Black Hat USA 2009
Conference Proceedings

Anti-Forensics: The Rootkit Connection

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

Abstract

Conventional rootkits tend to focus on defeating forensic live incident response and network monitoring using a variety of concealment strategies (e.g. hooking, detour patching, covert channels, peer-to-peer communication, etc.). However, the technology required to survive a post-mortem analysis of secondary storage, which is just as vital in the grand scheme of things, recently doesn't seem to have garnered the same degree of coverage. In this paper, we'll examine different approaches to persisting a rootkit and the associated anti-forensic tactics that can be employed to thwart an investigator who's performing an autopsy of a disk image.

Contents

Introduction

"If the campaign is protracted, the resources of the state
Will not be equal to the strain"
—Sun Tzu

Though live incident response has its place, particularly on mission-critical systems, it suffers from a notable flaw: a rootkit being pursued at runtime is a part of the machine's execution state and is in a unique position where it can interfere with data collection.

For example, back in March of 2005, forensic investigators from the Swedish Telecom Manufacturer Ericsson identified a rootkit that had been installed on a series of switches in central offices run by Vodafone Greece, the largest cell phone service provider in the country. Engineers at Vodafone inadvertently stumbled across the rogue software after its auto-update mechanism prevented the delivery of a text message (which generated an audit trail in the system's error logs). This memory-resident rootkit survived for as long as it did because it patched the switch's runtime image to conceal its presence and also modified the system's native integrity checking facilities to make it appear as though nothing had been altered [1].

The forensic software vendors who sell live response tools usually downplay the threat of rootkit interference. For example, Technology Pathways sells a forensic toolkit called ProDiscover IR. According to the online help for this product [2]:

"Some administrators will suppose that if a rootkit could hook (replace) File I/O request they could simply hook the sector level read commands and foil the approach that applications such as ProDiscover® IR use. While this is true on the most basic level, hooking kernel sector read commands would have a trickle-down effect on all other kernel level file system operations and require a large amount of real-to-Trojaned sector mapping and/or specific sector placement for the rootkit and supporting files. This undertaking would not be a trivial task even for the most accomplished kernel mode rootkit author."

As it turns out, the very same countermeasure that Technology Pathways brushed off as unlikely was presented by Darren Bilby at Black Hat Japan 2006 in a proof-of-concept rootkit called DDefy [3]. This rootkit employed a filter driver that intercepted `IRP_MJ_READ` I/O requests on their way to the official disk

driver so that requests for certain sectors could be altered to yield sanitized information. In this manner, a valid drive snapshot is generated that excludes the rootkit.

In an effort to deal with rootkit interference, some security tools vendors tried to sidestep the OS entirely and go right to the hardware to collect data with the guarded expectation that they'd be able to preclude a rootkit's involvement. For instance, a security tool vendor named Komoku (which was acquired by Microsoft in March of 2008 [4]) at one point offer a hardware-based RAM capturing tool named CoPilot. This tool used a PCI card to allow the investigator to extract a snapshot of a running computer's volatile memory.

Enter Joanna Rutkowska, who demonstrated at Black Hat Federal 2007 that it was possible to defeat this sort of hardware-based approach by fiddling with the map table of the motherboard's Northbridge [5]. In other words, it's entirely feasible for rootkit to foil hardware-based forensic tool by manipulating related motherboard components that the forensic tool relies upon to read memory. After all, peripheral devices don't exist in a vacuum; they depend upon other hardware elements that can be subverted by a knowledgeable attacker.

There's a lesson in this race to the metal. If a rootkit has embedded itself on your server: *even if you go straight to the hardware you can't necessarily trust what the server is telling you at runtime because the rootkit may somehow be altering the information that's being acquired*.

Post-Mortem Dance Steps

Given the quandary of live incident response, one option that an investigator can exercise during an investigation is to power down the server in question so that they can examine the machine's drives from a trusted forensic workstation. This is assuming that it's financially and legally tenable to do so.

Once a machine has been powered down, the classic post-mortem tends to follow a step-wise sequence of actions (see Figure 1). The investigator will begin by creating a duplicate image of secondary storage and then recover files from this replica with the immediate goal of collecting as large a data set as possible. To this end they may attempt to carve out deleted files, file fragments, alternate data streams, slack space, and other assorted "hidden" items.

Next, they'll take the resulting set of file system objects and generate a meta-data snap-shot (e.g. timestamps, checksums, etc.) which they'll use to help them weed out "known" file system objects.

Once they've identified known "good" and "bad" objects, what they're left with is a subset of unknown files. These unknown files will then take center stage for the remainder of the investigation. The analyst will search for binary signatures in an effort to locate executables and then dissect what they find using the tools of static and runtime executable analysis.

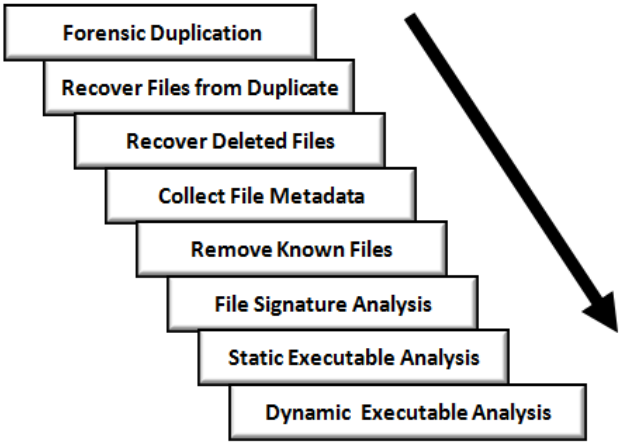

Figure 1

Disk Analysis

In this paper, we'll look at each phase of secondary storage analysis in turn and discuss ways in which an intruder can undermine the process and permit their rootkit to remain undetected. Ultimately, no countermeasure is perfect; there are tradeoffs and downsides associated with every technique. The basic idea is to leave as little evidence as possible, and to make the evidence that you do leave behind so expensive to analyze that the investigator is more likely to give up or perhaps be drawn to an incorrect, but more convenient, conclusion.

Worst-Case Scenario

Ultimately, the degree to which you'll need to employ anti-forensic measures is a function of the environment that you're targeting. In the best-case scenario, you'll confront a bunch of overworked, apathetic, system administrators who could care less what happens just as long as their servers are running and no one is complaining ("availability all at costs!"). In this instance, you could probably

get away with hiding in plain sight: camouflaging your rootkit so that it looks legitimate to a cursory inspection. As long as you don't do anything reckless, you'll probably coast quietly under the radar.

In the worst-case scenario, you'll be going toe-to-toe against a skilled investigator who has the resources, motivation, skill set, and authority to do what they need to in order to track you down. You know the sort of person. They're passionate, results-oriented, and constantly honing their skills. For grins they buy used hard drives off of eBay just to see what they can recover.

NOTE

For the sake of keeping this discussion interesting, we're going to assume the worst-case scenario.

In a truly locked-down environment, it's not unheard of for security professionals to assume that one of their systems has been compromised a priori, and *preemptively* employ forensic techniques to flush out an intruder. In other words, even if a particular system appears to be perfectly healthy they'll utilize a rigorous battery of auditing and verification procedures to confirm that it's trustworthy.

Strategies for the Worst-Case

There's no such thing as a burglar-proof safe. With the right tools and know-how, it's usually just a matter of time before you crack it open. This is why the Underwriters Laboratories rates safes in terms of their resistance to a specific set of tools for a certain period of time. For example, a safe that's designated with a rating of TRTL-30x6 can withstand hand tools (TL) and torch (TR) attack on all six sides for 30 minutes.

Likewise, there's no such thing as a foolproof anti-forensic tactic. With the right tools and know-how, it's just a matter of time before a savvy forensic investigator will overcome the fortifications that you've established. Given that an impenetrable defense is fairly unrealistic, the goal then is to *buy as much time as you can.*

This central tenet is based on the reality that every investigator has a resource budget. Even well-funded, dedicated, forensic analysts who are willing to work in their off hours have a finite amount of time they can devote to any particular

case. Can you imagine someone with a backlog of machines and a supervisor who's breathing down their neck?

Naturally, a forensic analyst will leverage automation to ease their workload and speed things up. There's an entire segment of the software industry that caters to this need. Even then, there's always that crucial threshold where relying on the output of an expensive point-and-click tool simply isn't enough. In some cases, an appropriate tool won't exist and will have to be custom built, requiring yet more effort.

As an attacker, your goal is to make the process of forensic analysis so grueling that the investigator is more likely to give up or perhaps be lured into prematurely reaching a false conclusion (one that you've carefully staged for just this very reason) because it represents a less painful, though logically viable, alternative. Put another way: why spend twenty years agonizing over a murder case when you can just as easily rule it out as a suicide? This explains why certain intelligence agencies prefer to eliminate enemies of the state by means of an "unfortunate accident."

To buy time and lead the investigator astray, you can employ five general strategies. Every anti-forensic tactic that a rootkit uses to evade a post-mortem disk examination can be seen as a manifestation of one or more of these strategies (see Table-1). This framework is an amalgam of categories originally presented by the grugq [6] and Marc Rogers [7].

Strategy	Tactical Implementations
Data Destruction	File scrubbing, Meta-data wiping, File System Attacks
Data Concealment	In-Band, Out-of-Band, and Application Layer Concealment
Data Transformation	Compression, Encryption, Code Morphing, Direct Edits
Data Fabrication	Introduce known files, String decoration, False audit trails
Data Source Elimination	Data Contraception, In-Memory DLL Injection

Table 1

Core Strategies

To reiterate our objective in terms of these five concepts:

If your rootkit persists on disk then you want to buy time by leaving as little useful evidence as possible (data destruction).

The evidence that you do leave behind should be difficult to capture (data concealment) and even more difficult to understand (data transformation).

Furthermore, you can augment the effectiveness of this approach by planting counterfeit leads and luring the investigator into following any number of false trails (data fabrication).

Though, to be honest, probably the best way to foil the investigator is not to leave any evidence to begin with (data source elimination).

NOTE

Some anti-forensic treatments consider attacking the investigator's forensic tools directly (e.g. leveraging a bug, or logical flaw, in the tool to run an exploit on the investigator's machine or implement a denial-of-service attack). I'm going to shy away from this sort of instance-specific strategy in favor of focusing on more transferable technology that's not as conspicuous.

Disk Analysis: Tactics and Countermeasures

"If you know the enemy and know yourself, you need not
Fear the result of a hundred battles."
—Sun Tzu

The basics of forensic disk analysis are well-known. Entire books have been devoted to the topic [8]. We can pretty much predict what a skilled investigator will do, why they'll do it, and this gives us a certain advantage. Let's see what weapons we have at our disposal to make the investigator's life miserable.

Defense in Depth

As I mentioned earlier, no counter-tactic is perfect. Thus, you should adopt a defense in-depth approach that takes several tactics and institutes them concurrently so that the minute that the investigator leaps one hurdle they fly head-first into the next one. Play your cards right and the forensic investigator just might throw their arms up and resign themselves to the most obvious conclusion.

Forensic Duplication

In an effort to preserve the scene of the crime, so to speak, the investigator will create a duplicate of the original hard drive. This first-generation copy can then be used to spawn a number of second-generation copies, so that the original can be bagged and tagged.

Reserved Disk Regions: One way to subvert duplication is to place your rootkit so far off the beaten path that it might be overlooked by the imaging tool. For example, the Host Protected Area (HPA) is a reserved disk region that's often used to store diagnostic tools and system restore utilities. Then there's also the Device Configuration Overlay (DCO), which facilitates RAID by allowing disparate hard drives from different OEMs to behave as though they all have the same number of sectors (often leaving leftover space in the process) [9].

Unfortunately, any contemporary tool worth its salt can detect and scan HPAs and DCOs. For instance, the EnCase tool suite sold by Guidance Software employs a drive docking platform called FastBloc 3 FE that can retrieve data hidden in HPAs and DCOs [10].

Recovering File System Objects

Having created a duplicate of disk storage, the investigator will then carve it up with their forensic tools to yield a set of file system objects (e.g. files, slack space, alternate data streams, file fragments, etc.). To protect our rootkit binary we can implement the following countermeasures:

- Full Disk Encryption
- File System Attacks
- File concealment

Full Disk Encryption: Though this sort of functionality is intended to safeguard data from an offline attack, it can also be enabled to hinder file carving. Privacy is a sword that cuts both ways. If a drive or logical volume has been encrypted, extracting files will be an exercise in futility (particularly if the decryption key cannot be recovered). As a matter of personal taste, I dislike this technique because it's too conspicuous.

NOTE

Some encryption packages use a virtual drive (i.e. a binary file) as their target. In this case, one novel solution is to place a second encrypted virtual drive inside of a larger virtual drive and then name the second virtual drive so that it looks like a temporary junk file. An investigator who somehow manages to decrypt and access the larger encrypted volume may conclude that they've cracked the case and stop without looking for a second encrypted volume. One way to encourage this sort of premature conclusion is to sprinkle a couple of well-known tools around along with the encrypted second volume's image file (which can be named to look like a temporary junk file).

File System Attacks: Some researchers advocate widespread destructive modification of the file system in order to prevent forensic tools from identifying files properly. In my eyes this approach, taken to this extreme, lies strictly in the "scorched earth" side of the spectrum. By maiming the file system you're inviting the sort of instability and eye-catching behavior that rootkits are supposed to avoid to begin with. Not to mention that contemporary file carving tools don't necessarily need the file system to begin with.

NOTE

Rootkits, like the one discovered at Vodafone Greece, have been discovered accidently, as result of the unintended side-effects that they introduce, rather than overtly triggering an alarm set by security software.

File Concealment: If we stow our files in unexpected places, the forensic tools being employed may not recognize them as files. In terms of hiding data on disk, within the confines of a given file system, there are three strategies that were enumerated by Irby Thompson and Matthew Monroe at Black Hat DC 2006 [11]:

- Out-of-Band Concealment
- In-Band Concealment
- Application layer Concealment

Out-of-Band Concealment: This strategy aims to store data in spots that are not explicitly described by the file system specification. Both the HPA and DCO are examples of out-of-band regions. *Slack space* is yet another well known example (a topic that's near and dear to the heart of any SubGenius).

"Eternal salvation, or triple your money back!"
— J. R. "Bob" Dobbs

On Windows XP, a user-mode application could stick data in a file's slack space using the following sort of code.

```
#define SZ_BUFFER  2000

HANDLE fileHandle;
char buffer[SZ_BUFFER];
DWORD nBytesWritten;

//Step [1] - set the FP to the logical EOF
SetFilePointer
(
      fileHandle,  //HANDLE hFile
      0,           //LONG lDistanceToMove
      NULL,        //PLONG lpDistanceToMoveHigh
      FILE_END     //DWORD dwMoveMethod
);

//Step [2] - write data between the logical EOF and physical EOF
WriteFile
(
      fileHandle,         //HANDLE hFile
      buffer,             //LPCVOID lpBuffer
      SZ_BUFFER,          //DWORD nNumberOfBytesToWrite
      &nBytesWritten,     //LPDWORD lpNumberOfBytesWritten
      NULL                //LPOVERLAPPED lpOverlapped
);

FlushFileBuffers(fileHandle);

//Step [3] - move FP back to the old logical EOF
SetFilePointer
(
      fileHandle,  //HANDLE hFile
      -SZ_BUFFER,  //LONG lDistanceToMove
```

```
      NULL,        //PLONG lpDistanceToMoveHigh
      FILE_CURRENT //DWORD dwMoveMethod
);

//Step [4] - truncate the file nondestructively (on XP)
SetEndOfFile(fileHandle);
```

The primary challenge (or perhaps tedium) of this method is that you'll need to keep track of which file's slack space you used and how much of your data you put there. These associated bits of metadata will need to be archived somewhere (in a secondary file, or perhaps in predetermined disk regions). If this metadata is corrupted, you'll lose your slack data.

The Metasploit Anti-Forensics Project (MAFIA) developed the first publicly available tool for persisting data in slack space on Windows. This tool, appropriately named `slacker.exe`, uses an external index file to track slack space usage and works beautifully on Windows XP. The folks from the MAFIA presented this tool back at Black Hat USA 2005 [12].

Microsoft has since plugged this user-mode hole in Vista, by re-wiring the `SetEndOfFile()` call to zero out slack space before returning. *But this doesn't mean we're out of the game*. Like it or not, that slack space is still there (heck, it exists by design). In fact, it wouldn't be unreasonable to implement a kernel-mode driver (KMD) that performed raw disk I/O to store data in slack space.

Anyone intent on utilizing slack space using such a KMD would be advised to rely on static files that aren't subjected to frequent I/O requests, as files that grow can overwrite slack space. Given this fact, the salient issue then becomes identifying such files (hint: built-in system executables are a good place to start). Also, be warned that a truly paranoid system administrator might periodically wipe slack space as a preventative measure.

In-Band Concealment: This approach stores data in obscure spots that are described by the file system specification. Modern file systems are a byzantine cityscape of data structures, teaming with any number of alleyways and hidden tunnels. The basic idea of stowing data in the nooks and crannies of a file system has been addressed at length by the grugq on the UNIX platform [13]. He referred to his approach as the File Insertion and Subversion Technique (FIST), as in "find a hole and then fist it." The grugq is fond of colorful acronyms.

In order for FISTing to work as intended, the following prerequisites must be fulfilled:

- The probability of the hidden data being overwritten is low.
- A file system integrity check should not generate errors.
- The tactic should allocate a nontrivial amount of storage space.

On Windows, the NTFS Master File Table (MFT) is a viable candidate for FISTing. The MFT is the central repository for metadata on an NTFS file system. It's basically a database consisting of a series of variable-length records, where each physical file in the file system is mapped to one or more records in the database.

If you want to dig into details of the format of the records in the MFT, the official Microsoft documentation on the NTFS file system is a good starting point [14], but ultimately it doesn't go beyond an introductory level of detail. To get the entire scoop on NTFS, you'll need to read a book like Brian Carrier's [15] and then cross the tracks to the Linux-NTFS project [16]. There's also a discussion of the MFT in my book on rootkits [17]. I find it ironic that the open source community ends up being the best source of information on a technology put out by the people in Redmond. By keeping this information proprietary they're making life more difficult for forensic investigators.

The Linux-NTFS project shows what can be done through exhaustive Reverse Code Engineering (RCE). The driver and its associated documentation represent a truly impressive accomplishment. Because NTFS is a proprietary file system, the engineers involved in this project had to reverse engineer NTFS from scratch. According to the project's web site, the methodology was roughly:

1. Look at the volume with a hex editor
2. Perform some operation, e.g. create a file
3. Use the hex editor to look for changes
4. Classify and document the changes
5. Repeat steps 1-4 forever

An implementation of FISTing with NTFS was presented at Black Hat DC in 2006 [11]. The project's tool, FragFS, used entries in the MFT that corresponded to relatively static, nonresident, system files. The hodgepodge of free space was used to build a storage region was logically formatted into 16-byte units. While

`slacker.exe` uses an external index file to track metadata, FragFS uses the last eight bytes of each MFT record to maintain storage-related metadata.

The 16-byte storage units allocated by FragFS are managed by a KMD in such a way that user-mode applications that employ the driver see a virtual disk drive. All of the details of storage management are hidden from view. The FragFS KMD presents a block-based storage device to the rest of the system. You can treat this disk like you would any other (e.g. create folder hierarchies, move files around, execute applications). Essentially, what FragFS does is to create a file system within a file system. Recall the X-Files, where Agent Mulder would make ominous sounding references to a shadow government within a government.

Application Layer Concealment: The goal of this strategy is to store data within the recesses of individual files whose internal logical structure is defined by some user-mode application. Steganography is a classic instance of this train of thought that's gained notoriety since 9/11. According to a former French defense ministry, terrorists who were apprehended in 2001 were instructed to transmit communications through stenographic messages embedded on pictures posted on the Internet [18].

The trick with application layer concealment is to find a file that can accommodate modification and that's normally subjected to frequent I/O operations, so that the investigator can't use checksum-based techniques to detect your updates. In other words, find a noisy spot in the file system and hide in plain sight. Database stores are a perfect fit for these requirements.

The Windows registry is analogous to Grand Central Station in New York City. One way or another, eventually everyone passes through it. The registry has lots of foot traffic, so to speak, and its contents are altered several times per second. Hence, there's no way to reasonable checksum this sort of file. It perpetually changes by design. The best that an investigator can hope to do is to take an ASCII dump and perform a cross-time diff of two snapshots, and even then the typical level of activity in the registry is such that they'll have no idea what to look for.

The Achilles heel of storing contraband in a database file is that manipulating such a file through the official channels is trivial to log. Thus, if you can help it, you might want to side-step the official API in favor of manually accessing the database file [19]. Manual parsing isn't as draconian as it seems. Specifically,

I'd suggest using the official API to initially allocate space inside of the database file and then prefixing the storage space with a signature of some sort. Then, when you need to manipulate or read this storage you can manually scan the file in question and use the signature as a beacon.

NOTE

Concealment has always been a short-term strategy. It's security through obscurity by another name. If the investigator searches long enough, eventually they'll uncover your hiding spot. Once the details of a specific hidden region are revealed (e.g. slack space, the MFT, ADS space), it's not long before the tool vendors add the necessary functionality to their security products to scan it. Thus, if you're going to hunker down in a murky corner, it would probably be a good idea to augment your defenses with a healthy dose of data transformation. This way, if the investigator does happen to find something, it looks like a load of random junk from `/dev/random`. Remember what I said earlier about wielding a defense in-depth?

Recovering Deleted Files

The preliminary goal of a post-mortem is to identify as many file system objects as possible. The more items that the investigator recovers the greater the chance is that they may come across a potential lead. To this end, they will attempt to carve out deleted files (i.e. files that no longer exist according to a naïve interpretation of the bookkeeping data structures, but whose bytes still reside somewhere on disk).

Again, tactics intended to ensure privacy can be commandeered as tools to destroy evidence. Specifically, I'm talking about:

- File Wiping
- Meta-Data Shredding
- Encryption

File Wiping: The Defense Security Service (DSS), an agency under the Department of Defense, provides a clearing and sanitizing matrix (C&SM) that specifies how to securely delete data. The DSS distinguishes between different types of sanitizing (e.g. clearing versus purging). "Clearing" data means that it can't be recaptured using standard system tools. "Purging" data kicks things up a notch. Purging "protects the confidentiality of information against a laboratory attack" (e.g. magnetic force microscopy) [20].

According to the DSS C&SM, "most of today's media can be effectively cleared by one overwrite." Current research supports this [21]. Purging can be performed via the Secure Erase utility, which is uses functionality that's built into ATA drives at the hardware level to destroy data. Secure Erase and can be downloaded from the University of California, San Diego (UCSD) Center for Magnetic Recording Research (CMRR) [22].

NOTE

Software-based tools that implement clearing often do so by overwriting data in place. For file systems designed to "journal data" (i.e., persist changes to a special circular log before committing them permanently), RAID-based systems, and compressed file systems, overwriting in place may not function reliably with regard to specific files.

Meta-Data Shredding: In addition to wiping the bytes that constitute a file, the metadata associated with a wiped file in the file system should also be destroyed. For example, in his seminal 2002 article in Phrack, the grugq presented a software suite known as the Defiler's Toolkit (TDT) to deal with such problems on the UNIX side of the fence. This toolkit included a couple of utilities called Necrofile and Klismafile to sanitize deleted inodes and directory entries [6].

Encryption: One way to avoid having to deal with the minutiae of securely wiping data from secondary storage is simply to ensure that every byte of data that you write to disk is encrypted. In my mind, this is a far simpler and more elegant solution than wiping after the fact. Given strong encryption keys, algorithms like triple-DES provide adequate protection. Once the keys have been destroyed, the data is nothing more than useless random junk.

Key Management: This highlights the truly salient aspect of this technique, which is preventing key recovery. The topic of securely managing encryption keys, from a developer's vantage point, has been discussed in a number of books [24]. White Hat technology to the rescue!

As long as you don't store an encryption key on disk, the most that a forensic investigator can do is to hunt for it in memory at runtime or perhaps wade through the system's page file post-mortem with the guarded expectation that they may get lucky.

To safeguard your encrypted data, you need to minimize the amount of time that a plaintext key exists in memory. In general, you should use it as quickly possible, encrypt it, and then zero out the memory that was storing the password. To protect against cagey investigators who might try and read the page file, there are API calls like `VirtualLock()` (on Windows machines [25]) and `mlock()` (on UNIX boxes [26]) that can be used to lock a specified region of memory so that it's not paged to disk.

Collecting File Metadata

Once the investigator has achieved a sufficient stockpile of file system objects, they'll collect metadata on each item to construct a snapshot of their data set. This can include information like:

- The file's name
- The full path to the file
- The file's size (in bytes)
- MAC times (Modified, Accessed, and Created)
- The cryptographic checksum of the file

The two sets of parameters that interest us in particular are the MAC times and checksums. By altering these values we can hinder the investigator's ability to create and plausible timeline and also force them to waste time chasing down false leads.

Altering Checksums: Though checksums expedite the process of file comparison, they're hardly robust. All it takes is a slight modification to generate a completely different hash value. A completely normal executable can be made to look suspicious by flipping a few bits.

Though patching an executable can be risky, most of them have embedded string tables that can be altered without much fanfare. If you really want to go full bore, you could implement a more substantial patch by inserting a nontrivial snippet of functioning code into unused space between the sections of the executable [27]. When the investigator performs a binary diff between the original file and the modified version, they'll see the injected code and alarms will sound. Hopefully this will lead the investigator away from the actual rootkit.

Modifying Timestamps: Changing MAC values on Windows requires the `ZwOpenFile()` and `ZwSetInformationFile()` system calls. Note that both of these routines can only be invoked at an IRQL equal to `PASSIVE_LEVEL`. The `FILE_BASIC_INFORMATION` argument fed to to `ZwSetInformationFile()` stores four different 64-bit `LARGE_INTEGER` values that represent the number of 100-nanosecond intervals since the beginning of the year 1601. When these values are small, the Windows API doesn't translate them correctly, and displays nothing. This behavior was first publicized by Vinnie Liu of the Metasploit project.

```
PCWSTR fullPath;          //assume this has been set elsewhere
BOOLEAN modTimeStamp;     //assume this has been set elsewhere

OBJECT_ATTRIBUTES         objAttr;
UNICODE_STRING            fileName;
HANDLE                    handle;
NTSTATUS                  ntstatus;
IO_STATUS_BLOCK           ioStatusBlock;
FILE_BASIC_INFORMATION    fileBasicInfo;
RtlInitUnicodeString(&fileName,fullPath);

InitializeObjectAttributes
(
      &objAttr,                        //OUT POBJECT_ATTRIBUTES
      &fileName,                       //IN PUNICODE_STRING
      OBJ_CASE_INSENSITIVE|OBJ_KERNEL_HANDLE,
      NULL,                            //IN HANDLE RootDirectory
      NULL                             //IN PSECURITY_DESCRIPTOR
);
if(KeGetCurrentIrql()!=PASSIVE_LEVEL){ return; }

ntstatus = ZwOpenFile
(
      &handle,                         //OUT PHANDLE
      FILE_WRITE_ATTRIBUTES,           //IN ACCESS_MASK
      &objAttr,                        //IN POBJECT_ATTRIBUTES
      &ioStatusBlock,                  //OUT PIO_STATUS_BLOCK
      0,                               //IN ULONG ShareAccess
      FILE_SYNCHRONOUS_IO_NONALERT     //IN ULONG CreateOptions
);
if(ntstatus!=STATUS_SUCCESS)
{
      DbgPrint("%s","unable to open file");
```

```
}

if(modTimeStamp)
{
      fileBasicInfo.CreationTime.LowPart     =1;
      fileBasicInfo.CreationTime.HighPart    =0;
      fileBasicInfo.LastAccessTime.LowPart   =1;
      fileBasicInfo.LastAccessTime.HighPart  =0;
      fileBasicInfo.LastWriteTime.LowPart    =1;
      fileBasicInfo.LastWriteTime.HighPart   =0;
      fileBasicInfo.ChangeTime.LowPart       =1;
      fileBasicInfo.ChangeTime.HighPart      =0;
      fileBasicInfo.FileAttributes = FILE_ATTRIBUTE_NORMAL;
}
else
{
      fileBasicInfo = getSystemFileTimeStamp();
}
ntstatus = ZwSetInformationFile
(
      handle,            //IN HANDLE FileHandle
      &ioStatusBlock,    //OUT PIO_STATUS_BLOCK IoStatusBlock
      &fileBasicInfo,    //IN PVOID FileInformation
      sizeof(fileBasicInfo),    //IN ULONG Length
      FileBasicInformation      //IN FILE_INFORMATION_CLASS
);
if(ntstatus!=STATUS_SUCCESS)
{
      DbgPrint("%s","unable to mod timestamp");
}
ZwClose(handle);
```

Identifying Known Files

Having acquired a meta-data snapshot of their current data set, the investigator will use this meta-data to perform a cross-time diff against a baseline snapshot (something that, hopefully, they had the presence of mind to create sometime before the incident) in an effort to identify known good and known bad files. They'll use the cross-time diff to winnow away at their original collection of files, leaving a subset of unknown files which will be the focus of the final stages of the post-mortem disk analysis (see Figure 2).

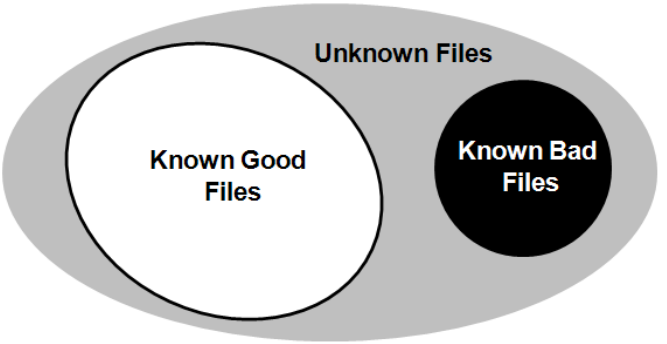

Figure 2

File Categories

Recall that we're interested in the worst-case scenario, where we're facing a heavily-armed alpha geek. If an investigator has the time and computing horsepower, they may compare files *at the binary level* [28]. This will preclude us from getting cute with checksums. Specifically, I'm talking about a *preimage attack*: altering a binary and then adding on whatever bytes are needed in order for the resulting file's checksum to match the original checksum.

Having identified known good files via the cross-time diff, the investigator may sweep their collection of files with an anti-virus package to identify known bad files. Note, this will only detect malware that's prolific enough that it ends up in the vendor's signature database and corresponding black list.

Injecting Known Bad Files: In his autobiography, Vinnie Teresa recounts a failed burglary where a gang of thieves missed out on a lucrative heist because they were misled by a small stash of valuables buried a couple of feet on top of a very large stash of valuables [29]. According to Teresa, they burglars hit the initial, smaller, stash and (thinking that they had hit the mother lode) stopped digging.

This is the basic train of thought behind introducing known malware into a system. One way to get a forensic investigator to stop is to make them believe that they've identified the source of their problems. The trick is to make the investigator work hard enough that they'll believe the decoy is genuine. In a world ruled by budgets and limited resources, it pays to lure the investigator with an easy way out.

Flooding the System: One way to buy time is to introduce a large number of legitimate (known good) files. This will cause the set of unknown files to balloon and consume resources as the investigator tracks down the origin of the introduced files. Though, I'll admit that this is another scorched earth defense. If an administrator sees a whole load of files on a server that they didn't put there, they'll probably assume that something fishy is going on. This is particularly true for servers that serve specific roles and which don't rely on third-party products (e.g. domain controllers).

File Signature Analysis

Faced with a pool of unknown files, which may or may not have the correct file name extension, the investigator will use signature matching algorithms to determine what sort of files they're dealing with. Executable files are of particular interest.

Transmogrification: One way to foil this technique is to alter the header of a file so that it doesn't match its normal signature. For example, According to the Windows PE specification, executable files begin with an MS-DOS stub whose first 16-bit word is the magic number `0x4D5A` (i.e. the ACSII string "MZ," which supposedly represents the initials of Mark Zbikowski) [30]. To foil a tool that relies on signatures like this, all you'd have to do is tweak the header. This header-based approach was adopted by Transmogrify, an anti-forensic tool developed by the MAFIA. It was the first publicly available tool to successfully foil EnCase's file signature capabilities [31].

Stenography and Encryption: We could take things a step further. Using a technique that's stenographic in nature, we could encapsulate a configuration text file by wrapping it as encoded string variables inside of a counterfeit executable. Likewise, we could Base64 encode an executable to make it look like a mish-mash of ASCII text or just encrypt it using a standard block cipher to do defeat the signature matching engine.

Static Analysis of an Unknown Executable

Confronted with an unknown executable, the first thing that an investigator will do is to examine it without actually running it. For instance, they may scan the raw binary for strings, use a tool like `dumpbin.exe` to see which routines it imports and export, and then maybe (if time permits) disassemble it to get a better idea of what it does.

Cryptors and Packers: The best way to defeat static deconstruction is to put your executable into a form that none of the tools in this phase can interpret. This is the motivation behind cryptors and packers. Both types of tools use the same basic underlying architecture, such that if you implemented a cryptor it wouldn't be very difficult to recast it as a packer and vice versa.

In a nutshell, these tools take an executable and either encrypt it or compress it. Next, these tools build a stub application and graft this onto the original (now encoded) application. When the combined executable is launched, the stub application decodes its payload and passes control to the original program's entry point. Applications like Skype use this technology as a form of software protection [32].

NOTE

The stub code is essentially acting in a manner that's reminiscent of a user-mode application loader (what's known as a *Userland Exec*, a substitute for the UNIX `execve()` system call). The stub takes an arbitrary stream of bytes, situates it in memory, and makes the necessary adjustments so that the resulting image can executed properly. This sort of functionality is useful because, to an extent, it frees us from having to depend upon the operating system's loader (which enforces certain restrictions that we'll look at shortly).

The specifics of exactly how this is implemented vary from one tool to the next. For example, the Ultimate Packer for eXecutables (UPX) takes an executable and merges its sections (`.text`, `.data`, `.idata`, etc.) into a single section named `UPX1`. The `UPX1` section also contains the stub program that will unpack the original executable at run time.

You can examine the resulting compressed executable with `dumpbin.exe` to verify that it consists of three sections:

- `UPX0`
- `UPX1`
- `.rsrc`

At runtime, the `UPX0` section loads into memory first, at a lower address. The `UPX0` section is nothing more than empty space. It doesn't take up any space at all on disk, such that both `UPX0` and `UPX1` start at the same file offset in the packed binary. The `UPX1` section is loaded into memory above `UPX0`, which

makes sense because the stub code in `UPX1` will decompress the packed executable and move the decompressed bytes down below, starting at the beginning of `UPX0`. As decompression continues, eventually the unpacked data will grow upwards in memory and overwrite data in `UPX1`. The best way to watch this happen is with a debugger.

The source code for UPX is available online if you're curious to see how it's implemented [33]. Be warned that it's not a trivial implementation (as is the case with any production grade cryptor or packer). If you'd like to take a look at the internals of a cryptor, the source code for Yoda's Cryptor is also online [34].

Without a doubt, using custom tools is the way to go. This is because there are forensic utilities available (like PEid) that will detect, and sometimes even decode, an executable that has been processed by well-known packers or cryptors [35]. By using a custom cryptor or packer you're not giving the investigator an easy out. Remember, this is all about buying time. For example, your custom tool could break up the executable file into multiple segments and use a combination of algorithms in concert. By far, the best treatment that I've read on how to create a custom tool was published back in 2006 by a researcher known as BigBoote in *CodeBreakers Magazine* [36].

Embedded Virtual Machines: Another way to stifle static analysis is to use a machine encoding that the investigator's tools can't interpret. In other words, implement your rootkit code in terms of an arbitrary bytecode, which is interpreted at runtime by an embedded runtime environment.

In this case the stub program, which normally unpacks or encrypts, is supplanted by a virtual machine. You could also merge this functionality together so that the stub program unpacks (or decrypts) its payload and then begins executing the bytecode inside. Again this tactic is far from perfect. Eventually the investigator would be able to reverse your bytecode and perform a static analysis.

Key Management Revisited: If you implement a packer, this won't be an issue. However, if you employ a cryptor you'll need to worry about where to store the encryption key that the stub uses to decrypt the original application. One approach is to partition the original code into different segments and use a different encryption key for each segment. The encryption keys can be generated at runtime from some seed value so that they're not readily available during static analysis.

A second approach would be to store the encryption key(s) outside of the stub executable, perhaps as an encoded series of bytes in an unused sector of a reserved disk region. Or perhaps you could stash an external key is PCI-ROM.

A third approach would be to use an external key that depends upon distinct environmental characteristics of the target machine (service tag number, NIC MAC address, etc.). This is what's been referred to as an "environmental key," an idea presented in a paper written by James Riordan and Bruce Schneier [37]. The BRADLEY virus uses environmental key generation to support code armoring [38].

Camouflage: The problem with packed/encrypted executables is that they have very little, if any, embedded string data and very few imports/exports. This is a telltale sign to the investigator that they're dealing with potentially malicious software. One way to address this issue is to wrap the stub with valid code so that it looks like a legitimate application. It's also possible to decorate the stub by inserting string data into it at compile time so that the investigator has something to read when they dump it. The more technical this sort of content is the better. You want the investigator to think that they're dealing with an offbeat value-added tool.

To further give the impression of authenticity, a resource definition script can be embedded into an application's `.rsrc` section at link time.

```
1 VERSIONINFO
FILEVERSION 3,0,0,5
PRODUCTVERSION 7,0,0,1
{
	BLOCK "StringFileInfo"
	{
		BLOCK "040904E4"
		{
			VALUE "CompanyName", "Microsoft Corporation"
			VALUE "FileVersion", "3.0.0.5"
			VALUE "FileDescription", "COM Object handler"
			VALUE "InternalName", "ATLHndlr"
			VALUE "LegalCopyright", "© Microsoft Corp."
			VALUE "OriginalFilename", "OLEObjMgr.exe"
			VALUE "ProductName", "Microsoft® Windows®"
```

```
                    VALUE "ProductVersion", "7.0.0.1"
              }
       }
       BLOCK "VarFileInfo"
       {
              VALUE "Translation", 0x0409, 1252
       }
}
```

Limitations: Using a packer or a cryptor should be viewed as a short-term defense (another hurdle in a long series of hurdles that you've set up to buy time). Inevitably, your program must run. Once the stub program has decoded its payload, the investigator can look at your executable in its original form in memory with a debugger. This leads to the next section…

Runtime Analysis of an Unknown Executable

Having acquired whatever they could from an inanimate unknown executable. The investigator will follow up their static analysis by launching the executable in a controlled environment to see what it does at runtime (e.g. enable logging, trace API calls, track registry edits, monitor network connections, capture packets, etc.). This differs from live response in that the investigator can stage things carefully in advance so that they can collect the maximum amount of useful information. To this end, virtual machines are often utilized so that repeated test runs can be made from a consistent starting state and also to sandbox the unknown executable. A debugger may also be brought into play so that the investigator can watch events unfold in slow motion.

The anti-forensic techniques that can be brought into play are the same set of tools that software vendors use to implement client-side software protection, namely:

- Tamperproofing
- Obfuscation
- Autonomy

NOTE

It's ironic that surmounting these anti-forensic tactics requires the forensic investigator to adopt the vantage point of a software cracker. In this scenario, traditional roles are inverted. The Black Hat is trying to protect their code and the White Hat is trying to reverse it. The White Hat becomes a Black Hat and the Black Hat becomes a White Hat!

Tamperproofing: The practice of "potting" electronic devices (also known as encapsulation) involves embedding the board components in a thermosetting resin to frustrate reverse engineering attempts. Tamperproofing is similar in spirit. It aims at making it difficult to modify a software application. Tamperproofing is relevant because most debuggers tend to alter the program they're inspecting (e.g. software breakpoints, inserting NOP instructions, etc.). To foil the investigators efforts to poke around we'll need to detect the presence of a debugger and then respond appropriately.

Detecting a Debugger: There are routines in both the Windows SDK and the WDK that detect the presence of a debugger.

```
BOOL WINAPI IsDebuggerPresent(void);   //called in user-mode
BOOLEAN KdRefreshDebuggerNotPresent(); //called in kernel-mode
```

The first routine indicates if the current process is running under the auspices of a user-mode debugger. It resolves to all of four lines of assembly code:

```
mov eax,dword ptr fs:[00000018h]
mov eax,dword ptr [eax+30h]
movzx eax,byte ptr [eax+2]
ret
```

A stealthy debugger might hook this call so that it always returns zero. One way to counter this is to sidestep the routine by inserting this assembly code directly into your application. The stealthy debugger could counter this by defense by directly altering the memory that the assembly code is referencing, which happens to be the `BeingDebugged` field in the Process Environment Block (PEB) of the application [39].

The second routine indicates if a kernel-mode debugger is active. It refreshes and then returns the value of `KD_DEBUGGER_NOT_PRESENT` global kernel variable. The same sort of interplay as described above, in terms of hiding a debugger and revealing it, can occur.

Given the limited efficacy of these API calls, another way we can detect the presence of tampering is by using checksums. This is essentially how Microsoft implements Kernel Patch Protection [40].

The trick to using checksums is to flood the reverser with a barrage of checks. Implement duplicate checks, rather than rely on a centralized API, and then make the checks dependent upon each other (have one checksum routine verify the validity of other checksum routines). This will prevent the reverser from isolating and disabling your checks by tweaking bytes in a single location. You should force them to track down dozens of checks.

You can augment this approach by building in routines that reinstate important sections of code at runtime in the event that a reverser has tried to NOP out integrity checks. It goes without saying that you'll probably want to implement your checksum code in such a manner that it doesn't look like checksum code. This calls for a bit obfuscation and misdirection (something that we'll discuss in short order). You may even want to institute an obvious set of fake integrity checks to lure the reverser away from the actual implementations.

Responding: The most effective landmines are the ones that are triggered silently, well in advance of the resulting detonation. In this spirit, you should try to situate your integrity checking code outside the perimeter of the code which responds to the modification.

It would be unwise to have your program shutdown abruptly. This would be too much of a tip off. One alternative is to introduce subtle bugs into your code and have the integrity checking code fix them if and only if it determines that the code under scrutiny has not been modified. These bugs should cause your application to behave erratically somewhere further down the line, making it difficult for the cracker to discover the cause and, more importantly, the location of the integrity checks.

Obfuscation: The goal of obfuscation is to recast an application into an equivalent form that's more complicated, such that the original intent of the code is difficult to recover. At the same time we'd like the program to function properly and maintain the readability of the code base. This is one reason why obfuscation is often performed at the machine code level, after an application has been compiled. *Code Morphing* is a variation of this tactic that represents a program using a machine neutral bytecode and then obfuscates the bytecode, such that complexity is added at the intermediate level, before machine code is generated [41].

With regard to obfuscation, there are tactics that can be applied to data and tactics that can be applied to code.

Data Encoding: One way to muddy the water is to alter how the bits within a variable are used to represent values. For example, on a platform that normally operates on integer values in little-endian format, store values in the big-endian format.

Data Aggregation: Aggregation determines how variables are grouped together to create compound data types. In the extreme case, you could merge all of the variables of an application into a heavily nested, arbitrarily sized, super structure replete with decoy fields and a custom heap.

Data Ordering: Ordering defines how related storage is arranged in memory. For example, you could alter the ordering of your data by interleaving several arrays into one big array to confuse someone trying to follow your code.

NOTE

It goes without saying that it's best to use these tactics concurrently; which is to say that you could construct a massive structure consisting of interleaved compound data types that use varying types of data encoding.

Reducing Code Abstraction: Procedure-based languages like C and FORTRAN emerged because, after a certain point, assembly code became too complicated to maintain. These languages keep complexity at bay by enabling the programmer to work at a higher level of abstraction. Hence, one way to make life more difficult for the investigator is to do away with higher-level constructs that would be visible to someone looking at an assembly code dump.

For example, to prevent the consolidation of logic you can *expand routines in-line* (which can often be implemented by toggling a compiler setting). A variation of this would be to have multiple versions of the same routine which are invoked at random, making it difficult for the investigator to nail down exactly where a program's logic is implemented.

On the other hand, you could take the concept of consolidation to an extreme and force all function calls to go through a *central dispatch routine*. Internally, this dispatch routine could use an address table to map return addresses on the caller's stack to specific functions. This way the dispatcher would know how to route different invocations.

Rearrange Code: Not all statements in a given routine are sequentially dependent. Hence, one way to sow confusion is to identify statements that are relatively independent and then shuffle them around. For added effect, this technique can be used with *code interleaving*, where the statements that make up routines are mixed together in what appears to be a single routine that uses opaque predicates to segregate the different code blocks.

Breaking Code Conventions: Most developers are taught that exception handling should not be used to intentionally transfer program control. This is one case where you should break this convention. Go ahead and perform non-local jumps by throwing exceptions. Try to make it look like error-handling so that the casual observer thinks that you're just being cautious.

Code Encryption: It's not unheard of to encrypt and decrypt sensitive routines at runtime, though such routines would probably not be thread-safe (as it's dangerous to have several threads concurrently encrypting/decrypting the same region of memory).

Then there's also the issue of key management. One way to solve this problem is to use a sensitive routine's checksum as the encryption key. You also may want to encrypt several non-critical routines so that you can avoid tipping off the investigator as to where the interesting stuff is hidden. Furthermore, you should make sure that the executable sections containing the sensitive routines are writable (using compiler-based configuration settings).

One final caveat of this tactic is that you'll probably want the encrypted code to start off in its encrypted state when the application loads into memory. To this

end, you can open up the compiled executable with a hex editor and manually overwrite a routine with its encrypted equivalent.

Table 2

Obfuscation Summary

Obfuscation Strategy	Tactical Implementations
Data Encoding	Use big-endian values on a little-endian platform
Data Aggregation	Merge variables into one big structure
Data Ordering	Array interleaving
Reducing Code Abstraction	Inline expansion, redundancy, centralized dispatching
Rearranging Code	Routine interleaving
Breaking Code Conventions	Exception handling as a flow control mechanism
Code Encryption	Use routine checksums as an encryption key

Automation: If you employ obfuscation at the bytecode or machine code level, you probably won't want to do it by hand. Instead, rely on an automated tool to obfuscate for you. There are solid commercial tools out there, like StrongBit's EXECryptor package [42] which offer code morphing features. However, if you have the requisite motivation and resources there's nothing stopping you from building your own tool to acquire an added degree of control and to help guard against automated deobfuscators.

Autonomy: Using the official API to get things done will leave a perceptible trail. In the absence of a debugger, the investigator can still crank up their audit settings to the max, trace API calls, capture network packets, etc. By staying off the beaten path, so to speak, you can make it harder for the investigator to see what you're doing. Fair warning, while this is a very powerful strategy, it can be equally difficult to implement.

This is the beauty of self-reliant code. The basic idea is to make the interface between your code and the native OS as small as possible. Rather than rely on the native TCP/IP stack, install and use your own. This will allow you to sidestep the host-based firewall and associate network connection logging. Rather than reading specific registry directly, make a copy of the relevant hives and read the copies. Rather than rely on native drivers and the Windows Hardware Abstraction Layer, communicate directly with hardware. The fewer subsystems you interact with, the harder it is for them to tattle on you.

At Black Hat Europe in 2006, Joanna Rutkowska's demonstrated a tool named deepdoor that implemented this train of thought [43]. The deepdoor rootkit was a relatively self-contained chunk of code that hid out in kernel space by patching a series of bytes in the NDIS data section. In other words, the only interface between the rootkit and the operating system was a couple of double words in the Windows networking subsystem.

A couple of years later, at Black Hat USA 2008, Sherri Sparks and Shawn Embleton kicked things up a notch with the Deeper Door [44]. In this case the researchers implemented a rootkit that interacted directly with the host machine's NIC hardware in an effort to bypass local security software.

Staying Memory Resident

> "True victory lies in breaking the enemy's resistance without fighting."
> —Sun Tzu

An ex-convict once told me that the surest way to out-smart a prison guard was never to end up in prison to begin with. The same thing could probably be said with regard to disk analysis. The best way to defeat an investigator performing a post mortem of a hard drive is probably never to end up on the drive to begin with.

Up until this point, we've assumed that our rootkit will persist on disk somewhere. Now we're going to loosen this constraint so that we can introduce the anti-forensic strategy of Data Source Elimination. There are a number of interesting ways to realize this strategy, though all of the variations that we'll examine revolve around the same basic goal: finding ways to execute an arbitrary stream of machine instructions on a targeted machine while storing little or nothing on disk.

Syscall Proxying

This is a technique that was originally popularized by Maximiliano Cáceres [45]. The basic idea behind this technique is to install a syscall server on a compromised machine that will execute low-level system calls on behalf of a specially written remote application running on the attacker's machine. This application uses a syscall stub to marshal its system call input parameters, send requests over the network to the server component, and then marshal output

parameters returned by the server. The general architecture of this scheme is reminiscent of the RPC model.

As the specially written application requests system services from the kernel, these requests are ferried over to the compromised machine via the extra syscall client and server layers. This gives the attacker the ability to launch an arbitrary user-mode program on their machine, which appears to be running locally. When, in reality, the end result of execution is taking place on the compromised machine.

There are a number of commercial tools that implement syscall proxying. For example, Core Security Technologies sells an automated exploitation tool named CORE IMPACT that uses this technique [46].

Though this novel approach is theoretically sweet, it has demonstrated drawbacks in the field. Performance is an issue, which isn't surprising given the number of system calls that the average application makes and the work that's required to marshal and un-marshal the associated parameters. There have also been problems associated with remote execution of `fork()`-like system calls.

Memory Resident Development Tools

A memory resident compiler allows the attacker to send source code snippets to a compromised machine that are then translated to position independent machine code. This code can be loaded and executed on the fly. The benefit of this tactic is that the attacker can focus on program logic rather than get bogged down in the details of a given platform-specific machine encoding. The drawbacks include all of the complexity and headaches associated with maintaining a compiler.

The MOSDEF compiler, originally developed by Dave Aitel, is a widely acknowledged example of this approach [47][48]. Written in Python, it was originally designed to support Immunity's CANVAS exploit framework. It has been released to the public under the LGPL license. The MOSDEF 2.0 compiler can be downloaded for free from Immunity's web site [49].

Data Contraception

Data Contraception, an approach conceived by the grugq [23], involves staying memory resident by means of a server component which offers access to its address space (or that of another application) on the compromised host. The client component, running on the attacker's machine, feeds a stream of executable code to the server. The server takes this stream of bytes and loads it into its address space, ultimately passing program control to the entry point of the loaded code.

NOTE

The grugq's server can be viewed as a network-enabled extension of the stub application that we encountered back in our discussion of packers and cryptors. Like the stub, the server is type of Userland Exec. The difference is that, in this scenario, the byte stream being loaded is retrieved from a network connection rather than the hard drive.

In an effort to minimize the amount of forensic evidence, data contraception mandates that the server component should be a common utility that won't stand out too much (i.e. a known good file) rather than a custom-built tool. The general-purpose nature of the server tends to force much of the work on to the client, which can end up being rather complicated.

The grugq's proof-of-concept server, named Remote Exec, was a combination of the GNU Project Debugger (`gdb`) and a custom library named `ul_exec` (as in Userland Exec) which was one of the grugq's earlier projects [50]. The `ul_exec` library does most of the real work necessary to load and execute an arbitrary byte stream. It clears out a region in memory, maps the program's segments into this region, sets up the stack, and transfers program control to the new entry point.

Userland Exec: Additional Work

Additional work has been done to revisit the implementation of the grugq's Userland Exec to make it amenable to use in an exploit's payload [51]. Also, on the Windows front, research that precedes the grugq's was done by Gary Nebbett (author of venerated *Windows NT/2000 Native API Reference*). Gary developed an in-memory loading technique that's referred to as "Nebbett's Shuttle," which was covered in detail by Nick Harbour at Black Hat USA 2007.

In-Memory Library Injection

In-memory library injection is a variation of the Userland Exec approach. Instead of loading an entire executable into the host's address space, this tactic involves loading a dynamically linked library without touching disk storage. The steps necessary to implement this tactic have been published in an article by Skape and Jarkko Turkulainen [52]. For example, on the Windows side of the fence, loading a DLL in-memory over a network connection can be performed using the following steps:

1. Load the system libraries that will be used during following steps
2. Read the size of the DLL byte stream from a socket
3. Use the above size to allocate memory for the DLL byte stream
4. Read the DLL from the socket and store it in the allocated memory
5. Establish and initialize the DLL's sections in memory
6. Hook the system calls used for loading DLLs
7. Load DLL with a predetermined pathname

The real voodoo occurs in terms of hooking the system calls normally used to load DLLs. These libraries are built from the ground up with the expectation that they'll be loading a DLL that resides on disk. To fool them into doing otherwise, extra functionality has to be introduced and the best way to do that is by intercepting the path of execution.

In-memory library injection has been used by the Metasploit Meta-Interpreter (i.e. Meterpreter). The Meterpreter is essentially a payload within the Metasploit framework that can be extended at runtime. In other words, it offers a shell that allows you to dynamically add new features. It facilitates this by enabling developers to write their own DLLs and then inject them into the address space of an exploited process [53]. The canonical example of a Meterpreter extension would be Sam Juicer, a DLL that dumps hashes from the Windows SAM database [54].

A Hybrid Approach: Persisting without Persisting

One way to have your cake and eat it too may be to rely on a heartbeat monitor. This approach is based on ideas presented by Joanna Rutkowska at Black Hat Federal in 2006 [43]. In this scenario, compromised machines run a memory resident rootkit that periodically emits a predefined network signal. This signal is detected by a heartbeat monitoring station. If the heartbeat monitor fails to detect

a signal from a machine that's known to be compromised, the monitor assumes that the machine has been restarted and proceeds to reinstall the deposed rootkit via some zero-day exploit.

Firmware-Based Rootkits

> "The enemy's spies who have come to spy on us must be sought out,
> Tempted with bribes, led away, and comfortably housed. Thus they
> Will become converted spies and available for our service."
> —Sun Tzu

Yet another way to stay off the disk and stymie a post-mortem is to hide in the firmware. Strictly speaking, this is less effective than staying memory resident as an anti-forensic tactic because its leaves more artifacts behind (more evidence for the investigator). Depending on your target, it can also be a significant investment with regard to R&D because of the hardware-specific elements involved. Nevertheless, in some circumstances it can be a viable option worth considering.

Publicly Available Research

The BIOS-based rootkit is an elusive species. There hasn't been a lot of work done, but the research that has been performed is instructive. In 2006 and 2007, John Heasman gave a number of well-received Black Hat presentations on BIOS-based Rootkits [55][56] and hacking the Extensible Firmware Interface [57]. More recently, at CanSecWest 2009, Anibal Sacco and Alfredo Ortega discussed how to hack the Phoenix Award BIOS [58]. If you want to sift through all of the gory details, there have been books written on the subject [59].

A Commercial Implementation

In terms of related commercial products, Absolute Software sells a product named Computrace that consists of two distinct components:

- An Application Agent
- A Persistence Module

The application agent (`rpcnet.exe`) is a user-mode service that phones home with inventory tracking data. The persistence module runs quietly in the background, checking to see if the application agent needs to be reinstalled.

Recently, Absolute Software has worked with hardware OEMs to implant the persistence module in the system BIOS [60]. It's not impossible to envision a scenario where someone finds a way to subvert the persistence module to reinstall an arbitrary agent, turning the spy against its original handlers…

Commentary

"When using our forces, we must seem inactive"
—Sun Tzu

We've waded around in the murky depths for a while. It's time to surface again for air. Let's step back from the trees to view the forest and consider the high level anti-forensic issues that a rootkit architect must address.

The Tradeoff: Footprint and Failover

Over the course of this paper we've looked at a number of solutions to the problem of disk analysis. So then, what's the best way to protect against a post-mortem disk analysis? Should we reside on disk and use a multi-layered defense? Or should we just stay Memory Resident? Or maybe we can get away with hiding in ROM?

There's no perfect solution. Different jobs require different tools. The ideal solution that works well in every environment doesn't exist. You'll have to evaluate your needs and decide which set of tradeoffs you can live with. Residing on disk will allow you to survive reboot, but will also leave a footprint on disk. Staying memory resident will minimize your footprint at the expense of resiliency in terms of system shutdown.

Ultimately, the solution you choose will reflect several factors:

- What's the nature of the machine(s) being targeted?
- How valuable is the data being sought after?
- What kind of security professionals are you dealing with?

If the deployment you're targeting involves a cluster of high-end mainframes, you can assume availability and stay memory resident. On the other hand, if you've staked out an outpost on an executive's laptop, then you'll probably want to store some components to disk (at least until you find a more stable environment).

Likewise, if the data you're after is classified as Top Secret and is guarded by a phalanx of dedicated security professionals, the ROI probably justifies the effort necessary to build an exact replica of the target environment and to develop a special-purpose custom solution to ensure that your code doesn't leave any artifacts behind. In other words, we're talking about an exotic one-off rootkit that would have limited utility outside the immediate area of operation.

Logistical Issues

You can destroy, hide, transform, fabricate, and contain all you want. Somewhere along the line you'll still need a way to get the ball rolling. In other words, execution has to start somewhere, so how do we launch our rootkit? Will this code necessarily have to reside on disk?

Recall that we're assuming the worst-case scenario; that we're facing off against a highly skilled analyst who has the time, resources, knowledge base, and mandate to do what needs to be done to unearth our rootkit.

Hence, the best way to initiate rootkit execution is probably through an exploit because doing so relies solely on a flaw that's located in existing executable, nothing new is introduced on disk. If possible, you should use a zero-day exploit that not many people know about.

NOTE

This leads one to ponder if flaws have intentionally been left as backdoors disguised as a legitimate bug. There've been rumors of this type of tactic being implemented by American intelligence agencies at the hardware level [61]. We're not alone in this sort of skullduggery. China is also rumored to have implemented this sort of tactic [62].

You may feel the need to hedge your bet. After all, even high-end financial institutions have been known to occasionally shut down their servers. For example, it's been reported by Microsoft that the Chicago Stock Exchange power cycles its systems every day after the market closes [63]. My guess is that this is done to protect against memory leaks and other subtle problems.

In you need to persist in this sort of environment; you may want to consider a BIOS-based persistence module that will reintroduce your rootkit in memory in the event of a system restart. If implementing a BIOS-based initiator isn't a feasible option, you'll have to fall back on finding refuge on disk. In this case,

you should expect that your rootkit launch code, which is naked (unencrypted) by necessity, will inevitably be discovered. Hence you should implement a minimal loader that acts like Data Contraception server which re-introduces the rootkit proper into its address space.

There are endless variations. For example, you could implement a persistence module in the BIOS that guarantees the presence of a Data Contraception server on disk, which in turn loads a rootkit.

Coming Full Circle

Forensic investigators and their ilk have openly bemoaned the fact that they can't keep up with the Black Hats [7]. In my opinion, state of the art anti-forensic tactics have pretty much rendered post-mortem disk analysis ineffective. This pushes us right back to where we started, to the arena of live incident response and network security monitoring. Perhaps this also explains why no one is talking about disk-based anti-forensics! It's in these two areas (live response and network monitoring) that the arms race between White Hats and Black Hats continues. A rootkit may not use disk storage at all, *but it still has to execute in memory and it almost always communicates with the outside*.

References

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

[2] http://www.techpathways.com/webhelp/ProDiscoverHelp.htm.

[3]Darren Bilby, "Low Down and Dirty: Anti-Forensic Rootkits" (presented at Black Hat Japan 2006), http://www.blackhat.com/presentations/bh-jp-06/BH-JP-06-Bilby-up.pdf.

[4] Paul McDougall, "Microsoft Buys Anti-Rootkit Vendor Komoku," *Information Week*, March 24, 2008.

[5]Joanna Rutkowska, "Beyond The CPU: Defeating Hardware Based RAM Acquisition. (part I: AMD case)" (presented at Black Hat DC 2007), http://www.blackhat.com/presentations/bh-dc-07/Rutkowska/Presentation/bh-dc-07-Rutkowska-up.pdf.

[6] Grugq, "Defeating Forensic Analysis on Unix," *Phrack*, Issue 59, 2002.

[7] Marc Rogers, "Anti-Forensics" (presented at Lockheed Martin, San Diego, September 15, 2005), http://cyberforensics.purdue.edu/Presentations.aspx.

[8] Keith J. Jones, Richard Bejtlich, and Curtis W. Rose, *Real Digital Forensics: Computer Security and Incident Response*, Addison-Wesley, October 3, 2005.

[9] Mayank R. Gupta, Michael D. Hoeschele, and Marcus K. Rogers, "Hidden Disk Areas: HPA and DCO," *International Journal of Digital Evidence*, Fall 2006, Volume 5, Issue 1.

[10] http://www.guidancesoftware.com/computer-forensics-software-fastbloc.htm.

[11]Irby Thompson and Mathew Monroe, "FragFS: An Advanced Data Hiding Technique" (presented at Black Hat DC 2006), http://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Thompson/BH-Fed-06-Thompson-up.pdf.

[12] James Foster and Vinnie Liu, "Catch me, if you can…" (presented at Black Hat USA 2005), http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-foster-liu-update.pdf.

[13] Grugq, "The Art of Defiling" (presented at Black Hat Europe 2005), http://www.blackhat.com/presentations/bh-europe-05/bh-eu-05-grugq.pdf.

[14] Microsoft Corporation, *NTFS Technical Reference*, March 28, 2003, http://technet.microsoft.com/en-us/library/cc758691.aspx.

[15]Brian Carrier, *File System Forensic Analysis*, Addison-Wesley, March 27, 2005.

[16] http://linux-ntfs.org/doku.php.

[17] Bill Blunden, *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*, Jones & Bartlett Publishers, May 4, 2009.

[18]Gina Kolata, "Veiled Messages of Terrorists May Lurk in Cyberspace," *New York Times*, October 30, 2001.

[19] Mihir Nanavati and Bhavesh Kothari, "Detection of Rootkits in the Windows Registry," MIEL e-Security Pvt. Ltd., Mumbai, http://helios.miel-labs.com/downloads/registry.pdf.

[20] Richard Kissel, Matthew Scholl, Steven Skolochenko, and Xing Li, *Special Publication 800-88: Guidelines for Media Sanitization*, National Institute of Standards and Technology, September, 2006, http://csrc.nist.gov/publications/nistpubs/800-88/NISTSP800-88_rev1.pdf .

[21] R. Sekar and A.K. Pujari (Eds.): ICISS 2008, LNCS 5352, pp. 243–257, 2008.

[22] http://cmrr.ucsd.edu/people/Hughes/SecureErase.shtml.

[23] Grugq, "FIST! FIST! FIST! It's All in the Wrist: Remote Exec," *Phrack*, Issue 62, 2004.

[24] John Viega and Gary McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*, Addison-Wesley, October 4, 2001.

[25] http://msdn.microsoft.com/en-us/library/aa366895(VS.85).aspx.

[26] http://www.opengroup.org/onlinepubs/007908775/xsh/mlock.html.

[27]Dawid Golunski, "Rogue Binaries: How to Own the Software," *hakin9*, 1/2008.

[28] http://www.gnu.org/software/diffutils.

[29]Vincent Teresa, *My Life in the Mafia*, Doubleday & Company, January 1, 1973.

[30] http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx.

[31]http://www.metasploit.com/data/antiforensics/BlueHatMetasploit_AntiForensics.ppt.

[32] Philippe Biondi and Fabrice Desclaux, "Silver Needle in the Skype" (presented at Black Hat Europe, 2006), http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-biondi/bh-eu-06-biondi-up.pdf.

[33] http://upx.sourceforge.net.

[34] http://yodap.sourceforge.net/download.htm.

[35] http://www.peid.info.

[36]BigBoote, "How to Write Your Own Packer," *CodeBreakers Magazine*, Volume 1, Number 2, 2006, http://www.codebreakers-journal.com.

[37] James Riordan and Bruce Schneier, "Environmental Key Generation towards Clueless Agents," *Mobile Agents and Security*, G. Vigna, ed., Springer-Verlag, 1998, pp. 15-24.

[38] Eric Filiol, "Strong Cryptography Armoured Computer Viruses Forbidding Code Analysis: The Bradley Virus," Proceedings of the 14th EICAR Conference, 2005.

[39] Nicolas Falliere, "Windows Anti-Debug Reference," *SecurityFocus*, September 12, 2007, http://www.securityfocus.com/infocus/1893.

[40] Skywing, "PatchGuard Reloaded: A Brief Analysis of PatchGuard Ver. 3," *Uninformed*, September, 2007, http://uninformed.org/?v=all&a=38&t=sumry.

[41]Mental Driller, "How I made MetaPHOR and what I've learnt," *29A e-zine*, Number 6, http://vx.netlux.org/29a/29a-6/29a-6.205.

[42] http://www.strongbit.com/execryptor.asp.

[43]Joanna Rutkowska, "Rootkit Hunting vs. Compromise Detection" (presented at Black Hat Federal 2006), http://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Rutkowska/BH-Fed-06-Rutkowska-up.pdf.

[44] Sherri Sparks and Shawn Embleton, "Deeper Door - Exploiting the NIC Chipset" (presented at Black Hat USA 2008).

[45] Maximiliano Cáceres, "Syscall Proxying: Simulating Remote Execution" (presented at Black Hat USA 2002), http://www.blackhat.com/presentations/bh-usa-02/caceres/bh-us-02-caceres-syscall.ppt.

[46] http://www.coresecurity.com/content/core-impact-overview.

[47] http://www.immunitysec.com/products-canvas.shtml.

[48] http://www.immunitysec.com/downloads/MOSDEF.ppt.

[49] http://www.immunitysec.com/downloads/MOSDEF2.0.pdf.

[50] Grugq, "The Design and Implementation of Userland Exec," http://archive.cert.uni-stuttgart.de/bugtraq/2004/01/msg00002.html.

[51] Pluf & Ripe "Advanced Anti-Forensics: SELF." *Phrack*, Issue 63, 2005.

[52] Skape and Jarkko Turkulainen, "Remote Library Injection," *nologin,* April 19, 2004, http://www.nologin.org/Downloads/Papers/remote-library-injection.pdf.

[53] Skape, "Metasploit's Meterpreter," *nologin*, January 12, 2005, http://www.nologin.org/Downloads/Papers/meterpreter.pdf.

[54] http://www.metasploit.com/data/antiforensics/SamJuicer.zip.

[55]John Heasman, "Implementing and Detecting An ACPI BIOS Rootkit" (presented at Black Hat Federal 2006), http://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Heasman.pdf.

[56] John Heasman, "Implementing and Detecting a PCI Rootkit" (presented at Black Hat Federal 2007), http://www.ngssoftware.com/research/papers/Implementing_And_Detecting_A_PCI_Rootkit.pdf.

[57] John Heasman, "Hacking the Extensible Firmware Interface" (presented at Black Hat USA 2007), http://www.ngssoftware.com/research/papers/BH-VEGAS-07-Heasman.pdf.

[58] Anibal Sacco and Alfredo Ortega, "Persistent BIOS Infection" (presented at CanSecWest 2009), http://cansecwest.com/csw09/csw09-sacco-ortega.pdf.

[59]Darmawan Salihun, *BIOS Disassembly Ninjutsu Uncovered*, A-List Publishing, October 28, 2006.

[60] http://www.absolute.com/resources/public/Datasheet/CT-FAQ-TEC-E-030309.pdf.

[61]David Sanger, "U.S. Steps Up Effort on Digital Defenses," *New York Times*, April 27, 2009.

[62]Michael Smith, "Spy chiefs fear Chinese cyber attack," *The Sunday Times*, March 29, 2009.

[63] Microsoft Corporation, *Windows NT Server Tour, 1999*, http://staging.glg.com/tourwindowsntserver/CHX/pdf/tech_road.pdf.