5 Defense and Analysis Techniques

5.1 Memory Forensics

Memory forensics refers to finding and extracting forensic artifacts from a computer's physical memory. This section explains the importance and capabilities of memory forensics and the tools used to support incident response and malware analysis.

While a system is on, random access memory (RAM) contains critical information about the current state of the system. By capturing an entire copy of RAM and analyzing it on a separate computer, it is possible to reconstruct the state of the original system, including the applications the user was running and the files or network connections that existed at the time. The concept of preserving RAM per the "order of volatility"[1] and inspecting it for signs of an intrusion is certainly not new; however, before the recent explosion of groundbreaking research and expandable analysis frameworks, many investigators relied on running the strings command on a memory dump to gather postmortem intelligence about an attack. Fortunately, times have changed, and memory analysis is not only a critical component in any forensic investigation, but also one of the most effective methods for malware reverse-engineering tasks such as unpacking and rootkit detection.

5.1.1 Why Memory Forensics Is Important

Analysts who bring memory forensics skills to an investigation are better equipped to handle malware incidents than analysts who do not have such skills. Here are a few reasons why:

- Attackers design some malware to run completely from RAM (i.e., memory resident codes) to avoid touching longer term storage devices such as the hard drive. Therefore, if analysts do not look for signs of intrusions in RAM, they might miss the most important, or perhaps the only, evidence that malware existed on the system.

- Attackers design some malware to hide its own code and the resources that it requires from the operating system using application program interface (API) hooks; however, these rootkit techniques typically only work against other processes on the infected computer while the system is running. Hiding from offline memory forensics tools requires a different set of capabilities that most malware authors have not implemented into their code.
- Similar to what Isaac Newton theorized about the real world, every action on a computer has a reaction. Even if attackers were able to study the Windows operating system (OS) well enough to anticipate the side effects of every API call, they would not be able to prevent or hide each side effect continuously and perpetually. If investigators become familiar with these side effects, they can use the information as clues when determining what might have happened on the suspect system.

5.1.2 Capabilities of Memory Forensics

Analysts can gather an extreme amount of information about the state of a system by using memory forensics. Table 5-1 shows a few of the default capabilities of a memory analysis framework and the corresponding tools that one might use on a live system to gather the same type of evidence.

Based on the information in Table 5-1, memory forensics frameworks can produce the same information that 10–20 standard tools that analysts frequently use on live systems can, but with the added benefit of being able to bypass rootkit tricks.

5.1.3 Memory Analysis Frameworks

In terms of memory analysis frameworks, there are a few options from which to choose. The most important factors are cost, the programming language for developing plug-ins for the frameworks, the

Table 5-1 Default Capabilities and Corresponding Tools

CAPABILITY	LIVE SYSTEM TOOLS
Determine which processes and threads were active at the time an analyst obtained the memory dump, including the process ID, thread ID, and process start and end times.	Process Explorer, Task Manager
Enumerate the dynamic link libraries (DLLs) loaded in any process, including their base address in virtual memory, the size of the DLL, and the full path to the DLL on disk.	Process Explorer, listdlls.exe
Determine which ports and protocols are in use, the local and remote Internet Protocol (IP) endpoints, and the process identifier (PID) of the process responsible for creating the connection or socket.	Fport, ActivePorts, TcpView, Netstat
Determine which kernel modules are loaded, including their base addresses, sizes, and names.	GMER, IceSword, WinDBG
Dump malicious process executables, DLLs, kernel drivers, and any nonpaged memory ranges in user mode or kernel mode memory for further inspection.	LordPE, Procdump, Debugger plug-ins
Print the addresses and sizes of all allocated memory regions in a process, including the page permissions and whether the region contains a memory-mapped file.	Vmmap, OllyDbg
Determine which files and registry keys were open in a process at the time of the memory dump.	Process Explorer, handles.exe

operating systems on which the frameworks run, and the reliability of the frameworks' output. See Table 5-2.

Because Volatility is free, is written in Python, and runs on multiple operating systems, it is the favorite framework of many iDefense engineers. Knowing how tools work, rather than just knowing how to use the tools, is a requirement to analyzing and understanding today's sophisticated malware. Volatility is open-source Python, so learning how Volatility harvests information is simple. In fact, one of the ways that iDefense engineers learned a lot about the technical aspects of memory analysis, including the format of kernel structures and how

Table 5-2 Memory Analysis Framework Factors

NAME	COST	PLUG-IN LANGUAGE	ANALYSIS OS
Volatility[a]	Free and open source	Python	Windows, Linux, and OSX
HBGary Responder[b]	$1,500–9,000	C#	Windows only
Mandiant Memoryze[c]	Free and closed source	XML and proprietary	Windows only

Source: a: Volatile Systems, "Volatility Website," 2006–2008, https://www.volatilesystems.com/default/volatility. b: HBGary, "HBGary Responder Tool," 2009, https://www.hbgary.com/products-services/responder-prof/; and c: Mandiant, "Free Software: Memoryze," 2010, http://www.mandiant.com/software/memoryze.htm.

to parse them, was by learning from Volatility's programmers by looking through the source code.

5.1.4 Dumping Physical Memory

To dump physical memory, iDefense recommends using win32dd[2] by Matthieu Suiche. The tool supports memory acquisition from a wide variety of OS versions, including Windows 2000, XP, 2003, Vista, 2008, 7, and 2008 RC2. Suiche recently provided an update that includes the capability to compute cryptographic checksums (MD5, SHA-1, or SHA-256) and client or server architecture so that an analyst can transmit the memory dump across the network easily. To get started, download a copy of win32dd from the tool's home page and extract the archive. To dump the full physical address space, save the output file to mem.dmp in the same path as win32dd and create a Secure Hash Algorithm 1 (SHA-1) hash of the dumped file; use the following syntax:

```
F:\>win32dd.exe /f mem.dmp /s 1
```

5.1.5 Installing and Using Volatility

To begin using Volatility, download the package from its home page on the Volatile Systems[3] website or grab a copy of the latest Subversion package[4] hosted at Google code. The Volatility Documentation Project[5] by Jamie Levy (a.k.a. gleeda) and a few anonymous authors contains some great manuals for installing Volatility on Windows, Linux, and OSX. In most cases, to get started, the only requirement is to extract the archive and invoke the "volatility" script with Python, as shown in the following command sequence:

```
$ tar -xvf Volatility-1.3.tar.gz
$ cd Volatility-1.3
$ python volatility

      Volatile Systems Volatility Framework v1.3
      Copyright (C) 2007,2008 Volatile Systems
      Copyright (C) 2007 Komoku, Inc.
      This is free software; see the source for
copying conditions.
      There is NO warranty; not even for
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

```
      usage: volatility cmd [cmd_opts]

      Run command cmd with options cmd_opts
      For help on a specific command, run 'volatility
cmd --help'

      Supported Internel Commands:
         connections  Print list of open connections
         connscan     Scan for connection objects
         connscan2    Scan for connection objects
                      (New)
         datetime     Get date/time information for
                      image
         dlllist      Print list of loaded dlls for
                      each process
         dmp2raw      Convert a crash dump to a raw
                      dump
         dmpchk       Dump crash dump information
         files        Print list of open files for
                      each process
         hibinfo      Convert hibernation file to
                      linear raw image
         ident        Identify image properties
         memdmp       Dump the addressable memory for
                      a process
         memmap       Print the memory map
         modscan      Scan for modules
         modscan2     Scan for module objects (New)
         modules      Print list of loaded modules
         procdump     Dump a process to an executable
                      sample
         pslist       Print list of running processes
         psscan       Scan for EPROCESS objects
         psscan2      Scan for process objects (New)
         raw2dmp      Convert a raw dump to a crash
                      dump
         regobjkeys   Print list of open regkeys for
                      each process
         sockets      Print list of open sockets
         sockscan     Scan for socket objects
         sockscan2    Scan for socket objects (New)
         strings      Match physical offsets to virtual
                      addresses (may take a while, VERY
                      verbose)
         thrdscan     Scan for ETHREAD objects
```

```
        thrdscan2     Scan for thread objects (New)
        vaddump       Dump the Vad sections to files
        vadinfo       Dump the VAD info
        vadwalk       Walk the vad tree
```

All of the commands shown in the output are available by default. Analysts can learn any required arguments for individual commands by issuing "python volatility <command> --help"; however, many of the commands work without arguments. The full syntax for extracting evidence from the memory dump created with Volatility follows:

```
$ python volatility <command> <arguments> -f mem.dmp
```

5.1.6 Finding Hidden Processes

The Windows kernel creates an EPROCESS object for every process on the system. The object contains a pair of pointers, which identifies the previous and subsequent processes. Together, this creates a chain of process objects also called a *doubly linked list*. To visualize a doubly linked list, think of a group of people who join hands until the group is standing in a big circle. By joining hands, each person connects to exactly two other people. To count the number of people in the group, one could pick a person to start with and then walk in either direction along the outside of the circle and count the number of heads until ending back at the starting point. Tools like Process Explorer, Task Manager, and many other system administration programs use API functions that enumerate processes by walking the linked list using this same methodology.

Enumerating processes in memory dumps is different because the system is offline and therefore API functions do not work. To find the **EPROCESS** objects, Volatility locates a symbol named _ **PsActiveProcessHead**, defined in ntoskrnl.exe. Although the symbol is not exported, it is accessible from the **_KPCR** structure, which exists at a hard-coded address in memory, as described in "Finding Some Non-Exported Kernel Variables in Windows XP"[6] by Edgar Barbosa. This **_PsActiveProcessHead** symbol is a global variable that points to the beginning of the doubly linked list of **EPROCESS** objects. Exhibit 5-1 shows the path that Volatility takes to find the desired data in a memory dump.

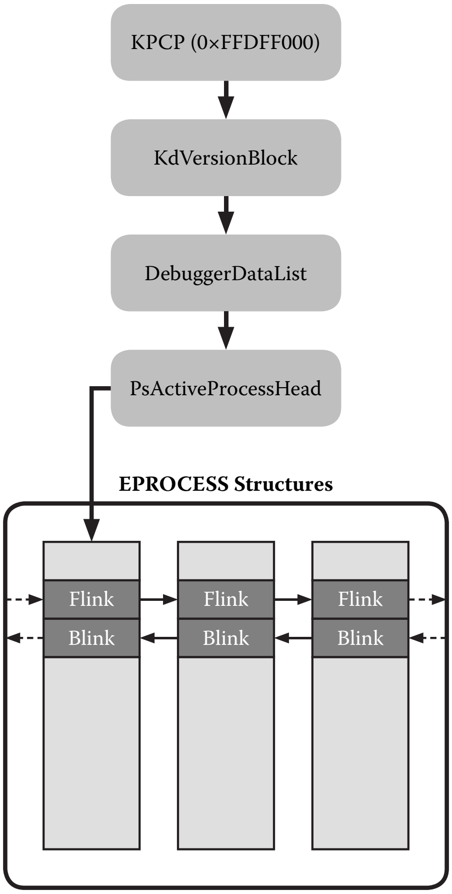


Exhibit 5-1 The path used by Volatility to locate the EPROCESS object list.

To use Volatility to generate a process listing by walking the linked list of processes, use the following syntax:

```
$ python volatility pslist –f mem.dmp
Name            Pid  PPid Thds Hnds Time
System          4    0    54   232  Thu Jan 01 00:00:00 1970
smss.exe        368  4    3    21   Tue Dec 01 15:58:54 2009
csrss.exe       516  368  10   324  Tue Dec 01 15:58:55 2009
winlogon.exe    540  368  18   505  Tue Dec 01 15:58:55 2009
services.exe    652  540  16   252  Tue Dec 01 15:58:55 2009
lsass.exe       664  540  21   326  Tue Dec 01 15:58:55 2009
VBoxService.exe 816  652  4    76   Tue Dec 01 15:58:55 2009
svchost.exe     828  652  19   196  Tue Dec 01 15:58:55 2009
svchost.exe     908  652  10   225  Tue Dec 01 15:58:55 2009
svchost.exe     1004 652  67   1085 Tue Dec 01 15:58:55 2009
svchost.exe     1064 652  5    57   Tue Dec 01 15:58:55 2009
svchost.exe     1120 652  15   205  Tue Dec 01 15:58:56 2009
```

```
spoolsv.exe      1528 652   12   111  Tue Dec 01 15:58:56 2009
explorer.exe     1572 1496  10   284  Tue Dec 01 15:58:56 2009
VBoxTray.exe     1644 1572  7    39   Tue Dec 01 15:58:57 2009
alg.exe          780  652   6    104  Tue Dec 01 15:59:07 2009
wscntfy.exe      696  1004  1    27   Tue Dec 01 15:59:09 2009
cmd.exe          984  1572  1    31   Tue Dec 01 16:05:26 2009
win32dd.exe      996  984   1    21   Tue Dec 01 16:05:42 2009
```

After understanding how the pslist command works, it is possible to evaluate why it might not always be reliable. One reason is due to rootkits that perform direct kernel object manipulation (DKOM). In the book *Rootkits: Subverting the Windows Kernel*, Greg Hoglund and James Bulter show how to hide processes by unlinking entries from the doubly linked list. The authors overwrite the forward link (Flink) and backward link (Blink) pointers of surrounding objects so that they point around the **EPROCESS** object that represents the process to hide. As shown in Exhibit 5-2, the overwriting effectively hides a process from any tool that relies on walking the linked list, regardless of if the tool runs on a live system or a memory dump. Since central processing unit (CPU) scheduling is thread based, the hidden process remains running on the operating system even when rootkits unlink the EPROCESS objects.

Consider the previous analogy of people joining hands and forming a circle to depict the doubly linked list depicted in Exhibit 5-1. If one person releases both hands to step outside the circle (see Exhibit 5-2), the people on the left and right will join hands and close the gap. The

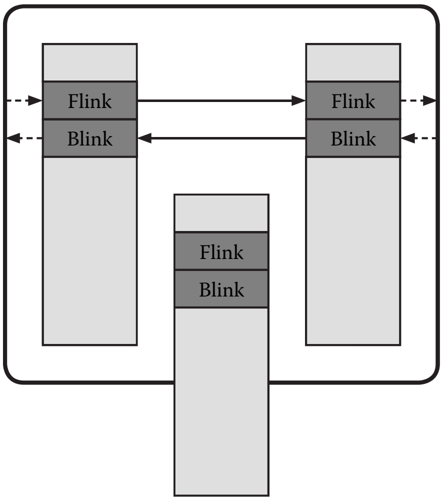


Exhibit 5-2 An EPROCESS object removed from a doubly linked list.

disconnected person does not disappear; instead, he or she is now free to walk about the room. Counting people using the original method will result in one fewer person; however, by changing techniques and scanning the entire room using a thermal imaging device, the results would be accurate, even if one or more people were no longer standing in the circle.

The Volatility command psscan2 is not exactly a thermal imaging device, but it works similarly in theory. Instead of walking the linked list of **EPROCESS** objects like pslist, psscan2 scans linear memory for pools with the same attributes (paged versus nonpaged, tag, and size) that the kernel uses for **EPROCESS** objects, and then applies a series of sanity checks. This way, psscan2 is able to find **EPROCESS** objects in memory even if a rootkit has unlinked it from the list. The same concept applies to finding hidden kernel drivers, sockets, connections, services, and various other kernel objects.

5.1.7 Volatility Analyst Pack

Volatility Analyst Pack (VAP)[7] is a collection of plug-ins designed for malware analysis and rootkit detection. Table 5-3 describes the purpose of the plug-ins and their statuses. If the status is "Public," then the plug-in is publicly available. If the status is "By request," then the plug-in is currently only available to iDefense customers upon request (BETA mode).

5.1.8 Conclusion

Memory forensics is a rapidly growing aspect of incident response and malware analysis. Its powerful default capabilities can replace 10–20 live system tools, not to mention the features provided by third-party plug-ins such as VAP. Although there are several options, iDefense recommends the free, open-source Volatility framework, which also provides an analyst with the opportunity to learn about the operating system.

5.2 Honeypots

Creating an asset to attract malicious activity for monitoring and early warning is a well-established activity. Not only do honeypots, isolated

Table 5-3 Plug-In Statuses and Descriptions

NAME	STATUS	DESCRIPTION
apihooks	Public	Detects Import Address Table (IAT), Export Address Table (EAT), and inline API hooks in user mode processes and kernel drivers.
callbackscan	By request	Scans for callback objects; this can expose rootkits that register system-wide notification routines (see notify_routines below).
csrss_pslist	By request	Detects hidden processes with csrss.exe handles and CsrRootProcess links.
debugged	By request	Detects debugged processes; this can expose attempts for malware to perform self-debugging, which is a common antidebugging trick.
desktopobjscan	By request	Links desktop objects with Window stations and owning processes; this technique can expose malware that uses hidden windows.
driverirp	Public	Detects attempts to hook drivers by printing input–output request packet (IRP) function addresses.
idt	Public	Detects attempts to hook the interrupt descriptor table (IDT)
impscan	By request	Scans unpacked user mode processes and kernel drivers for imported API functions; this can help rebuild dumped code for static analysis.
ldr_modules	Public	Detects unlinked (hidden) DLLs in user mode processes.
malfind2	Public	Detects hidden and injected code.
notify_routines	By request	Detects thread, process, and image load notify routines—a technique used by rootkits such as Mebroot, FFsearcher, Blackenergy, and Rustock.
orphan_threads	Public	Detects hidden kernel threads and carves out the rootkit code.
svcscan	By request	Detects hidden services by scanning for SERVICE_RECORD structures.
windowstations	By request	Scans for Window station objects that can expose rogue terminal services and Remote Desktop Protocol (RDP) sessions.

technical assets configured with a high level of logging, provide valuable attack data for analysis, but security analysts also periodically use them as decoys that deliberately contain known vulnerabilities. When deployed as a distinct network, known as a honeynet, a firewall is specially configured to collect and contain network traffic. The placement and configuration of a honeypot largely determine its success, and because malicious activity is likely to occur, it is crucial that it be isolated from true IT assets and legitimate traffic.

Network and information security relies on in-depth defenses to limit unauthorized access and dissemination of sensitive information. These in-depth defenses provide a hardened posture but give no insight on vulnerabilities and other weaknesses exploited by attackers in the wild. This lack of visibility requires a reactive approach to a security incident, which is a norm within the IT security field as a whole. An ideal approach involves proactive measures using knowledge and information of upcoming vulnerabilities, malicious code, and attackers to build up defenses prior to an incident. One method of obtaining the necessary data to create safeguards requires sacrificing a specially configured system, known as a *honeypot*, to lure in malicious activity for analysis.

A honeypot is an information system resource whose value lies in unauthorized or illicit use of that resource.[8] A honeypot is a concept that capitalizes on the isolation of a resource and subsequent activity that interacts with the resource. Designed to resemble an interesting target to attack, probe, exploit, or compromise and configure with a high level of logging, honeypots attract attackers and malicious code to capture their activity for analysis. Honeypots thrive in isolated environments because they have no production value or business purpose and all activity observed is suspicious. Placement of these resources is important to minimize the amount of legitimate or unintentional traffic.

Honeypots are beneficial if properly deployed and maintained. The fact that honeypots resemble an easy target may act as a decoy to keep attackers from attacking production systems. Honeypots also provide data and insight on who attacks those honeypots and on attack strategies used during exploitation. If properly handled and left untainted while gathering, these data can provide evidence in an incident investigation in the form of digital fingerprints. Another benefit results in the building of safeguards in production security defenses to minimize the threat of attacks and targets based on the information gathered from the honeypot.

Honeypots fit into two different classifications based on the level of system interaction available to the attacker. *Low-interaction honeypots* emulate vulnerable services and applications to entice inbound exploit attempts from attackers. Emulation occurs by mimicking real network responses to inbound connections allowing an attack to progress to completion. The attacks do not compromise the honeypot because the honeypot itself is not vulnerable; rather, it follows along by emulating vulnerabilities. Logs of the activity capture the exploit attempt, and postattack analysis provides information to protect other production devices from falling victim to the attack. The second type of honeypots, known as *high-interaction honeypots*, utilize actual services and vulnerabilities to attract inbound attacks. The use of real services provides detailed information on the steps involved in exploitation and the postcompromise activity. This type of honeypot requires close and constant observation because the system is likely to fall victim to compromise. High-interaction honeypots also need extra security measures to contain subsequent attacks or malicious code propagation.

The two types of honeypots have strengths and weaknesses that need consideration before deployment. Emulation keeps the low-interaction honeypots relatively safe from compromise and lowers the amount of effort required for maintenance. Low-interaction honeypots have limited data logged, reducing analysis time; however, emulation requires prerequisite knowledge of vulnerabilities and cannot capture attacks on unknown vulnerabilities. A drawback to a lack of compromise is a limited amount of data available after an attack is attempted. High-interaction honeypots provide more information on malicious activity than low-interaction honeypots but require more resources to analyze and maintain. Creating and maintaining a high-interaction honeypot consume significant resources because they typically involve customized technologies such as firewalls, intrusion detection systems (IDSs), and virtual machines, which need frequent rebuilds after compromise. Honeypot analysis consumes large quantities of time and resources, as this type of honeypot logs the full attack and subsequent activity, not just the initial inbound connection; however, after an attack, the system will remain compromised and will require cleansing. A heightened level of risk is involved with a compromised honeypot because the attacker can launch further attacks on other systems. Investigations would show the honeypot as the source of the attack, which raises legal concerns.

Many commercial and open-source honeypot solutions are available and vary in intended use. Typically, honeypots act as a decoy to lead attacks away from production systems. Specter, a commercial honeypot, is an example of a low-interaction honeypot that advertises vulnerabilities and acts as a decoy and data collection solution.[9] A collection of honeypots used to simulate a network of systems, known as a *honeynet*, requires a system called a *honeywall* to capture and analyze the data on the network and contain the risks presented

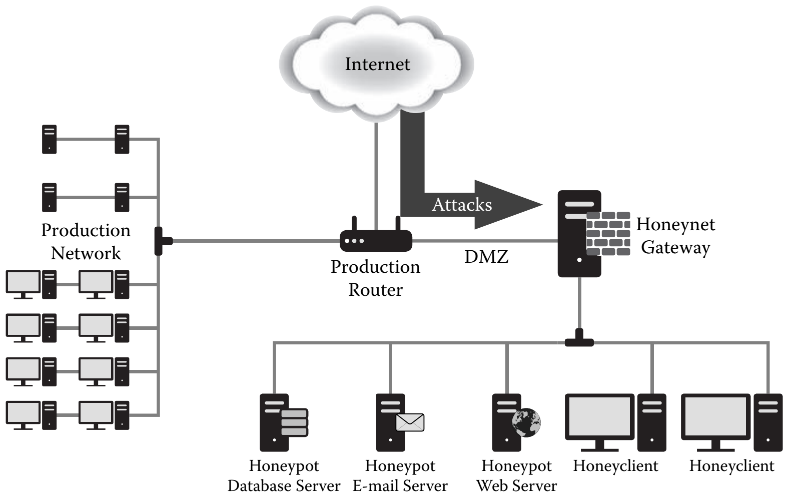


Exhibit 5-3 A honeynet infrastructure.

by these high-interaction honeypots.[10] Exhibit 5-3 shows a honeynet's infrastructure, including the honeynet gateway residing in the demilitarized zone (DMZ) to expose its vulnerable infrastructure for inbound attacks.

Honeypots can also collect malicious code. Applications like Nepenthes also advertise vulnerabilities and capture and download malicious code or analyze shellcode resulting from exploitation. Nepenthes also includes submission modules to submit captured malicious code to a number of other servers including Norman's SandBox for analysis.[11] Honeypots also have the ability to track spam e-mail. Honeyd, a lightweight honeypot daemon configured to simulate a mail relay or open proxy, captures e-mail spam for tracking and spam filter creation.[12] Honeypots are not limited to servers capturing information regarding malicious activity. Capture-HPC[13] and MITRE's Honeyclient[14] are client-based honeypots that act as clients interacting with malicious servers to download malicious code and log changes made to the system. For more honeypot-related applications, the Honeynet Project offers a list of projects available for download.[15]

The legality of honeypot deployment is under constant debate and generally involves discussions on entrapment, privacy, and liability. Entrapment occurs when a law enforcement officer or agent induces a person to commit a crime that the person would be unlikely to commit. Entrapment does not apply to honeypots, as they do not induce the attacker to break into the system. In addition, entrapment is a defense to a crime, meaning one cannot sue another for this reason. Privacy raises a big issue with honeypots in regard to logging information on attackers. The Federal Wiretapping Act and the Electronic Communication Privacy Acts, among others, come into play when logging an attacker's activity. Typically, a logon banner stating that the server monitors and logs activity proves enough for to waive attackers' privacy laws. Liability in the event of a compromised honeypot used to attack another system is a major legal concern.[16] Consultation with a legal team can reduce the occurrence and impact of these legal issues before deploying a honeypot.

Honeypots lure attackers into performing malicious actions within their systems for information-gathering purposes; however, seasoned attackers can detect a honeypot. An attacker who knows that he or she is in a honeypot will not perform the malicious activity that the honeypot intends to catch, or he or she avoids the honeypot's ability to log activity before he or she performs the activity. Honeypot detection techniques range in complexity and depend on the honeypot technology in use. Emulated services used in low-interaction honeypots may not perform exactly as the real service does. An attacker can use comparative analysis between a real service and an emulated service to detect a honeypot. The detection methods for high-interaction honeypots include virtualization checks, network traffic modification, and latency checks. Most high-interaction honeypots run in a virtualized environment, which allows attackers and malicious code to check for strings and environmental settings to fingerprint a honeypot. An example of an environmental setting used in a virtualized system is registry values added for virtual devices required for the guest OS to utilize hardware. Honeynets require data control to limit outbound attacks, and analyzing outbound network traffic for modification or connection blocking is an indication of a honeynet. For example, honeynets typically employ snort inline to scrub outbound attacks, and a popular test attempts to run /bin/sh on an external host to see if a snort modifies the packet or drops the connection. High latency from communication tests can provide an indication of a honeypot using logging modules. Modules used for logging typically log all activity performed on the system; therefore, running instructions that increase the load on the system results in network latency. A common example of this detection method uses the *dd* command to copy an endless amount of data to /dev/null, which produces overwhelming amounts of data for the honeypot to log. The *ping* command can check the network latency during the heavy load that the *dd* command invokes.

Honeypots provide security professionals and network administrators with information on state-of-the-art attack techniques seen in the wild. Using this information to implement security safeguards strengthens a network's posture and reduces exposure to threats. Proactive responses to threats and attacks are possible with obtained information, which makes honeypots valuable tools to help survive the malicious nature of the Internet.

5.3 Malicious Code Naming

This section clarifies the malicious code–naming conventions within the industry, which can be confusing and difficult to reference. The differences in procedures used by antivirus tools and those used by analysts are at the heart of the problem. This section also discusses in depth this and other challenges to naming malicious code consistently.

Many security researchers and administrators confuse viruses with one another because of the way that antivirus companies name or refer to them. There are several reasons for the confusion, and a few organizations are trying to improve the currently dismal state of malicious code naming in the antivirus industry. Some basic tools and advice can help administrators determine whether they, in fact, are dealing with a generic virus name or one that accurately describes the entire malicious code family. Administrators might expect that antivirus detection names would be a good metric to determine the malicious code family; however, this is often not an accurate or reliable measurement. The media and researchers often tend to use different names, sometimes even within the malicious code itself, while other professionals may alter or hide the true name of the virus for their own reasons. Many factors make analysts the best sources for determining the name of a malicious code over any other currently available automatic solution.

iDefense analysts usually assign a malicious file a new name when nothing previously describes it or when it provides a more valuable

ANTI-VIRUS VENDOR	VIRUS NAME
AntiVir	DR/Delphi.Gen
Avast	Win32:Trojan-gen {Other}
AVG	VB.FTL
BitDefender	Trojan.AgentMB.CSDN4118442
ClamAV	Trojan.Downloader-35380
DrWeb	Trojan.MulDrop.origin
F-Secure	Suspicious:W32/Malware!Gemini
GData	Trojan.AgentMB.CSDN4118442
Kaspersky	Trojan.Win32.VB.ieq
McAfee+Artemis	Generic!Artemis
Microsoft	VirTool:Win32/CeeInject.gen!J
NOD32	probably a variant of Win32/Injector.DV
Panda	Suspicious file
SecureWeb-Gateway	Trojan.Dropper.Delphi.Gen
Sophos	Sus/Dropper-R
TrendMicro	WORM_SOBIG.GEN
VBA32	Trojan.Win32.VB.ieq

Exhibit 5-4 An antivirus scan of a typical banking Trojan.

reference point. Other organizations may have different policies about renaming viruses when they create detections for them because it prevents the revealing of new hacking tools and techniques to attackers. Similarly, attackers can insert fake authors, tool names, or other details to confuse analysts. To determine if a malicious program already has a malicious code name, analysts can use online virus-scanning services such as Virustotal or av-test.org. As an example, Exhibit 5-4 displays the results of such a scan showing the detection for a typical banking Trojan.[17]

In these Virustotal results, seventeen out of thirty-eight (44.74 percent) antivirus engines detect this file as malicious. Despite the average rate of detection from various antivirus companies, none of them indicates either the true name of the virus family or the fact that its purpose is to target banks. Naming a malicious file using this technique is highly likely to be wrong or too generic to convey any useful information; however, it is very easy and quick for anyone to do. It does not require any understanding of the code or its purpose. Despite weaknesses in this technique, administrators may be able to use detection names to research the threat further and potentially identify a similar threat and malicious behavior. This is not completely reliable because signatures often detect generic threats, packed or hidden code, and behavior rather than malicious code names that convey the most information. As an example, an antivirus product may name two different files the same if they download additional functiality (downloaders) or if the antivirus engine detects the same type of packer used against two completely different threats. Using automatic programs such as antivirus programs is a useful preliminary step to help determine the risk of a malicious file, but such programs are not as accurate or reliable as reverse engineering or behavioral analysis. The signatures are a technique for analysts to write and improve detection of threats but may not require the engine or analyst to analyze the purpose of the code. Antivirus scanning products have a one-to-one relationship between a signature and a detection name. This helps them track alerts from customers but often does little to help customers understand the nature of the threat, especially with generic signatures.

There are many reasons for the differences between the naming that researchers and reverse engineers use compared to that of automatic antivirus scanning products. Analysts do not suffer from many of the same problems because they are able to collect and inspect many different things that are not available to antivirus products. Observing network traffic, modified or created files, memory contents, and other information from reverse engineering the binary or the propagation technique allows analysts to more accurately identify and name malicious files; however, such naming is still imperfect. The detection of the banking Trojan, mentioned above, carries just as many different and unique names that researchers use to describe it. Researchers commonly refer to this particular code as *wsnpoem/ntos*, *Zeus/zbot*, and *PRG*. This type of naming depends upon awareness rather than an automated tool, and is therefore subject to human error or purposeful renaming when multiple researchers are to assign different names. Multiple names could be the result of private information. For instance, iDefense names new viruses when there is no public information available for them; however, when a public analysis or virus name becomes available, it becomes necessary to identify that both threats are, in fact, the same based upon behavior or other attributes.

Antivirus names use many different categories, and they all follow the same format:

```
Family_Name.Group_Name.Major_Variant.Minor_
Variant[:Modifier]
```
[18]

Although this format is common, many antivirus vendors have a lot of flexibility when they name a new virus, including the family, group, and variant names. Common types include *generic* or *heuristic* (*heur* for short) in their names. Administrators should understand the meaning of antivirus product naming that they use in their environments by referring to naming documentation from vendors.[19]

Other virus names may originate through an analyst's creativity or a virus's circumstances. For instance, the W32/Nuwar@mm threat actually originated from e-mails that initially spread using the attention-grabbing headline "Nuclear War." Researchers know this threat better as "Storm Worm" because it also spread using a different e-mail subject line, such as "230 dead as storm batters Europe." In fact, the community disputed the naming of this particular virus as a worm because it spreads using massive e-mail campaigns.

There is also some disagreement between different organizations on whether to depend upon attacker-supplied information to name viruses. Some researchers argue that hiding the names can protect the innocent, for instance if the attacker artificially inserts an enemy's name or website out of spite. Hiding real names or mutating them also attempts to hide origin. This prevents attackers from identifying new tools from new virus names. Mutating names can also help protect an innocent website or avoid giving out information that may allow new attackers to locate public tools. For example, Brian Krebs of the *Washington Post* documented one such incident related to the virus named Blackworm, Nyxem, My Wife, and Kama Sutra. The origins of each name are clear if you understand how the malicious code works, but it can often be difficult to determine that they are, in fact, equivalent threats (see Exhibit 5-5).

VIRUS NAME	EXPLANATION
Blackworm	Creates \WINDOWS\system32\About_Blackworm.C.txt
Nyxem	Target the New York Mercantile Exchange (Nymex)
My Wife/Kama Sutra	Subjects surrounding the e-mail attacks

Exhibit 5-5 Virus names and their explanations.

Different goals and perspectives influence the naming of viruses and may encourage researchers to invent new names even when an existing name is already available. According to Brian Krebs, *Nyxem* was derived by transposing the letters *m* and *x* in *Nymex*, a shorthand term for the New York Mercantile Exchange.[20]

Virus names are often cryptic on purpose because of a lack of verifiable information. Overlapping names can further confuse naming when an antivirus product assigns a well-known name to a new or unknown virus. For example, iDefense analyzed a Chir.B worm variant in 2007 that the Avast antivirus scanning engine determined was a much older threat called Nimda (or Win32:Nimda [Drp]). The reason for the alert is that the signature detected the behavior of the file-infecting worm functionality and assigned an older name that has the same behavior. The detection of new threats is commendable using older signatures, but it is clear that rule writers are not able to express themselves sufficiently to tell users what an alert actually means and whether it detects a behavior, a particular virus, or something else. Analysts cannot predict the future evolution of viruses; therefore, it is difficult to choose names reliably that will not detect multiple threats.

The Common Malware Enumerations (CME) list, while encouraging in its early days, never reached a critical mass and was not sustainable with the large volume of new viruses and limited resources. It provided a catalog of thirty-nine total different threats over several years with naming from various antivirus vendor names and virus descriptions. Although using this as a tool to investigate potential virus names and families can be useful to administrators, it has been largely neglected during the last two years according to the CME website.[21] Groups like the Computer Antivirus Researchers Organization (CARO) have experienced similar problems when attempting to standardize the naming of viruses.[22]

5.3.1 Concluding Comments

Administrators should attempt to understand abbreviations and standard naming conventions for incidents because it may help them look for certain behavior or ask questions; however, dependence on virus naming is not reliable or capable of conveying enough information to be very useful. Analysts and reverse engineers are still the best sources for identifying virus families because of the high variation of names assigned to viruses. Extensive research, including reverse engineering and behavioral analysis, is usually necessary to determine how to name a threat accurately.

5.4 Automated Malicious Code Analysis Systems

The massive volume of distinct pieces of malicious code in existence exceeds the capacity of human analysts. Fortunately, researchers can automate much of the initial analysis. This automation allows much greater efficiency and prioritization of analysis of malicious code samples.

With attackers producing tens of thousands of new pieces of malicious code every day,[23] it is impossible to analyze each sample by hand. Behavioral analysis, the process of running an executable in a safe environment and monitoring its behavior, is one way to determine what malicious code does. Automated malicious code analysis systems (AMASs) perform this process quickly and efficiently to produce a report that a human analyst can use to determine what actions the malicious code took. In this section we explore the advantages and disadvantages of different techniques used by AMASs to analyze malicious code.

In recent years, researchers have built many AMASs that differ in capability and analysis techniques but all operate under the same principle. To be effective, malicious code has to perform some action on the infected system, and monitoring the malicious code's behavior is a useful way to determine the malicious code's functionality. Behavioral analysis cannot determine everything that malicious code is capable of, but it can tell you what malicious code will do under certain circumstances. There are two main techniques to analyze the behavior of malicious code:

1. *Passive analysis*: Record the state of the system before and after the infection. Then, compare these states to determine what changed.
2. *Active analysis*: Actively monitor and record malicious code actions during execution.

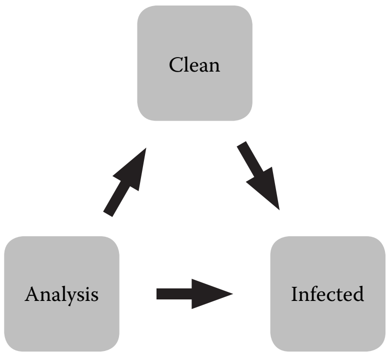


Exhibit 5-6 An automated malicious code analysis cycle.

5.4.1 Passive Analysis

Passive analysis is the hands-off approach to behavioral malicious code analysis. All it requires is a computer to infect, some way to capture the state of that computer, and a way to restore the system to its original state. Passive analysis systems work in the three-stage cycle shown in Exhibit 5-6. First, someone installs the operating system and any necessary applications on a computer, recording the "clean" state. The recorded information includes any features of the system that malicious code might alter, such as the file system and Windows registry.

Second, the malicious code in question is executed on the system for a period of time. The amount of time depends on how quickly the analysis must be performed. Two- to three-minute runtimes are common, as this is normally a sufficient amount of time for the malicious code to complete its initial installation.

After the malicious code infects the system, it must be shut down before an external system analyzes its disk and memory to record the new "infected" state. An external computer may be used to record the infected system's state to avoid any interference from the malicious code. Malicious code often hides files and processes from the user using rootkits, but an external system (such as a virtual machine host or a system working from a copy of the infected disk) is not susceptible to this interference.

During the analysis stage, the external system compares the infected state to the clean state already recorded. AMASs can make comparisons between any features of the system that have a state. Common analysis features include the following:

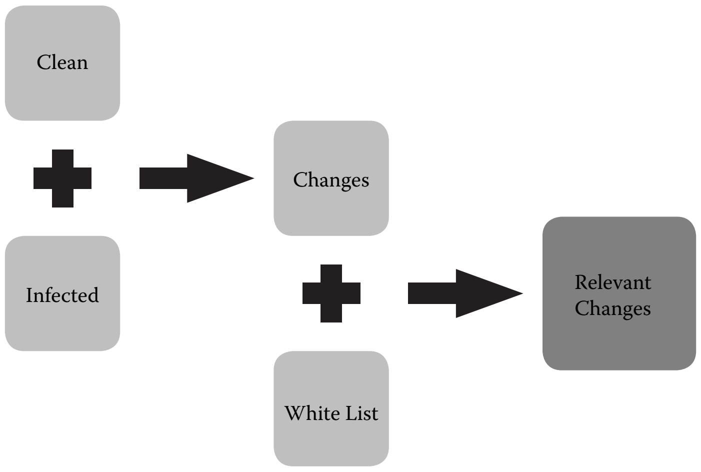


Exhibit 5-7 A passive analysis comparison process.

- File system
- Windows Registry content
- Running processes
- Listening ports
- Memory contents

The comparison between the clean and infected states is where the passive analysis system shines. The analysis typically consists of two stages (see Exhibit 5-7). In the first stage, it compares the clean and infected states and creates a list of all changes in the monitored features. While it may seem that this list of changes is sufficient, it is important to remember that while the malicious code was infecting the system, Windows was also performing thousands of tiny tasks that might also make changes to the file system. This is especially true if anyone has rebooted the system since the original clean state was recorded, as might be the case in analysis systems that use physical hardware. To filter out these nonmalicious changes, the system uses a second stage (middle section) to remove all entries that are included in a predefined *white list*. The result is a report that contains all changes on the system that are relevant to the malicious code analysis.

In addition to static information about the malicious code (file name, size, and MD5), the resulting report might contain the following information:

New files:

- C:\WINDOWS\system32\lowsec\user.ds
- C:\WINDOWS\system32\lowsec\local.ds
- C:\WINDOWS\system32\sdra64.exe

Registry modifications:

- Key: HKLM\software\Microsoft\Windows NT\CurrentVersion\Winlogon
- Old Value: “Userinit”=“C:\\WINDOWS\\system32\\userinit.exe”
- New Value: “Userinit”=“C:\\WINDOWS\\system32\\userinit.exe, C:\\WINDOWS\\system32\\sdra64.exe”

This information shows us that not only did the malicious code create three new files but also it altered the Windows Registry so that the file sdra64.exe is run when the user logs on to the system.

Passive analysis systems also frequently include network monitoring, as long as the monitoring system occurs outside of the infected system. Network traffic is a key component to many AMASs because it includes any external communication the malicious code might make and reveals the source of the malicious code’s command-and-control (C&C) server if one exists. In the mentioned example report, analysis of the network traffic revealed that the URL visited was http://index683.com/varya/terms.php.

Knowing that the malicious code visits this particular website is very valuable. Security personnel can search proxy logs for any systems that visited this site to pinpoint infected systems. Blocking access to this URL will also help prevent the malicious code from conducting its malicious activity.

Malicious code cannot typically detect a passive analysis system because the system does not interfere with its operation. Malicious code can make passive systems ineffective by taking advantage of the system’s analysis timeout. If the system only allows the malicious code to run for three minutes before recording the infected state, the malicious code could simply sleep for four minutes before taking any action.

While passive analysis is simple, it cannot tell the malicious code’s entire story. For instance, if the malicious code creates a temporary file while installing its components and then deletes that file before the system captures the infected state, the analysis report will not include this evidence. Passive monitoring also fails to capture the timeline of the infection. The sample report above shows that the malicious code creates three files, but it does not show the order in which the malicious code created them. It is possible that the malicious code created sdra64.exe first, and that executable created the additional files. To capture this information, the system must actively monitor the malicious code.

5.4.2 Active Analysis

Unlike passive systems, active analysis AMASs install software on the soon-to-be-infected system that monitors the malicious code and keeps a log of its activity. This process creates a much more complete report that can show the order in which the malicious code made changes to the system during the infection and can record which specific process took each action. Some may classify many modern Trojans as *downloaders*, as their primary functionality is to download and execute a secondary payload. Active analysis systems can differentiate between files and registry keys created by the downloader and those created by the new file. This functionality is one way that active systems provide much more detail than a passive system ever could.

One way that active systems monitor malicious code activity is through a process known as *application program interface* (API) hooking. An API hook allows a program to intercept another application's request for a built-in Windows function, such as InternetOpenUrlW(), which applications use to make requests for Web pages. API hooking is a technique often used by rootkits because it allows malicious code to not only record what the user is doing but also alter the data sent to and returned by the API function. A rootkit might use this to hide the presence of particular files or ensure that the user does not stop any of its malicious code processes.

Active analysis systems can install their own rootkits that hook the APIs that the malicious code will use, allowing it to keep track of every API call the program makes. If malicious code can detect the AMAS processes, it could simply exit without taking any actions that would reveal its functionality. This is the primary disadvantage to active systems, but a well-written rootkit can hide its own processes to prevent the malicious code from detecting it and altering its behavior.

Active systems are not vulnerable to the same waiting technique that malicious code uses to fool passive systems. An active analysis rootkit can hook the sleep()function that malicious code uses to delay execution and then alter the amount of time the malicious code sleeps to just 1 millisecond.

Active analysis systems also work in a cycle between clean and infected states, but do not require a comparison of the clean and infected states to perform their analysis. After the malicious code completes execution or runs for the maximum time allowed, the system records the activity in a report and begins restoring the system to the clean state.

Another form of active analysis involves using an emulator rather than infecting a traditional operating system (OS). The most prominent emulation-based analysis system is the Norman SandBox.[24] Instead of installing a rootkit and hooking the Windows APIs, Norman created software that emulates the Windows OS. When malicious code running in the Norman SandBox makes a call to the sleep()function, it actually calls a Norman function that acts just like the Windows sleep()function. Malicious code can detect emulated systems if they do not perfectly mimic the operating system's API, and malicious code authors frequently attempt to evade these systems. The main advantage of emulated systems is speed. Emulated systems do not require swapping between a clean and infected state and can run the malicious code faster than a standard OS because they do not need to provide full functionality of each API; they need merely to emulate the OS in a convincing way. For any organization that processes thousands of samples each day, speed is a key factor in rating an AMAS.

5.4.3 Physical or Virtual Machines

For nonemulated AMASs, both passive and active, analysis time is spent in two primary categories. First, time is spent allowing the malicious code to execute. If the runtime is too short, the analysis might miss a critical step taken by the malicious code, but the more time allotted for the malicious code to run, the longer the system takes to generate a report. The second major source of analysis time is restoring the infected system to a clean state. This must be done to prepare the system for the next analysis and makes up a significant portion of the analysis time.

Virtualization systems like VMWare and VirtualBox have many features that make them an excellent choice when developing an AMAS. These programs allow a user to run one or many virtual computer(s) on top of another OS. Researchers use these systems to run many analysis instances on a single physical computer, saving time, power, and money. Virtual machines (VM) also have the ability to store a clean "snapshot" of the operating system. After the analysis is complete, restoring the system to the clean snapshot typically takes less than 30 seconds; however, as with active analysis systems, it is possible for malicious code to detect that it is running in a VM and alter its execution path to trick the system into producing an inaccurate report. One recent example of VM-aware malicious code is Conficker, which did not execute in VMs in order to increase analysis difficulty.

Physical machines are not as simple to restore compared to their virtual counterparts, but there are multiple options available. One possible solution is Faronics DeepFreeze.[25] DeepFreeze is a Windows program that allows administrators to revert a system to a clean state each time it reboots. Internet users at universities and Internet cafes, where many users access the same pool of computers, commonly use DeepFreeze. iDefense tested DeepFreeze for use in one sandbox and found that it was not sufficient to prevent malicious code from altering the system. Software solutions are not reliable for this purpose because malicious code can disable them or use methods to write to the disk that the software does not monitor.

CorePROTECT makes a hardware product named CoreRESTORE that acts as an interface between a computer's integrated drive electronics or advanced technology attachment (IDE/ATA) controller and hard drive (Exhibit 5-8).[26] CoreRESTORE prevents the system from making any changes to the disk but returns data as though someone already altered the disk. This solution is effective but is only available for systems that use IDE/ATE interfaces. A third solution is to save a complete copy of the system's hard drive in a clean state and write this copy to the system's disk each time a restoration is necessary. Joe Stewart of SecureWorks first introduced this method in The Reusable Unknown Malware Analysis Network

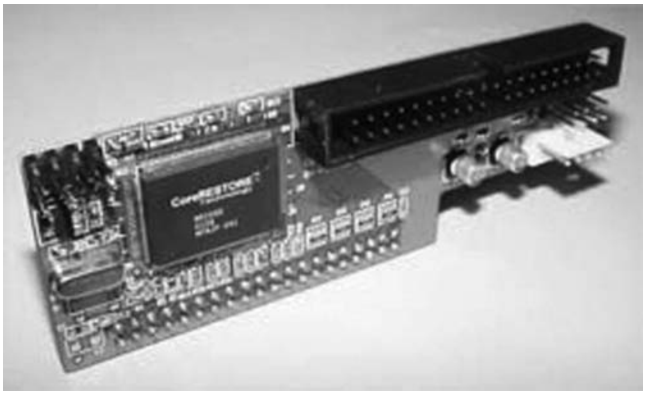


Exhibit 5-8 A CoreRESTORE ATA/IDE bridge.

(TRUMAN) system, and iDefense currently uses this method in its Malcode Rapid Report Service (ROMAN). This method takes two to three minutes per analysis but is undetectable by malicious code and ensures that each analysis begins with a known clean image.

Pure passive and active analysis systems are common, but there is no reason that a single system cannot employ techniques from both categories. iDefense is currently developing a new AMAS known internally as Automal, which uses a combination of passive and active analysis using a custom rootkit. The primary functionality of Automal is based on memory forensics using custom plug-ins for the Volatility framework.[27] Memory forensics is relatively new in the world of AMAS but allows systems to discover critical information about data and processes that are hidden from tools running on an active system and show no evidence in features typically monitored by passive systems. Automal runs Volatility on a snapshot of the infected system's memory when the system is offline, which prevents the malicious code from detecting it or changing tactics based on its use. AMASs are valuable tools to anyone who regularly analyzes malicious code, not just to those who process thousands of samples per day. Many organizations do not have the resources or need to develop their own AMASs. Fortunately, many are available free online. Table 5-4 shows some of the most popular AMASs currently available.

Each system uses a different analysis mechanism and may return different results. Submitting files to multiple systems can be beneficial since the combination of the resulting reports may be more

Table 5-4 Current Popular AMASs

SYSTEM NAME	URL
Norman Sandbox	http://www.norman.com/security_center/security_tools/submit_file/en-us
Sunbelt CWSandbox	http://www.sunbeltsoftware.com/Developer/Sunbelt-CWSandbox/
Anubis	http://anubis.iseclab.org/index.php
ThreatExpert	http://www.threatexpert.com
TRUMAN	http://www.secureworks.com/research/tools/truman.html
Comodo	http://camas.comodo.com
BitBlaze	https://aerie.cs.berkeley.edu
JoeBox	http://www.joebox.org

complete than what a single system can produce. Using AMAS is an excellent first step during any malicious code investigation, as a fully automated analysis can be performed quickly and requires little human interaction.

5.5 Intrusion Detection Systems

Network security encompasses any safeguards deployed to increase the safety of interconnected systems and the information that traverses the network between these systems. Connecting computers allows for communication and the exchange of information, but also exposes these computers to threats from remote locations. This exposure to external threats needs a monitoring and detection solution to ensure the safety of interconnected systems. In this section, iDefense describes a network detection solution called an intrusion detection system (IDS).

Every day, new vulnerabilities and malicious code threaten systems on networks. The constant update of threats requires strenuous patching schedules and antivirus updates. Patching and antivirus updates in an enterprise environment take time, which prolongs the period in which devices are vulnerable. In the event that no patch exists for a given vulnerability (such a case is known as a *zero-day vulnerability*), devices are vulnerable for an even longer period while the vendor develops a patch. There is a need for systems to detect vulnerabilities and malicious code activity during these vulnerable periods. An IDS can satisfy this need very quickly, as these devices can receive one update and detect malicious activity across an entire network of computers.

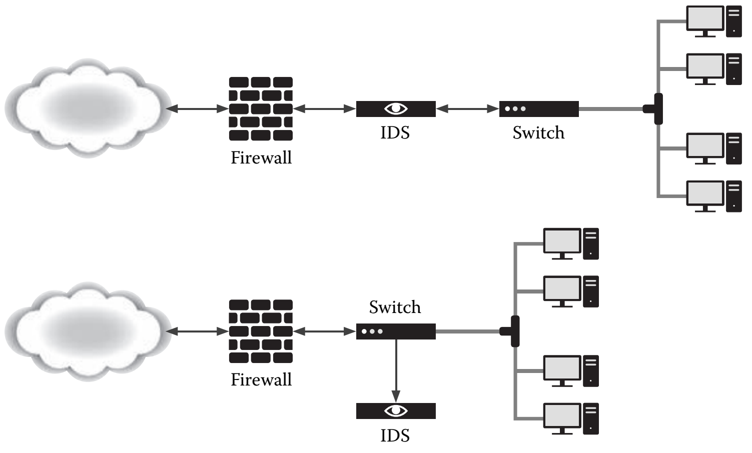


Exhibit 5-9 Out-of-line and inline topologies.

An IDS is a device that monitors network traffic for malicious activity. IDS devices, referred to as *sensors*, detect malicious activity by searching through traffic that traverses a network. The IDS sensor requires access to network packets, which is possible through two different implementations called *out of line* and *inline*. Exhibit 5-9 shows the difference in network topologies between out-of-line and inline sensors.

Out-of-line sensors connect to a switched port analyzer (SPAN), an action also known as *monitoring*, *port mirroring*, or a *network tap*. A SPAN port is a port on a network device, such as a switch or firewall, that receives a duplicate feed of the real-time traffic for monitoring purposes. A network tap operates in a similar manner; however, these are standalone devices that send and receive traffic between two ports and have a third port that receives a copy of this traffic for monitoring purposes. Out-of-line sensors connected to a SPAN either port or tap monitor traffic and produce alerts in response to malicious activity.

Inline sensors differ from out-of-line sensors in that they physically sit in the path of the network traffic. Network traffic travels from its source through the inline device to its destination. The inline sensor checks the traffic sent through it for malicious activity to produce alerts or block the malicious activity. Inline sensors configured to block malicious traffic, known as *intrusion prevention systems* (IPSs), have a greater impact on reducing the occurrence of malicious activity on a network.

Both types of sensors use rules, also known as *signatures*, to detect malicious activity. IDS sensors rely on these signatures to detect malicious activity; therefore, the overall effectiveness of an IDS sensor mostly depends on the caliber of the signatures. Most IDS vendors have different rule structures or languages, but such rules generally use content matching and anomalies to detect events.

Content-matching rules use specific pattern matches or regular expressions to search network traffic for specific strings or values associated with malicious traffic. These rules are very specific and require prior knowledge of the particular malicious content within network activity. The use of regular expressions provides flexibility to a signature by allowing it to search for multiple variations of a string. For example, the following shows a content match and regular expression that search network activity for HTTP GET requests related to a client infection. The content match is static and straightforward, but the regular expression enhances the effectiveness and accuracy by searching for multiple different actions.

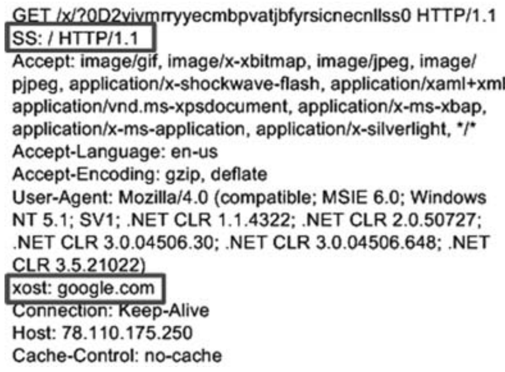
```
Example HTTP request:
GET /controller.php?action=bot&entity_list=&uid=1&firs
t=1&guid=412784631&rnd=94

Content Match
"GET /controller.php?action="

Regular Expression
"/GET\s/controller.php?action=(bot|loader|report)/"
```

Sensors also detect malicious activity based on anomalous network traffic. These anomalies include protocol-specific anomalies and traffic thresholds. Network protocols abide by standards, and abnormalities to these standards are an indication of suspicious activity. Signature authors capitalize on these protocol abnormalities to detect malicious activity. For example, Exhibit 5-10 shows such a protocol anomaly witnessed within the HTTP header of a GET request generated by an infected client. The malicious code author added the fields *SS* and *xost* to the header, allowing for easy detection by an IDS signature as they are not part of the HTTP protocol.

Traffic thresholds detect anomalous increases in traffic compared to a baseline amount of traffic. This approach requires a baseline figure that accurately represents the normal amount of traffic expected to



```
GET /x/?0D2yivmrryyecmbpvatjbfyrsicnecnllss0 HTTP/1.1
SS: / HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/
pjpeg, application/x-shockwave-flash, application/xaml+xml,
application/vnd.ms-xpsdocument, application/x-ms-xbap,
application/x-ms-application, application/x-silverlight, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows
NT 5.1; SV1; .NET CLR 1.1.4322; .NET CLR 2.0.50727;
.NET CLR 3.0.04506.30; .NET CLR 3.0.04506.648; .NET
CLR 3.5.21022)
xost: google.com
Connection: Keep-Alive
Host: 78.110.175.250
Cache-Control: no-cache
```

Exhibit 5-10 Abnormal fields within an HTTP header.

observe an increase. The baseline figure needs constant adjustments to reflect legitimate increases and decreases in traffic patterns. Without these adjustments, the IDS will generate many alerts on legitimate traffic and waste investigative resources. Threshold-based detection does not often detect a specific threat but provides a heuristic approach to malicious activity detection. These events require investigation to determine the specific issue, as they are prone to trigger on nonmalicious traffic.

By name, IDS suggests that such systems simply detect inbound attempts to gain entry to a device; in reality, they have the ability to detect much more. An IDS device can detect any type of malicious activity that traverses a network based on the rules used for detection, with some exceptions described later in this section. The success of an IDS device in detecting a particular event depends on the accuracy and flexibility of the signatures within its rule set.

A *rule set* is a list of signatures that the IDS device uses to detect malicious activity. IDS vendors supply a rule set for their products, and many allow the creation of custom signatures. The signatures within these sets can detect inbound attacks on servers and clients, malicious code infections, and propagation.

An IDS device has the ability to detect inbound attacks on a server or client from specially crafted signatures. To detect these attacks, the signature author needs prior knowledge of the attack or the vulnerability to match its network activity. Equipped with a signature for the attack or vulnerability, the IDS sensor can detect the activity and trigger an alert for the possible compromise on the destination. The

IDS, however, is unable to determine if the end system was vulnerable to the detected attack. An investigation is pivotal to determine if the attack was successful.

Rules can also detect worm propagation via content matches or anomalies. The content match approach requires prior knowledge of the network activity the worm generates when it attempts to spread to other systems. A signature match provides the source of the worm propagation, which is an infected system that needs remediation. An investigation of the destination in this event will determine if the worm successfully spread to the system.

An anomaly-based rule can provide worm detection in a heuristic manner. By using thresholds, a signature can trigger an alert on an increase in traffic over a worm-able service, such as MS-RPC, NetBIOS, or SUNRPC, to investigate a possible worm outbreak. For example, if an increase in traffic occurs from one system over Microsoft NetBIOS port 139, it could be a worm attempting to propagate to other systems. This alert, however, could also be the result of a legitimate file transfer to a shared resource. This shows the need for investigation to determine the cause for the anomalous increase in traffic.

IDS sensors can be effective at detecting Trojans installed on compromised machines. Trojans communicate with their command-and-control (C&C) servers to download updated configuration files and binaries, to receive commands to run on the infected systems, or to drop stolen data. The network activity generated by this communication usually uses a common protocol, such as HTTP, to avoid rejection from a firewall. Content-matching rules specifically created for the C&C communication can accurately detect Trojan infections. The example HTTP request discussed previously in this section was activity from a Trojan, and using the content match or regular expression in an IDS signature would successfully detect infected machines. Occasionally, malicious code authors omit fields or include additional fields to standard protocols within their code, which generates anomalous traffic, as seen in Exhibit 5-10. This allows an anomaly-based IDS signature to detect the C&C traffic easily by searching for these protocol abnormities.

IDS devices detect a variety of threats to a network, but they do have issues that limit their effectiveness. IDS evasion is a concept that encompasses all techniques used to avoid detection during malicious activity. Varieties of techniques are available, but the most common evasion methods include obfuscation, encryption, compression, and traffic fragmentation.

Obfuscation, encryption, and compression can evade detection from an IDS. IDS signatures searching for content as the result of malicious activity have difficulty matching if the known patterns change. Although obfuscation, encryption, and data compression are different in functionality and purpose, all three change the representation of data transmitted over the network. Obfuscation of data and exploit code evades detection through structural changes while retaining its original functionality through encoding, concatenation, and obscure variable and function names.

Encryption of data or the network traffic itself can evade detection from an IDS. An IDS signature can detect malicious activity within unencrypted channels by searching for malicious content within cleartext data sent over the network; however, an IDS has difficulty detecting malicious activity within encrypted communications because it does not have the key to decrypt the cipher text into cleartext data.

Compression changes the representation of data by passing it through an algorithm to reduce the size of the data. Compressing information is common for communication, as it requires less network bandwidth to transmit such information from one device to another. Evading the detection occurs when the sender compresses the data using a compression algorithm and sends the compressed data over the network to the destination. The destination uses a decompression algorithm to view the original data sent by the source. The IDS device sees the communication between the source and the destination but inspects the compressed data, which does not resemble the original data.

Traffic fragmentation and reassembly can also evade IDS. Malicious activity split into multiple different packets and sent from the source to the destination requires the IDS to reassemble the fragmented packets before inspecting the traffic.[28] For example, an attacker can spread the transmission of the attack's payload across fifty packets. To detect the attack payload, the IDS sensor has to track and reassemble the fifty packets in memory and then scan the reassembled payload using the rule set. Many fragmentation techniques are available to further complicate IDS evasion, such as fragment overlapping, overwriting, and timeouts, but such techniques do not fit inside the scope of this book.

In addition to IDS evasion techniques, the network environment that the IDS sensor monitors can affect the sensor's ability to detect malicious activity. Placement of the IDS sensor is key to monitoring the appropriate traffic. Overlooking sensor placement leads to visibility issues, as the sensor will not monitor the correct traffic.

Placement in high-traffic areas can severely affect the performance of the IDS sensor. Sensors in high-traffic environments require a great deal of hardware to perform packet inspections. Packet inspections become more resource intensive as the amount of traffic increases; if the sensor does not have enough resources, it will fail to detect the malicious traffic. This results in the IDS not creating an alert about the malicious activity.

The rule set used by the IDS sensor also affects the sensor's detection performance. To increase performance, each IDS vendor uses different rule set optimization techniques. Despite the optimization techniques used, the sensor checks all traffic monitored for signature matches. Checking traffic with a smaller set of rules will result in faster performance but fewer rules with which to detect malicious content. Larger rule sets will perform slower than a smaller set but have more rules for detecting malicious activity. This shows the need for compromise between speed and threat coverage.

Threat coverage shows the need for another compromise. An overflow of alerts will dilute critical alerts and valuable information with low-priority alerts and useless data. This dilution caused by excess noise makes triaging alert investigation difficult. The rule set for a sensor needs constant attention and custom tuning to reduce the number of alerts about legitimate traffic.

The last consideration for sensor placement involves inline devices. Inline devices physically sit between two network devices and have the ability to block malicious activity; however, legitimate traffic can also match signatures for malicious activity. This situation occurs often and results in the sensor blocking legitimate traffic. Another situation in which an IDS device can block traffic occurs when the sensors go offline or are overwhelmed with traffic. If the device does not fail to open in the event of system failure, then the device will block all traffic at its network interface. The inline device will also drop traffic if it exceeds its processing power.

Despite the issues facing IDSs, they are still beneficial to the security of a network. Proper consideration to the network environment that the IDS sensor will monitor is a must. An appropriate operating environment can reduce the issues previously discussed that plague a sensor's ability to detect malicious activity. Supplementing a proper network environment with continuous updates and tuning of the sensor's rule set will provide excellent coverage for a majority of malicious events.

IDS devices provide an invaluable stream of information to aid in security investigations and to improve the overall security of a network. IDS sensors can improve security by detecting a network's vulnerable areas and inbound attacks that can threaten the network. In cases involving an inline sensor, an IDS device can greatly improve network security by blocking malicious activity before it performs malice.

Luckily, the vast majority of inbound attempts to compromise systems do not use the IDS evasion techniques discussed in this section. Attackers overlooking evasion techniques allow IDS sensors to remain a viable monitoring solution. IDS can also detect compromised hosts based on network activity; however, the coverage for threats requires auditing to make sure the IDS detects malicious traffic. An IDS can provide a false sense of security if a signature exists for a threat but does not properly generate an alert in the event of its occurrence.

References

1. D. Brezinski, "RFC3227: Guidelines for Evidence Collection and Archiving," Advameg, Inc., February 2002, http://www.faqs.org/rfcs/rfc3227.html.
2. Matthieu Suiche, "Matthieu Suiche's Win32dd Tool," n.d., http://win32dd.msuiche.net/windd/. Acessed August 11, 2010.
3. Volatile Systems, "Volatility Website," 2006–2008, http://www.volatilesystems.com/default/volatility.
4. Google Code, "Volatility: An Advanced Memory Forensics Framework: Summary," 2010, http://code.google.com/p/volatility.
5. Google Code, "Volatility: An Advanced Memory Forensics Framework: Doc Files," 2010, http://code.google.com/p/volatility/wiki/DocFiles.