DeepSight™ Threat Management System
Threat Analysis

Userspace Windows Rootkits via Code Injection

Version 1: January 13, 2004, 00:30 GMT

Analyst: Jesse Gough

Executive Summary

During the past several months, the DeepSight Threat Analyst Team has encountered several pieces of malicious code that employ a technique commonly known as “DLL injection”. DLL injection is a technique that allows foreign code to be introduced into another process during runtime. This was particularly attractive at the time of discovery due to the additional stealth offered by allowing a malicious program (such as a backdoor) to be executed within the process space of a legitimate process.

While this method of infecting systems was first documented publicly several years ago[1], until recently, it has seen very little use among malware captured in the wild.

This document further explores this technique, detailing a way attackers may leverage this type of attack to deploy rootkits with only userspace privileges that provide a degree of control similar to that of a kernel-based rootkit.

Affected Systems

Microsoft Windows NT 4.0
Microsoft Windows 2000
Microsoft Windows XP
Microsoft Windows 2003

[1] Dll Injection was first discussed by Jeffrey Richter in his book titled, “Programming Applications for Microsoft Windows”

Introduction

After an attacker successfully compromises a system, they will typically deploy some kind of utility to ensure that access to the system may be regained at a later time. Once a backdoor has been implemented for this purpose, the compromised host is then used for malicious activity at the attacker's discretion. In order to maintain a presence on such systems, it is important that attackers deploy utilities to provide a certain degree of stealth to hide the unauthorized activity.

These utilities, known as "rootkits", have been in use for decades, and have evolved significantly during their existence. Originally designed for Unix platforms, rootkits in their most basic form involved replacing specific administrative executables with modified versions to produce false output that would withhold information regarding the attacker's activities. After this technique became known, the ease in which one could detect modified files on a system prevented it from being an effective way to maintain stealth.

The next generation of these rootkits involved loading a module into the kernel similar to that of a device driver, which would intercept functions at the lowest level, and modify them to report false output to the calling process. Once the kernel has been compromised, it can be very difficult to detect the presence of the attacker, introducing a significant threat to system administrators. While these rootkits have been primarily targeting Unix environments, they have been gaining more attention in Windows environments in recent years. One drawback to the approach of using a kernel module from the attacker's point of view is that increased attention has been given to ensuring unauthorized drivers are not loaded into the kernel.

This document focuses on a novel approach to accomplishing a comparable degree of stealth that has received little attention and is less widely known. Using the technique described, attackers may be able to create an environment similar in functionality to that of a kernel-based rootkit, with no need to modify the kernel in any way, and as such can be executed by an administrative user without access to the kernel. This technique applies to Microsoft Windows NT-based operating systems.

Technical Description

In order to avoid detection, an attacker must somehow prevent applications used by the system administrator to monitor activity from displaying evidence of their presence. In order to accomplish this without sufficient privileges to modify the kernel, the most viable alternative is to modify the calling application at runtime. The technique that this type of attack relies on, commonly referred to as "DLL injection", allows an attacker to compile all functionality of the rootkit into a DLL, which is to be loaded by the target process.

The steps taken to successfully achieve a userspace DLL injection rootkit include the following:

- ***Process Enumeration***
  Determining which processes must be attacked, and compiling a list of target processes

- ***Code Insertion***
  Writing malicious code to the memory space of a target process

- ***Code Execution***
  Causing code written to a foreign process to be executed by that process

- ***Function Hijacking***
  Using code introduced by the attacker to hook the desired functions

These steps, which are further detailed below, will ultimately give an attacker a relatively firm grasp on the general operation of the system.

Process Enumeration

The first requirement for achieving this stealth is to modify all processes that may be used in detecting the attacker. Because it is virtually impossible to foresee which applications may be used by an administrator, the attacker is most likely to simply gather a list of every process on the system and attack them sequentially. This approach also allows a convenient way to omit certain processes from being attacked. As a result, tools used by the attacker will be exempt from any functionality employed by the rootkit, allowing them to view the system in its natural state. This list may be gathered with the `Process32First()` and `Process32Next()` API calls. The prototypes for these functions, taken from the Microsoft Developer's Network (MSDN), are below:

```
Process32First()

Retrieves information about the first process encountered in a system
   snapshot.

BOOL WINAPI Process32First(
  HANDLE hSnapshot,
  LPPROCESSENTRY32 lppe
);

Process32Next()

Retrieves information about the next process recorded in a system
   snapshot.

BOOL WINAPI Process32Next(
  HANDLE hSnapshot,
  LPPROCESSENTRY32 lppe
);
```

Alternatively, similar results may be possible by using the `EnumWindows()` API call, which is described in MSDN as follows:

```
The EnumWindows function enumerates all top-level windows on the screen by
   passing the handle to each window, in turn, to an application-defined
   callback function. EnumWindows continues until the last top-level
   window is enumerated or the callback function returns FALSE.

BOOL EnumWindows(

   WNDENUMPROC lpEnumFunc,
   LPARAM lParam
);
```

The EnumWindows() API will cycle through every application that has a window, including the root desktop window, and execute the callback function registered with the lpEnumFunc parameter. While this method will not ensure that the rootkit is loaded into every process on the system, it will be loaded into every process that has a graphical user interface, as well as every process that is created by a process with a graphical user interface. This may be sufficient to prevent detection, and is known to be used by the Vanquish rootkit, which is one of the few pieces of malicious code publicly available known to employ the technique described in this document.

Code Insertion

Upon selection of target processes, the attacker modifies the memory of the running application in such a way that additional code may be introduced into the process image. This involves the use of two significant Windows API functions designed to aid in manipulating the memory of a remote process. The first function used, VirtualAllocEx(), is described on MSDN as follows:

```
The VirtualAllocEx function reserves or commits a region of memory within
   the virtual address space of a specified process. The function
   initializes the memory it allocates to zero, unless MEM_RESET is used.

LPVOID VirtualAllocEx(
  HANDLE hProcess,
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD flAllocationType,
  DWORD flProtect
);
```

By issuing a call to the VirtualAllocEx() Win32 API function, it is possible to allocate a region of memory within the address space of another process. After successful allocation, WriteProcessMemory() is used, which is described on MSDN as follows:

```
The WriteProcessMemory function writes data to an area of memory in a
   specified process. The entire area to be written to must be accessible,
   or the operation fails.

BOOL WriteProcessMemory(
  HANDLE hProcess,
  LPVOID lpBaseAddress,
  LPCVOID lpBuffer,
  SIZE_T nSize,
  SIZE_T* lpNumberOfBytesWritten
);
```

Using VirtualAllocEx() in conjunction with a call to WriteProcessMemory(), an attacker is able to write arbitrary data to an allocated region of memory within the target process. For this technique, a string containing the name of the rootkit DLL filename is written.

Code Execution

After insertion of the attackers rootkit code is complete, the attacker must be able to execute the inserted code. Fortunately for the attacker, an API is included in the Windows NT-based operating systems called `CreateRemoteThread()`, which simplifies this task. The prototype for `CreateRemoteThread()`, as listed on MSDN, is as follows:

```
The CreateRemoteThread[2] function creates a thread that runs in the virtual
   address space of another process.

HANDLE CreateRemoteThread(
  HANDLE hProcess,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  SIZE_T dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID lpParameter,
  DWORD dwCreationFlags,
  LPDWORD lpThreadId
);
```

Of note are the `hProcess`, `lpStartAddress`, and `lpParameter` parameters. By supplying a handle to the target process in the `hProcess` parameter, a function pointer to the `LoadLibrary()` API call as the `lpStartAddress` parameter, and a pointer to the string containing the name of a malicious DLL compiled by the attacker (which has previously been written via the `WriteProcessMemory()` call discussed prior), the target process will be forced to spawn a child thread which will essentially execute `LoadLibrary("rootkit.dll");` and terminate. `LoadLibrary()` is explained in the MSDN reference below:

```
The LoadLibrary function maps the specified executable module into the
   address space of the calling process.

HMODULE LoadLibrary(
  LPCTSTR lpFileName
);
```

During the initialization stage of loading a dynamically linked library, execution is passed to the entry point of the DLL module, typically declared as `DllMain()`. By supplying malicious code within the `DllMain()` function, the attacker is now able to execute arbitrary commands within the context of the target process.

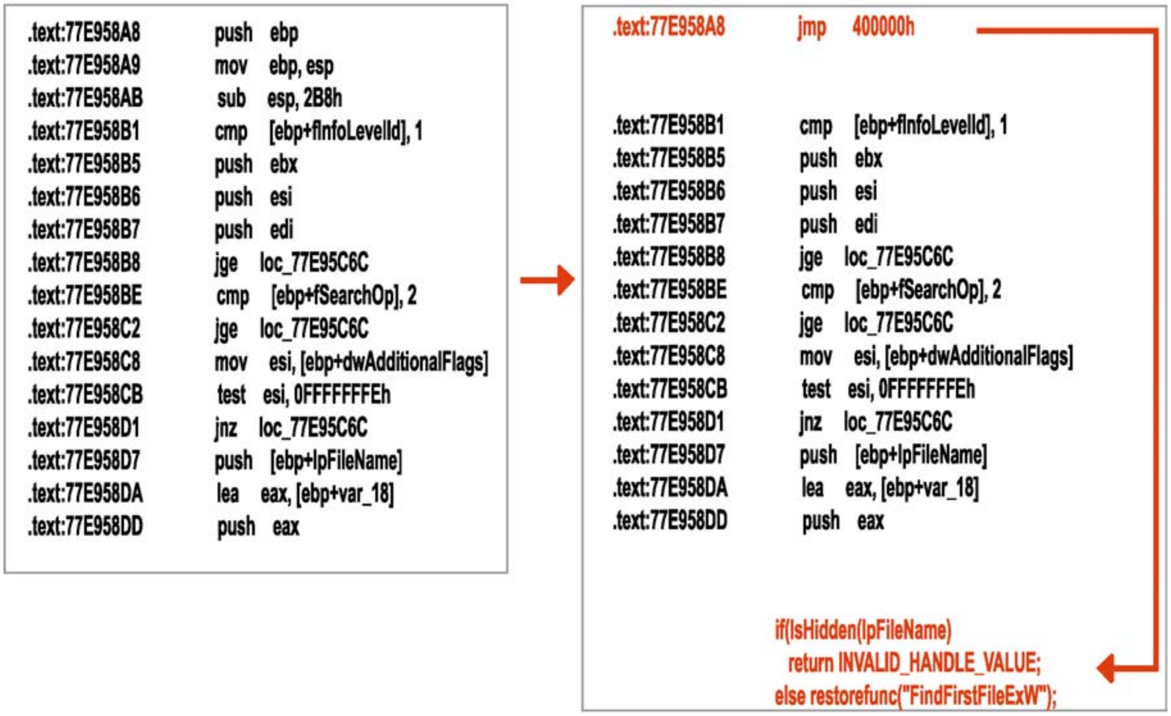
Function Hijacking

Once the attacker is able to introduce and execute arbitrary code, the final step is to use the malicious DLL module to implement hooks on all necessary functions in order to cloak the attacker's presence on the system. This is done by modifying the function code loaded into the memory space of the current process in such a way that it is redirected to a routine designed to sanitize the input or output of the call to that function. To accomplish this, the rootkit will modify the first few instructions of the function to be hooked, with an unconditional jump instruction to somewhere in memory where the rogue function has been written. In order to maintain system stability, it is necessary to save the bytes that are overwritten

[2] The CreateRemoteThread API is only available on Windows NT, XP, and 2000. As a result, this method of code insertion will not function on Windows 9x, or ME.

to an alternative location, which will then be preserved and executed upon completion of the rogue function. This allows complete control over the input and output of any function calls made by the process in question. To ensure persistence of the rootkit across all processes created subsequent to the initial enumeration and injection, functions capable of spawning new processes, such as `CreateProcess()` may also be hooked in a similar fashion.

The code segment is not writeable by default, which prevents any code from being modified subsequent to runtime. Unfortunately, the permissions on the write-protected pages can be trivially modified by an attacker, by using the `VirtualProtect()` call.



```
.text:77E958A8      push   ebp
.text:77E958A9      mov    ebp, esp
.text:77E958AB      sub    esp, 2B8h
.text:77E958B1      cmp    [ebp+fInfoLevelId], 1
.text:77E958B5      push   ebx
.text:77E958B6      push   esi
.text:77E958B7      push   edi
.text:77E958B8      jge    loc_77E95C6C
.text:77E958BE      cmp    [ebp+fSearchOp], 2
.text:77E958C2      jge    loc_77E95C6C
.text:77E958C8      mov    esi, [ebp+dwAdditionalFlags]
.text:77E958CB      test   esi, 0FFFFFFFEh
.text:77E958D1      jnz    loc_77E95C6C
.text:77E958D7      push   [ebp+lpFileName]
.text:77E958DA      lea    eax, [ebp+var_18]
.text:77E958DD      push   eax
```

```
.text:77E958A8      jmp    400000h


.text:77E958B1      cmp    [ebp+fInfoLevelId], 1
.text:77E958B5      push   ebx
.text:77E958B6      push   esi
.text:77E958B7      push   edi
.text:77E958B8      jge    loc_77E95C6C
.text:77E958BE      cmp    [ebp+fSearchOp], 2
.text:77E958C2      jge    loc_77E95C6C
.text:77E958C8      mov    esi, [ebp+dwAdditionalFlags]
.text:77E958CB      test   esi, 0FFFFFFFEh
.text:77E958D1      jnz    loc_77E95C6C
.text:77E958D7      push   [ebp+lpFileName]
.text:77E958DA      lea    eax, [ebp+var_18]
.text:77E958DD      push   eax


if(IsHidden(lpFileName)
  return INVALID_HANDLE_VALUE;
else restorefunc("FindFirstFileExW");
```

Figure 1. *Function Hijacking*

The pseudo-code in **Figure 1** illustrates an example where the `FindFirstFileExW()` API function is modified to immediately jump to a procedure which first checks to ensure the filename to be located is not to be marked hidden by the rootkit, and then restores the original `FindFirstFileExW()` function to execute normally. In the event that a hidden file is being passed, the procedure immediately returns with the error code INVALID_HANDLE_VALUE, causing the calling application to believe that the file does not in fact exist.

Functions likely to be modified include those used to display processes, files, registry keys, and network statistics.

Exploiting Inter-process Trust Relationships

By executing code within the context of another process, it may be possible to escalate privileges or circumvent access controls based on restrictions as to which processes are permitted to use a given resource. An example of this is the use of traditional desktop firewalls to prevent unauthorized applications from communicating with other hosts on the network. Suppose a restriction has been placed on a system to ensure that the only application permitted to send and receive data is a Web browser. This would hinder an attacker's ability to communicate with the system via a backdoor, such as an IRC bot designed to make an outbound connection to an IRC server controlled by the attacker, which has been covertly placed on the system. Such an attempt is likely to generate an alert and reveal the presence of the attacker and their utilities on the compromised system. However, by inserting the malicious code into the Web browser process via DLL injection, this communication can take place. From the perspective of the access control mechanism, there is no anomalous communication vector, as the Web browser is performing the communication on behalf of the attacker. This also does not require an additional process to be spawned, which may arouse the suspicion of an administrator.

Using DLL injection in this manner has been slowly gaining popularity among malicious code authors for this reason, being demonstrated in recent backdoors such as Backdoor.Spotcom, as well as W32.Randex.E (a variant of Spybot).

Conclusion

The ability to intercept and redirect any given library call imported by an application makes any malicious code employing this technique potentially very difficult to detect. It is possible for an attacker to circumvent any detection mechanisms that rely on code that the rootkit is able to write to, as the output of any function calls made cannot be trusted. For example, the rootkit may add a registry key to ensure that it is started automatically upon system boot. Savvy administrators will monitor these particular registry keys, as it is a method very often required to ensure that backdoors are executed, and it is well known that unauthorized entries may indicate suspicious activity. However, if the registry editor is attacked immediately after loading, it is virtually impossible for an administrator to view the rootkit entry, as it will have been modified to omit the suspicious key from the list displayed.

Because of the restriction imposed by the operating system that disallows modification of kernel space code by user space applications, it is possible to write a detection mechanism with the ability to execute code at a lower level than the rootkit. This requires the use of a device driver, and ensures that the code within the device driver has not been tampered with by this rootkit. By taking advantage of this, reliable detection of this type of threat may be possible.

It is also for this reason that Network Intrusion Detections Systems (NIDS) should be used whenever possible. While much of the information reported by a compromised system could potentially have been tampered with, remote systems and appliances may aid in providing clues as to whether or not an attack has taken place. Witnessing suspicious communications to and from a given machine that is inconsistent with what the machine in question is reporting may be a sure sign that it is infected with a rootkit.

Due to the requirement of Administrator privileges to successfully load this type of rootkit, properly secured systems are not at risk. By implementing strict security policies and auditing system activity regularly, the risk of becoming victim to such threats is reduced significantly.

Change Log

Version 1: January 13, 2004, 00:30 GMT
Initial Threat Analysis released.

Glossary

If you are unfamiliar with any term this report uses, please visit the Symantec glossary at http://www.securityfocus.com/glossary for more details on information security terminology.

Contact Information

World Headquarters

Symantec Corporation
20300 Stevens Creek Blvd.
Cupertino, CA 95014
U.S.A.
+1 408 517 8000
www.symantec.com

Symantec DeepSight Solutions

Symantec DeepSight Customer Service
+ 1 866 732 3682 (Toll-Free)
+ 1 541 335 7020
DeepSightCustServ@symantec.com

About Symantec

Symantec, the world leader in Internet security technology, provides a broad range of content and network security software and appliance solutions to enterprises, individuals, and service providers. The company is a leading provider of client, gateway, and server security solutions for virus protection, firewall and virtual private network, vulnerability management, intrusion detection, Internet content and e-mail filtering and remote management technologies, as well as security services to enterprises and service providers around the world. Symantec's Norton brand of consumer security products is a leader in worldwide retail sales and industry awards. Headquartered in Cupertino, Calif., Symantec has worldwide operations in 38 countries. For more information, please visit www.symantec.com.

DeepSight Conditions: NO WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT, SHALL APPLY TO THE DEEPSIGHT SERVICES OR THE MATERIALS PROVIDED BY SYMANTEC TO USERS OF THE DEEPSIGHT SERVICES. SYMANTEC PROVIDES THE SERVICE(S) AND MATERIALS "AS IS" AND "AS AVAILABLE." IN NO EVENT WILL SYMANTEC BE LIABLE FOR THE TRUTH, ACCURACY, RELIABILITY OR COMPLETENESS OF THE SERVICE(S) OR MATERIALS. SYMANTEC MAKES NO WARRANTY THAT THE SERVICE(S) OR MATERIALS WILL BE UNINTERRUPTED OR TIMELY, OR THAT THEY WILL PROTECT AGAINST COMPUTER VULNERABILITIES. Please refer to your services agreement or certificate for further information on conditions of use for the Services and materials.

Quoting Symantec Information and Data: Authorized Users of Symantec's DeepSight Services may use or quote individual sentences and paragraphs from the materials provided as part of the Services, but not large portions or the majority of such materials, solely for purposes of internal communications. Unless otherwise specifically agreed in writing by Symantec, no external publication of all or any portion of any materials provided by Symantec is permitted.