discount passive stealth because uncovering it is merely a matter of looking in the right place. However, active stealth is another matter entirely, and it's on this approach that we focus our attention. In this column, we examine several stealth detection techniques, and (hopefully) demonstrate that the specter of undetectable stealth isn't yet upon us.

Detecting the invader's presence

Attackers implemented the concept of stealth fairly early in the virus writing world, forcing defenders to scurry to respond. At the time, virus scanners were the first line of defense for users. As such, if attackers owned the environment in which the scanner was executed, the scan results could be incorrect—or worse yet, the virus could use the `FileOpen/FileClose` system call issued by the scanner when examining the disk as a trigger to infect even more objects.

To counter this, virus scanners often scan memory for the signatures of known viruses before scanning files. If they find signs of infection, scanners typically halt or attempt to remove the virus from memory. This virus-specific approach doesn't detect stealth natively, but it's still the mainstay of most users' stealth protection today. Although sufficiently skilled attackers can in fact remain hidden from memory scans, such system calisthenics are tricky. Thus, one anti-stealth technology is simply looking for the signatures of known employers of stealth.

Detecting the hook

As we discussed in our previous column, most stealth malware hides by hooking and redirecting system calls, either at the kernel or the operating system (OS) level. This tends to leave a distinct signature at the point of the hook placement. Although it's entirely possible to avoid this, the typical malware author is generally striving for compatibility with most machines. Thus, the logical place to hook the operating system is at well-defined interfaces, such as the transition undergone when executing a `SYSENTER` instruction (the Intel fast system call that handles transitions from Ring 3 to Ring 0).

Fortunately, this makes it relatively easy to check the obvious hooking points in the OS by tracing through the code that's executed. We can then highlight any deviation from the expected chain of processing and alert the user.

This approach is effective, but it suffers from two major weaknesses. First, it detects only stealth that's implemented by intercepting OS calls at a standard point. This weakness lets clever attackers hook elsewhere, at the cost of complexity or compatibility. Second, this approach merely tells users that a hook has intercepted a call—it says nothing about why the hook has rerouted the call. At first glance, it might seem that all such interceptions are signs of malicious activity, but this is far from the truth—for example, many anti-malware solutions rely on OS-level hooks to provide real-time protection. Worse yet, this technique is really better thought of as hook detection because it really doesn't tell us anything about the hook's nature. As such, it's prone to false positives and easy to defeat.

Cross-view diff techniques

Given the limitations with the techniques previously discussed, it

Additional resources

The tools listed below use the "cross-view diff" technique to compare views of system components (processes, files, registry data, and so on) taken from different levels. When components are visible in the low-level view, but are hidden from a higher-level scan, it's most likely that the operating system (OS) has been modified to hide malicious activity. Although limited in scope, this is currently one of the best rootkit detection approaches.

- RootkitRevealer v1.71 (www.microsoft.com/technet/sysinternals/utilities/RootkitRevealer.mspx). This tool created by Bryce Cogswell and Mark Russinovich performs a detailed scan of the file system and Windows registry, looking for inconsistencies that might indicate the presence of a rootkit.
- F-Secure BlackLight (www.f-secure.com/blacklight/). This product, now integrated with an antivirus scanner, uses a different set of heuristics to scan for hidden processes and files and can detect some active rootkits.
- Strider GhostBuster (http://research.microsoft.com/rootkit/). This tool provides two modes of operation: a multilevel internal comparison similar to the approach taken by RootkitRevealer and comparison with an external scan performed by booting from a clean OS.

Unfortunately, no general solution exists for detecting rootkits. Further research is needed to determine the best approach, but it might require a more complex solution, such as a combination of scanning techniques with a hardware-based reference monitor.

would be nice if we could detect that something is actually wrong as opposed to detecting threats we already know about or producing a generic list of abnormalities in the OS. Luckily, a class of stealth detection techniques can do just this: cross-view diffs.

The basic idea of a cross-view diff is as simple as it is elegant. Essentially, most hook-based stealth systems aren't perfect; it's time-consuming and laborious to intercept every possible call and modify it in real time. Consider a piece of malware, for example, that hides a file in the root of the C drive. Although it's perfectly possible for attackers to intercept the directory listing functions and thereby prevent the system from enumerating the file, the bottom line is that the file exists on the disk. It's taking up space and using an entry in the file system.

Finding the "stealthed" file could be as simple as enumerating the files on the disk after booting from a trusted CD and comparing this view of the disk to one measured when the malware was active. This is essentially a cross-time diff—looking at the file system at two different times and highlighting any discrepancies. This approach is really a mainstay of traditional security and is the foundation of most integrity-checking solutions. However, because the comparison represents two different times, it's possible for discrepancies to have completely benign origins. For example, after taking a snapshot of the system, any files modified as the system shuts down will show up as changed when scanned later. Such changes are real—that is, they represent real changes to the system—but essentially benign in origin.

In contrast, a cross-view diff approach measures the system not at two different times, but using two (or more) different techniques. The benefit of this approach is that, at a specific point in time, we expect a system to be consistent regardless of how it's measured. So the system should return the same set of files when we manually parse the physical disk and when we enumerate files using OS-level APIs, for example. Discrepancies can represent errors in the system's design, data corruption, or the presence of stealth. Apart from system design issues—which are easy to account for—administrators will want to know about data corruption or stealth. Thus, the cross-view diff approach has the advantage of telling us things we want to know with few false positives. For a more in-depth discussion of this technique, Yi-Min Wang and his colleagues provide an excellent overview.[1]

Detecting hypervisors

Previously, we introduced the idea of hypervisor-based stealth on the Windows platform. Since then, there has been considerable discussion among researchers of how effective a hypervisor-based attack could be. Is the stealth provided by virtualization really that impenetrable?

Recently, a good overview of detection strategies for virtual machine- (VM)-based stealth has received a certain amount of coverage in the popular media. Tal Garfinkel and coauthors argue quite convincingly that although such attacks seem to hold great promise, creating a truly undetectable solution is impractical.[2] We follow their high-level arguments here.

The first challenge attackers must overcome is providing environments that are free from logical inconsistencies—for example, when developing virtualization support on a CPU, the chip manufacturer's emphasis is on speed and compatibility, not invisibility. This means that some instructions, such as `SIDT` and `SLDT`, can behave slightly differently when ex-

ecuted in a virtual environment. A program can detect these changes in behavior and report them.

Similarly, because the hypervisor effectively shares system resources with the machine it's hosting, it's possible to carry out operations that are sensitive to the competition for system resources. For example, most CPUs have a Translation Look-aside Buffer (TLB) that's responsible for mapping virtual addresses to physical addresses. By creating software to monitor TLB hits and misses, it's possible to draw conclusions about the system's state.

Finally, because the hypervisor is actually causing the machine to do more work (a hypervisor that never intercepts anything wouldn't actually be doing anything!), it affects the timing of certain operations on the host machine. Here, we should note that it usually isn't the absolute magnitude of call timings that's of interest, but their variance. Timing variance can have many sources on a virtualized environment. Virtualized hardware is difficult to accurately model, as VMs often make heavy use of caching in modeled hardware. Likewise, hypervisors must use real memory for their own code, and accessing this memory can generate page faults that betray the machine's presence. In our own work, we've found that timing signatures of even simple Windows 32 system calls vary dramatically between real and virtual machines.

Because of these concerns, Garfinkel argues that machines attempting to spoof all these detection channels will be so slow as to be obvious from users' perspectives. As such, hypervisors are eminently detectable.

Virtualization for reverse engineering?

Our arguments only address hypervisors and virtualization for rootkit creation. However, this is just one application of stealth; another is reverse engineering. In this case, performance is much less important: the actual application user isn't a good measure of system speed so we can safely assume that the reverse engineer, while not wishing the software to work prohibitively slow, is satisfied with any speed that allows for task completion.

Thus, OS virtualization might be a highly effective approach when reverse-engineering. At the lowest level, we can essentially proxy all instructions made by the program under study at the instruction level, which makes detecting that the code is operating in an ersatz environment more difficult because it's possible to

emulate functionality perfectly—especially in environments built with this approach in mind.

In such circumstances, attackers have essentially unlimited attempts to improve the environment's fidelity. Thus, leveraging an inconsistency in the way a particular instruction or piece of hardware functions becomes a game of cat and mouse, in which attackers can always modify the environment and start again.

Perhaps the most difficult thing for attackers to handle is timing variations because they would have to build good statistical models of real machines. Furthermore, programmers can leverage concurrency to their advantage, requiring attackers to perform fairly sophisticated modeling of timing in a real environment. To add to the burden, attackers must also spoof outside timing sources because the program under scrutiny could always attempt to determine time distortion by comparing internal measures with external sources.

The more checking an anti-malware solution has to do to verify its environment, the slower it will function. Thus, for practical purposes, stealth detection remains a balancing act between threats and preventative measures.

In terms of hypervisor-based rootkits, it seems that impenetrable stealth comes at far too high a price to be practical. However, the more checking an anti-malware solution has to do to verify its environment, the slower it will function. Thus, for practical purposes, stealth detection remains a balancing act between threats and preventative measures.

It's important to point out that once hostile code has loaded itself into the kernel, the defender has almost already lost. Indeed, perhaps the most powerful stealth remediation technique is to prevent the injection of stealthy code in the first place. Signed drivers and aggressive protection of kernel integrity (as seen in 64-bit Windows Vista) can go a long way toward preventing the problem. However, this applies more to rootkit and malware development, not reverse engineering.

For reverse-engineering, synthetic environments are more troublesome. Although a high-fidelity emulation is more difficult to create and is likely to exhibit poor performance, the rewards (cracking digital rights management or extracting encryption keys) can justify the cost of development, and the slowdown in execution won't be sufficient to reduce its usefulness. More work needs to be done here to fully understand the implications virtualization might have on software reverse engineering. Right now, several groups are working on the problem from the perspective of the attacker and the defender. Time will tell which side wins. ☐

References

1. Y.-M. Wang et al., "Detecting Stealth Software with Strider GhostBuster," *Proc. IEEE Int'l Conf. Dependable Systems and Networks* (DSN 05), IEEE CS Press, 2005, pp. 368–377.
2. T. Garfinkel et al., "Compatibility Is Not Transparency: VMM Detection Myths and Realities," *Proc. 11th Workshop on Hot Topics in Operating Systems* (HotOS XI), Usenix Assoc., 2007.

William H. Allen *is an assistant professor of computer science at the 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.*

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.*