Black Hat DC 2010

Conference Proceedings

An Uninvited Guest (Who Won't Go Home)

Bill Blunden
Principal Investigator
Below Gotham Labs
www.belowgotham.com

$$-K_B \sum_i P_i \log_e(P_i)$$

Abstract

While there are a multitude of battle-tested forensic tools that focus on disk storage, the discipline of memory analysis is still maturing. Even the engineers who work at the companies that sell memory-related tools have been known to admit that the percentage of incident responders who perform an in-depth examination of memory is relatively small. In light of this, staying memory resident is a viable strategy for rootkit deployment. The problem then becomes a matter of remaining inconspicuous and finding novel ways to survive a system restart. In this white paper I'll look at rootkit technology that tackles both of these challenges on the Windows platform.

Introduction

From the standpoint of a developer in the field, the process of software engineering involves balancing what's possible with what's realistic. Even a Black Hat designing a rootkit has to walk this line. Concealment takes effort, and (in the absence of a federal budget) at a certain point concessions will need to be made in the interest of expediency and practical limitations.

On the Windows platform, the functionality provided by the Windows API, the Native system calls, and the kernel's associated internal data structures are a common reservoir that both the attacker and defender drink from. Both sides can try to poison this communal well (so to speak) in hopes of disabling the other guy. The attacker can quietly modify objects to mislead an observer and fabricate results. Likewise, a defender can monitor system components, examine their composition, and set baited traps to reveal the presence of an intruder.

Hence, the more independent a rootkit is from the operating system's indigenous facilities, the stealthier it can be. In the milieu of rootkit design, autonomy is the coin of the realm.

At one end of the spectrum there's the classical approach, where an intruder basically hides in a crowd. This is what Rutkowska refers to a *Type 0 malware* [1]. This sort of software doesn't take any measures to actively conceal its presence; which is to say that it doesn't modify the host operating system in any way. It uses standard API routines to request core services from the underlying OS (e.g. file access, network I/O, IPC, etc.) and is scheduled for execution by the kernel in the same manner as a legitimate module. In other words, it runs like any other application, or driver, with the guarded expectation that it will blend in with the throng of executing code well enough to escape casual inspection by a harried system administrator.

Standard forensic analysis was made to smoke out rootkits like this [2]. All it takes is sufficient familiarity with the target platform, a baseline snapshot, and the time necessary to do a thorough job. Once all of the known-good binaries have been accounted for, Type 0 malware tends to stick out like a sore thumb.

At the other extreme, you move towards a Microkernel design where the rootkit doesn't use any of the services provided by the OS proper. It runs without assistance from the targeted system, communicating directly with the hardware, relying entirely on its own code base. In this case, nothing will be gleaned from reading the event logs and no traces of the rootkit will be unearthed by analyzing the operating system's internal bookkeeping data structures. This is because nothing in the OS itself has been modified. A hypervisor [3] or a firmware-based rootkit [4] can be viewed as an instance of this school of thought.

The problem with this latter approach is that it's extremely hardware dependent. You're essentially writing your own little OS with all the attendant driver code and processor-specific niceties.

In the absence of inside information, many attackers don't necessarily have the benefit of knowing what hardware they'll be facing once they breach the target's defenses. Sometimes all you have to start with (if you're lucky) is an OS fingerprint and a set of open ports. Most developers don't have the luxury of preparing for every contingency and they end up opting for a solution that trades stealth for portability.

This doesn't mean that malware engineers haven't taken the other route. If circumstances demand it, and the attacker has the necessary funding, they can scope out the target so that a customized one-of-a-kind rootkit can be constructed for a specific chipset, peripheral device, or system software interface. This is exactly what happened in Greece back in 2005, where intruders compromised a series of telephone switches belonging to the country's largest cellular service provider. Given the level of sophistication demonstrated by the intruders, investigators suspect that the rootkit was planted by an intelligence agency with the assistance of an insider [5].

What all of this demonstrates is that there's a cost associated with stealth. The more difficult you want to make life for the incident responders, the more resources you will spend in terms of development effort and reconnaissance. Everyone has a budget.

Black Hats on a Budget – Part I

One way to limit the number of artifacts that you leave on a system is to stay memory resident. If you take this route, you'll need to find ways to evade memory analysis and survive a system restart.

As Jesse Kornblum has pointed out, if the operating system can find a rootkit's code (to execute it), then so can the investigator [6]. This doesn't mean that rootkit detection will be an easy process, and indeed there are steps you can take to foil the incident responder.

The majority of memory forensic tools seem preoccupied with enumerating tasks and threads. To subvert this defense, you simply avoid creating the bookkeeping entries that represent tasks and threads. One field-expedient way to do this is to allocate a region of memory from the nonpaged pool and inject shellcode into it, allowing an attacker to sidestep the Windows Loader (which might otherwise be invoked to map a module into RAM, resolve addresses, etc.). Granted, the concealment we achieve is far from perfect, but this approach does offer a modicum of transferability across motherboards. Think bullet-resistant, not bullet-proof.

Our shellcode has to find some way to get the attention of the processor. Otherwise the code is just a harmless series of bytes floating adrift in kernel space. It's inevitable: somehow we have to plug in to the targeted system and institute modifications. Think of this limitation as the Achilles heel of kernel-mode injection.

In this regard, if you're going to interface with the OS, it's always better to alter system components that are inherently dynamic and thus more difficult to monitor. Thankfully the Windows kernel is rife with entropy; the average production system is a roiling sea of pointers and constantly morphing data structures. Watching this sort of system evolve is like driving down a desert highway at midnight with your headlights off. You're not entirely sure what's happening, or where you're headed, but you know you're getting there fast.

Black Hats on a Budget – Part II

If all we did was hide out in memory, our foothold on the targeted system could prove to be short-lived. Murphy's Law applies to attackers just as much as it does to everyone else. Once more, there are enterprise-class deployments (e.g. The Chicago Stock Exchange) that are restarted on a regular basis ostensibly to guard against memory leaks and gradual runtime decay [7].

From 10,000 feet, preparing for Murphy and his ilk is a matter of building fault tolerance into our rootkit. To this end, the Computrace inventory tracking product from Absolute Software serves as an illustration of how this can be done in practice. Computrace can be configured to use a persistence module embedded in the BIOS (or firmware) [8]. If the tracking service installed in the OS is removed, the persistence agent springs to life and re-installs the service.

Naturally, Absolute has the benefit of collaborating directly with hardware OEMs. The average Black Hat does not have this advantage, much less foreknowledge of the targeted system's chipset. Still, the idea of instituting a monitoring component is something that we can borrow from. Instead of deploying a single rootkit, deploy a primary rootkit and a secondary rootkit.

For example, we could install the primary rootkit on the targeted system and the secondary rootkit on another machine somewhere in the vicinity. By keeping the two rootkits on separate machines we limit our potential exposure to a catastrophic system failure that would take both components out. The secondary rootkit could check for the presence of a heartbeat, which the primary rootkit emits periodically. If the secondary rootkit fails to detect a heartbeat after a certain amount of time, it could mimic the behavior of the Computrace persistence module and re-install the primary rootkit.

Black Hats on a Budget – Part III

Then there's the matter of emitting a heartbeat signal. Network communication has traditionally been a challenge for rootkit architects. If you're too brazen, and use the existing network stack to initiate communication over a nonstandard port, you risk being blocked by the resident firewall, or (even worse) detected by the system administrator.

One work-intensive alternative is to build a self-contained networking stack into the rootkit. Not only does this allow an intruder to bypass restrictions imposed by the local firewall, it also hides the intruder's network connections from an admin who's logged onto the machine's console. On the surface, this would seem to be an ideal solution.

Nevertheless, this over-engineered approach poses serious complications. An investigator who's monitoring network traffic both locally and from a line tap may notice the discrepancy. Specifically, they'll see packets running over the wire, to and from the target system, that don't correspond to connections that are visible from the system's console. If this isn't a tip-off, I don't know what is.

A more subtle tactic would be to tunnel the heartbeat over a common protocol (e.g. ARP, HTTP, DNS, etc.) so that the connection is visible on the host but appears to correspond to legitimate traffic [9]. This is another variation of the "hide in a crowd" strategy. Many protocols have slack space, or general-purpose fields, that can be employed to ferry data. In other words, it's the network-based incarnation of the grugq's FISTing technique [10].

The problem with all of this is that you're still generating new packets, ones that don't really belong, and these new packets in and of themselves may be enough to give you away. For example, an elderly mainframe that's running COBOL apps written in the 1980s to execute financial transactions deep in a LAN probably wouldn't have any reason to generate HTTP traffic. A security officer perusing NSM logs would probably choke on their coffee and raise the alarm if they saw something like that.

This is the beauty of Passive Covert Channels (PCC). Rather than emit new packets, why not make subtle modifications to existing packets to transmit information. In other words, it's steganography at the packet level. There has been some publicly available work done in this domain both inside [11] and outside of academia [12].

You could argue that all of this fuss really isn't necessary. Do you really have to crack a heavily guarded mainframe to access the data that it stores? Rather, would it be simpler just to compromise a client machine that has access to that data? Imagine, for a moment, the desktop machine of a high-ranking executive officer who has all sorts of little toy applications and browser extensions installed on their system. This sort of noisy environment makes it much easier to tunnel out data.

Sample Code and Build Environment

The sample code that accompanies this presentation consists of three packages, each placed in a separate folder:

- HeartBeat
- Bin2Array
- KMDLoader

The HeartBeat directory houses a shellcode payload that generates a heartbeat signal tunneled over DNS. The Bin2Array package is just a primitive tool that takes this shellcode and translates it into an array in the C programming language.

The KMDLoader directory contains a user-mode component that passes the shellcode C array to a kernel-mode staging driver, which in turn injects the shellcode into memory and modifies the OS so that the shellcode is periodically executed at random intervals. The KMDLoader is basically the software equivalent of training wheels. In practice, the shellcode would most likely be deployed via exploit (e.g. either a user-mode exploit that loads its own staging driver or a direct attack against a buggy KMD). I've stuck with the training wheels in an effort to focus on post-intrusion topics.

Strictly speaking, the build script in the HeartBeat folder generate a vanilla kernel-mode driver. The shellcode that we're after is embedded in this KMD.

Standard toolsets like Visual Studio and the WDK weren't really designed to produce shellcode. They were designed to emit libraries, executables, and drivers that adhere to the Windows PE file format. Thus, coaxing them into emitting position independent code takes a bit of tweaking.

For example, one step that I took was to merge all of the relevant code and data into a single executable section (i.e. the `.code` section). By default, the compiler will emit warnings about this that are treated like errors, putting the kibosh on our shellcode dreams. To disable this behavior I changed the `LINKER_WX_SWITCH` macro in the WDK's `makefile.new` file from `/WX` to `/WX:NO`.

The hardest part was figuring out the correct combination of compiler options. Based on my experience, building shellcode is a matter of ensuring that the compiler doesn't mix in all of those extra value-added features (like frame pointers, buffer checks, type checks, optimization, etc.). As Shel Silverstein observed, "some kind of help is the kind of help we all can do without." So, it's not what you introduce into the final byte stream, but actually what you keep out. After several hours of trial and error, I ended up using the following set of parameters:

```
USER_C_FLAGS=/Od /Oy /GS- /J /GR- /FAcs /TC
```

The end result is a driver module (a `.sys` file) which has shellcode snookered away in the `.code` section. To get at this shellcode you can use the ever-handy `dumpbin.exe` utility to determine the physical offset of the `.code` section within the driver. I opted for a low-tech solution and used a hex editor to extract the shellcode once I located it. I suppose it wouldn't be too hard to write a tool that would automate the process. An even more elaborate solution would be to write a full-fledged compiler that spits out kernel-mode shellcode as its final product. Then you could build an IDE with an integrated debugger, a profiler, a virtual machine testing ground, and … and …

Last but not least, in the staging driver I reference a nonstandard API so that I can feed the shellcode the necessary fix-up address at runtime. To link this API into the staging driver I had to append the path to the `aux_klib.lib` library to the end of the `GETLIB` macro in the WDK's `makefile.new` file.

State-Sponsored Rootkits

The recurring theme of this white paper has been that you can't have your cake and eat it too. But this isn't always the case. With enough money and the proper resources (read: staffing, equipment, time), you can build a rootkit that, at least over the short term, can attain near perfect levels of stealth. The organizations that can build this sort of rootkit are the same ones capable of building a MIRVed SLBM. I'm talking about the heavy hitters; the groups funded by a national budget [13].

These high-end purveyors have advantages not afforded to the independent labs and lone Black Hat developers. They have close ties with governmental departments that can leverage their clout to encourage vendors to cooperate. Why spend the better part of a year reverse-engineering a proprietary chipset, with somewhat limited success, when you can simply read the architect's original design specification and be done with it? All of the energy that would otherwise be spent deciphering magic numbers and performing differential analysis can be funneled directly into more productive software development, saving who knows how much hair pulling and resulting in a more stable, powerful, rootkit.

In the race between the Black Hats and the White Hats, victory often goes to whoever burrows deeper into the core regions of a system (recall what I said about autonomy). In the early days of the Intel platform, this meant descending into Ring 0. As related technology has matured, rootkits that execute in Ring -1 (hypervisor host mode), Ring -2 (SMM Mode), and Ring -3 (the Intel AMT Environment) have appeared. As the path of program control submerges into the lower rings, it becomes more entangled with the intricacies of the native chipset and much harder to detect.

Thus, it should come as no surprise that major league players have gone all the way down into the hardware, placing backdoors at the circuit level [14]. Scan with Anti-Virus software all you want, it will do little to protect against this sort of embedded subversion.

At the end of the day, the independent labs (the ones that make their code available to the general public) are probably several steps behind the cutting edge. In fact, I'm pretty sure our tech is neither Black Hat nor White Hat; it's old hat. In all honesty, if you were an attacker who was actively engaged in collecting intelligence (and perhaps breaking laws in other countries), would you publish the blueprints for your offensive weaponry? Thus, take what I provided in this white paper and extrapolate it out a bit and you may at least have an idea of what the White Hats are up against. It's enough to make you want to unhook from the network and bury your servers in a thick slab of concrete.

Closing Thoughts

In a sense, rootkits are nothing new. They're merely the hi-tech embodiment of techniques that have been practiced for ages. The best way to maintain control over a system is to burrow deep into the infrastructure, where core components can be manipulated both to manage the flow of information to the outside and to orchestrate events that increase the interloper's relative level of privilege. All it takes is an intimate understanding of the existing system and the right kind of access [15].

The rootkits are there, as are the informal back channels that they use to control our institutions. As any skilled forensic investigator will tell you, recognizing what's *really* going on is simply a matter of finding new ways to discern their presence and trace their movement. Mind control [16], subterfuge [17], and Hegelian dialectics [18] are more prevalent than you may think. Pay no attention to the man behind the curtain, says the great ball of fire named Oz. Regrettably, the bulk of society dutifully heeds this advice, perpetuating the silent reign of the Wizard.

References

[1] Joanna Rutkowska, *Introducing Stealth Malware Taxonomy*, November 2006, http://www.invisiblethings.org/papers/malware-taxonomy.pdf

[2] Harlan Carvey, *Windows Forensic Analysis DVD Toolkit*, 2nd Edition, Syngress, June 2009, ISBN-10: 1597494224

[3] http://bluepillproject.org/

[4] Alexander Tereshkin, Rafal Wojtczuk, *Introducing Ring -3 Rootkits*, Black Hat USA 2009, http://www.blackhat.com/presentations/bh-usa-09/TERESHKIN/BHUSA09-Tereshkin-Ring3Rootkit-SLIDES.pdf

[5] Vassilis Prevelakis, Diomidis Spinellis, "The Athens Affair," *IEEE Spectrum*, July 2007, http://spectrum.ieee.org/telecom/security/the-athens-affair

[6] Jesse Kornblum, "Exploiting the Rootkit Paradox with Windows Memory Analysis," *International Journal of Digital Evidence*, Fall 2006, Volume 5, issue 1

[7] Microsoft Corporation, *Windows NT Server at The Chicago Stock Exchange: Technical Roadmap*, http://staging.glg.com/tourwindowsntserver/CHX/technical4.htm

[8] http://www.absolute.com/resources/public/FAQ/CT-FAQ-TEC-E.pdf

[9] Alhambra and daemon9, "Project Loki: ICMP Tunneling," *Phrack Magazine*, Volume Seven, Issue 49

[10] grugq, *The Art of Defiling*, Black Hat Asia 2003, http://www.blackhat.com/presentations/bh-asia-03/bh-asia-03-grugq/bh-asia-03-grugq.pdf

[11] Steven J. Murdoch and Stephen Lewis, *Embedding Covert Channels into TCP/IP*, Information Hiding Workshop 2005 proceedings, http://www.cl.cam.ac.uk/~sjm217/papers/ih05coverttcp.pdf

[12] Joanna Rutkowska, *The Implementation of Passive Covert Channels in the Linux Kernel*, Chaos Communication Congress, December 2004, http://www.invisiblethings.org/papers/passive-covert-channels-linux.pdf

[13] Christopher Drew and John Markoff, "Contractors Vie for Plum Work, Hacking for U.S." *New York Times*, May 30,2009, http://www.nytimes.com/2009/05/31/us/31cyber.html?_r=2

[14] John Markoff, “Old Trick Threatens the Newest Weapons,” *The New York Times*, October 26, 2009, http://www.nytimes.com/2009/10/27/science/27trojan.html?pagewanted=1&_r=1

[15] William Greider, *Who Will Tell The People? : The Betrayal of American Democracy*, Simon & Schuster 1993, ISBN-10: 0671867407

[16] Edward S. Herman and Noam Chomsky, *Manufacturing Consent: The Political Economy of the Mass Media*, Pantheon 2002, ISBN-10: 0375714499

[17] Bethany McLean and Peter Elkind, *The Smartest Guys in the Room: The Amazing Rise and Scandalous Fall of Enron*, Portfolio Trade, 2004, ISBN-10: 1591840538

[18] Alan Greenspan, *The Age of Turbulence: Adventures in a New World*, Penguin, 2008, ISBN-10: 0143114166, “I am saddened that it is politically inconvenient to acknowledge what everyone knows: The Iraq war is largely about oil.”