Thema der Ausgabe

Rootkits unter Windows-Plattformen

Nzeka Gilbert 

Schwierigkeitsgrad

Was ist die Verbindung zwischen Kernel-Hackern (in diesem Artikel werden wir den Begriff Kernel anstatt der Kern eines Betriebssystems verwenden), Unternehmen, die Webmarketing-Betriebe haben, welche Spyware oder Adware entwickeln, um Websurfer zu profilieren und Unternehmen wie Sony (welches ein DRM-System verwendet, das von First 4 Internet entwickelt wurde)?

Die Rootkits, die oft von Hackern benutzt werden, die schon Systeme kompromittiert haben, versuchen unsichtbare Tools aufzusetzen, werden mehr und mehr gebräuchlich in letzter Zeit. Diese erlauben ihnen einfach auf ihre Spuren zurückzukehren (solche Tools werden Hintertüren genannt) und die Modifikationen, die sie gemacht haben, zu verstecken, bevor ein Administrator bemerkt, dass seine Systeme geknackt wurden.

Die Rootkits sind bereits in der Unix-Welt bekannt. Sie gehören in jeden Werkzeugkoffer zum Überleben eines Hackers. Unter Linux bestehen die Rootkits generell aus einer Backdoor, einem Sniffer, einem Log-Wischer (einem Log-Zerstörer) und einigen anderen Programmen. Diese Programme ersetzen legitime Komponenten eines Systems (wie ps, netstat). Es gibt zwei Arten von Rootkits: Die, die wie normale Programme funktionieren und jene, die modular sind, wie LKM (Ladbares Kernel Modul oder Linux Kernel Modul). Das Charakteristische an LKM-Rootkits (und was ihre Kraft ausmacht) ist, dass sie fähig sind, Systemaufrufe abzufangen und das Verhalten von Unix (bzw. seinem Kernel) zu modifizieren, wenn er einigen spezifischen Aktionen gegenübersteht.

Diese Art von bösartigem Code existiert auch auf Windows-Plattformen. Es gibt aber einen großen Unterschied: Wir können unsere Arbeit nicht auf gültigen Quellcodes erklären, um zu verstehen, wie der Windows-Kernel und alle seine Komponenten (Objekte im Windowsjargon genannt)

In diesem Artikel erfahren Sie...

- die leitenden Prinzipien von Rootkits und die Techniken/Tools, die von Rootkits-Entwicklern verwendet werden;
- wie Sie Ihre eigenen Rootkits erstellen können, die im Userland- und/oder im Kernel-Modus arbeiten;
- wie Sie den Windows-Kernel dank freier/Open Source Software analysieren können;
- wie Sie eine personalisierte GINA erstellen können.

Was Sie vorher wissen/können sollten...

- wie der Speicher in der INTEL-Architektur gesteuert wird;
- wie das PE Dateiformat arbeitet;
- wie man Programmsoftware und DLLs programmiert.

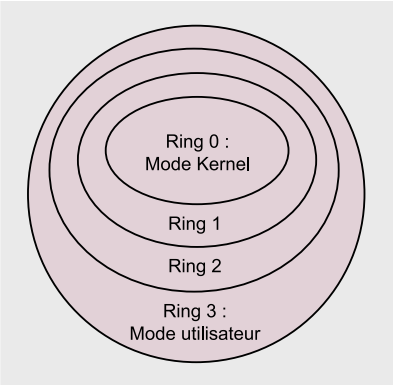


Abbildung 1. *Die Ringe*

arbeiten. Das ist der Grund, warum die Fähigkeit, Software umzukehren (d.h., seinen ASM-Code auszugeben und zu verstehen) eine Basisfertigkeit ist, die alle Windowshacker haben müssen.

In diesem Artikel werden wir Ihnen helfen, die Welt der Rootkits unter Windowsplattformen zu betreten, indem wir damit beginnen, die leitenden Grundsätze darzulegen. Dann werden wir die Entwicklung von Nicht-Kernel- und Kernel-Toolkits besprechen. Zum Abschluss werden wir einige Absätze den Erkennungstools und den fortgeschrittenen Techniken, die sehr wahrscheinlich von zukünftigen Rootkits verwendet werden, widmen.

Zwei Rootkits, die von der Webseite des Autors heruntergeladen werden können, wurden für diesen Artikel erstellt. Das erste ist *Ring3RK*, in dem Techniken implementiert wurden, die von Nicht-Kernel-Rootkits verwendet werden. Das nächste, *Ring0RK*, basiert auf einer modifizierten Version des FU Rootkits (ein Rootkit, das von James Butler entwickelt wurde). Die Quelltexte dieser Rootkits werden nicht in ihrer kompletten Version angeboten, da sie in ihrer vollständigen Version verwendet werden, wie ein Framework durch den Autor, der regelmäßig die neuesten netten und modischen Techniken implementiert: So wird es möglich sein, dass Sie einige Zeilen Code finden, die nicht von der angebotetenen Version verwendet werden.

Definition eines Rootkit

Ein Rootkit ist ein Programm oder eine Reihe von Programmen, die dem Entwickler ermöglichen, auf einem Computer seine Spuren und seine Waffen zu verstecken, alles, was in größter Diskretion getan wird. Ein Rootkit ist weder ein Virus noch ein anderer Typ von Malware, die versucht, die größte Anzahl an Leuten oder Dateien zu infizieren. Wenn ein Hacker ein System bereits kompromittiert hat, wird er danach suchen, wo er Hintertüren verstecken kann, um einfach auf ein frisch gehacktes System zurückzukehren. Ein Administrator kann aber die Hintertüren und Hackerdateien finden: Das letztere muss somit das Verhalten des infiltrierten Systems modifizieren, um unsichtbar zu sein. Das ist der Zeitpunkt, wo Rootkits eingreifen: Sie versuchen, einige der Sicherheitstools, die der Administrator verwenden kann, in die Irre zu führen. Das tun Sie, indem Sie das System dazu bringen, anzunehmen, es sei gesund, während es einige Dateien und Programme versteckt, die der Hacker auf der Festplatte haben will. Es ist somit möglich, die Basisfunktionen eines System zu modifizieren, um Dateien zu verstecken, indem man dem System mitteilt, dass letztere nicht existieren, oder um Netzwerkverbindungen und Prozesse zu verstecken und um Analysetools zu Fehlern zu verleiten, während man direkt auf die Speicherseiten einwirkt.

Windows-Sicherheitsmodell

Wir werden keine Inventur der in Windows implementierten Sicherheitssysteme machen, wir werden nur über das Privilegienmanagement in diesem Betriebssystem sprechen: Die Hauptelemente, über die wir während der Entwicklung eines Rootkits nachdenken müssen. Ganz einfach, weil die Rootkits in zwei Familien unterteilt sind, die wir später vorstellen werden.

Es gibt 2 Ausführungsarten für ausführbare Dateien unter Windows: die Userland und die Kernel (der Kern). Im Userland bietet Windows eine API (über seine DLLs), die jeder Entwickler benutzen kann. Es ist die Stelle, wo Software wie Paint oder Dev-Cpp einsetzt. Auch wenn er die Systemaufrufe bietet, auf denen die API basiert, muss der Kernel geschützt werden und nicht von Userland Software zugreifbar sein. Mit dieser Intention im Hinterkopf haben Windowsentwickler einen zweiten Modus erstellt: den Kernel-Modus. Die Binärdateien, die in diesem Modus ausgeführt werden, haben auf das ganze System ohne Einschränkung Zugriff: Speicher, Prozessortabellen, Systeme, die die Prozesse steuern, Sicherheitssysteme…

In Übereinstimmung mit dem Modus, in dem das Rootkit arbeiten wird, wird es mehr oder weniger Fähigkeiten haben. Es gibt somit zwei Arten von Rootkits: die Userland Rootkits und die Kernel Rootkits. Die Userland Rootkits sind gemeinhin aus einer Reihe kleiner Tools zusammengestellt, die benutzt werden, um gesunde Programme zu ersetzen. Die ermöglichen dem Angreifer, unsichtbar zu sein. Sie können ebenso Techniken ausnutzen, die etwas weiter fortgeschritten sind, wie API-Hooking (deutsch: Einhaken, Anm. d. Übers.), DLL Injektion oder das inline Funktions-Hooking, um die Arbeit gesunder Software *on the fly* zu modifizieren, ohne sie zu ersetzen. Bei diesem arbeiten sie auf den privaten Daten der Software direkt im Speicher. Die Kernel Rootkits sind im allgemeinen so geschrieben wie Windows-Treiber (erstellt wie alle anderen Treiber unter Benutzung der Microsoft DDK), die Zugriff auf alle Objekte des Systems haben: Sie können somit alles tun, was sie wollen. Wie unter Linux wird es z.B. für einen Kernel-Treiber möglich sein, die SSDT zu modifizieren, welche in

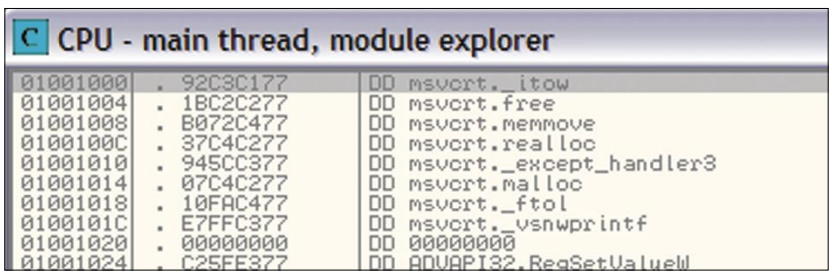


Abbildung 2. *Wie man die IAT einer Software mit OllyDbg lokalisiert*

Listing 1. *Explorer.exe's PE Header*

```
C:\khaalel>pedump.exe
PEDUMP - Win32/Win64 EXE/OBJ/LIB/DBG file dumper - 2001 Matt Pietrek
Syntax: PEDUMP [switches] filename
  /A    include everything in dump
  /B    show base relocations
  /H    include hex dump of sections
  /I    include Import Address Table thunk addresses
  /L    include line number information
  /P    include PDATA (runtime functions)
  /R    include detailed resources (stringtables and dialogs)
  /S    show symbol table
C:\khaalel>pedump.exe /A C:\WINDOWS\explorer.exe >> explorer.exe.txt
C:\khaalel>explorer.exe.txt
…
Imports Table:
  msvcrt.dll
   Import Lookup Table RVA:  00042C68
   TimeDateStamp:            FFFFFFFF
   ForwarderChain:           FFFFFFFF
   DLL Name RVA:             00042BC8
   Import Address Table RVA: 00001000
…
```

der Windows-Welt äquivalent zur Unix syscalls Tabelle ist.

x86 Prozessorenarchitektur: Die Ringe und ihre Konsequenzen

Die Ringe sind die Grundlage des Privilegienmanagements unter Windowsplattformen (aber auch unter Linux). Die Ringe sind Konzepte, die von Intel undseinen Microprozessoren x86 eingeführt wurden (Abbildung 1).

In dieser Prozessorenfamilie gibt es vier Ringe (von *ring0* bis *ring3*), um die Art zu kontrollieren, wie die Systemobjekte arbeiten. Aktuell werden nur zwei dieser Ringe von allen Betriebssystemen verwendet: der *ring0* und der *ring3*. Im *ring0* ist der Kernel-Modus und im *ring3* der Userland. Dieser Wille verursacht ein Sicherheitsproblem: Alle Objekte, die im KernelModus ausgeführt werden, können alle Ressourcen des Systems erreichen. Der Kernel selbst ist nicht von den dritten Treibern und anderen Arten von LKM (Ladbare Kernel Module) getrennt. Letztere sind in der Lage, die verschiedenen Objekte des Kernels zu erreichen und damit Unfug zu treiben.

x86 Prozessorenarchitektur: Die Adresstabellen

Um der Userland zu ermöglichen, mit dem Kernel-Modus zu kommunizieren, benutzt das System Unterbrechungen. Wenn die CPU eine Unterbrechung erhält, versteht sie, dass sie einen Übergang von Userland auf den Kernel-Modus durchführen und die adäquaten Routinen ausführen muss. Stellen wir uns z.B. eine Dateisuchsoftware vor. Um eine Auswahl zu durchsuchen, wird sie die INT2E-Unterbrechung senden, während sie z.B. die `NtQueryDirectoryFile`-Funktion braucht (solch ein Aufruf wird erledigt, indem adäquate Informationen in die Prozessorregister gelegt werden). Wie zu erwarten, wird die CPU, eine erhebliche Menge an Routinen brauchen, um in der Lage zu sein, alle möglichen Aktionen im System zu steuern. Sie ist nicht in der Lage, diese in ihren eigenen Speichersegmenten zu speichern; daher werden einige Adresstabellen verwendet. Diese Tabellen werden die Speicheradressen einiger Routinen speichern.

Global Descriptor Table (Schlüsselwörter: GDT – SGDT) und Local Descriptor Table (Schlüsselwörter: LDT)

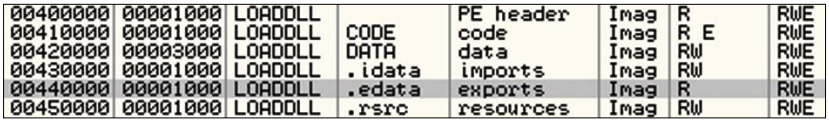
Die GDT und die LDT ermöglichen, den Speicher in Segmente zu unterteilen. Sie sind Tabellen, die Listen von Segmentbeschreibern enthalten. Ein Segmentbeschreiber ist eine 8 Byte-Struktur, die Daten über das physische Segment des Speichers enthält. Wie sein Name anzeigt, erlaubt ein Segmentbeschreiber, Segmente des Speichers zu beschreiben. Diese Nummer zeigt auf den gewünschten Deskriptor. Ein Element in den Segment-Deskriptoren, das Rootkits-Entwickler interessieren kann, ist das DPL (für Descriptor Privilege Level), das ermöglicht zu erfahren, ob dieses oder jenes Segment im Modus Kernel oder User zugreifbar ist.

Es ist ebenso möglich, die GDT-Position zu modifizieren – dank LGDT-Anweisung. Warum? Weil die GDT irgendwo im Speicher gespeichert werden kann, so lange der Prozessor weiß, wo sie sich befindet. Das erste GDT-Elemente kann unter anderen Dingen gefunden werden - dank der SGDT-Anweisung. Der große Unterschied zwischen dem GDT und dem LDT liegt in der Tatsache, dass ein System nur einen GDT haben kann und das auf der anderen Seite etliche LDT erstellt werden können (jedes hat natürlich andere Aufgaben). Wir sprachen weiter oben über die Segmentregister. Es gibt 6 Segmentregister. Sie sind durch die folgenden Kennzeichnungen identifiziert: CS, DS, ES, FS, GS, SS. Sie werden verwendet, um die Anfangsadresse eines Segments zu speichern (Startadresse einer Anwendungsanweisung, Daten oder Stack).

Hier, entsprechend den offiziellen INTEL Entwicklerhandbüchern (*http://www.intel.com/design/pentium4/*

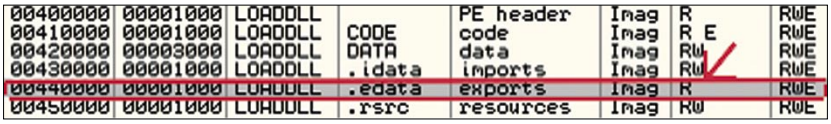
Avant Propos

Der Großteil der Konzepte, die hier angegangen werden, benötigen Wissen über das PE Dateiformat (Portable Executable). Es beschreibt die Architektur von Binärdateien unter Windowssystemen. Artikel zum Thema im MSDN-Portal:
http://msdn.microsoft.com/msdnmag/issues/02/03/PE2/default.aspx
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndebug/html/msdn_peeringpe.asp.



00400000	00001000	LOADDLL		PE header	Imag	R	RWE
00410000	00001000	LOADDLL	CODE	code	Imag	R E	RWE
00420000	00003000	LOADDLL	DATA	data	Imag	RW	RWE
00430000	00001000	LOADDLL	.idata	imports	Imag	RW	RWE
00440000	00001000	LOADDLL	.edata	exports	Imag	R	RWE
00450000	00001000	LOADDLL	.rsrc	resources	Imag	RW	RWE

Abbildung 3. *Lokalisierung der .edata Sektion*



00400000	00001000	LOADDLL		PE header	Imag	R	RWE
00410000	00001000	LOADDLL	CODE	code	Imag	R E	RWE
00420000	00003000	LOADDLL	DATA	data	Imag	RW	RWE
00430000	00001000	LOADDLL	.idata	imports	Imag	RW	RWE
00440000	00001000	LOADDLL	.edata	exports	Imag	R	RWE
00450000	00001000	LOADDLL	.rsrc	resources	Imag	RW	RWE

Abbildung 4. *Sichten des .edata read-only Flag*

manuals/index_new.htm), ist die Beschreibung dieser Segmentregister. CS (für Code Segment) ist ein 16 Bit-Register, das die Startadresse der Binäranweisungen eines Programms oder einer Subroutine anzeigt, die der Prozessor ausführen muss.

SS (für Stack Segment) ist ein 16 Bit Register, das auf die Speicherzone des Programmstacks zeigt, der ausführt. Ein wichtiger Punkt muss hervorgehoben werden: Das CS Register kann nicht von den Anweisungen unserer Programme modifiziert werden, da wir es nicht erreichen können: Auf der anderen Seite kann das SS-Register zum Benutzen verschiedener Stacks gehandhabt werden.

Die folgenden Register (DS, ES, FS und GS) zeigen auf Datensegmente. Vier Register wurden erstellt, um den Zugriff auf die diversen Datenstrukturen eines Programmes zu erleichtern, das auf vier verschiedene Segmente verteilt sein kann.

DS (für Daten Segment) ist ein 16 Bit Register, das die Startadresse der Programmdaten enthält. Zur Information: Der Wert dieses Registers wird modifiziert, wenn verschiedene Segmente benutzt werden.

ES, GS und FS sind zusätzliche Register, die von Entwicklern verwendet werden können, die die Intel-Architektur ausnutzen wollen: Sie können sie benutzen, wie sie wollen. Sie werden oft benutzt, um sich auf andere Arten von Daten zu beziehen.

Interrupt Descriptor Table (Schlüsselwörter: IDT – IDTR)

Die IDT ist eine 256-Einträge-Tabelle, die die Routinenadressen speichert. Diese werden die (256) Unterbrechungen steuern. Die IDTR (Interrupt Descriptor Table Register) enthält die IDT-Adresse. Um ihren Wert zu laden, müssen wir die SIDT-Anweisung (Speichere IDT) benutzen. Um sie zu modifizieren, müssen wir die LIDT-Anweisung (Lade IDT) benutzen. Wie wir später sehen werden, ist es leicht möglich, die IDT-Inhalte aufzulisten, einen Hook einzubringen und sogar eine neue IDT unter größter Diskretion zu erstellen. Die IDT erlaubt es, Systemaufrufe und andere Dinge zu machen: Zum Beispiel verwendet SoftIce eine Unterbrechung (die 0x03) für seinen BPX-Befehl.

System Service Dispatch Table (SSDT)

Die SSDT (oder Dispatcher Table) ist unter der Windowsplattform äquivalent zu der Systemaufruftabelle der Unixsysteme. Windows bietet dem Userland eine Menge von APIs, um die Entwicklung von Anwendungen zu ermöglichen, ohne etwas im Kernel-Modus ausführen zu müssen. Um in der Lage zu sein, jede Handlung zu beantworten, machen die Systeme Systemaufrufe. Sie senden eine 0x2E-Unterbrechung und fügen in die passenden Register die verschiedenen Parameter ein, die ein Systemaufruf brauchen kann. Tatsächlich wird die 0x2E-Unterbrechung auf den alten Plattformen verwendet. Unter Windows XP ist es die SYSENTER-Anweisung, die verwendet wird.

Import Address Table (IAT)

Das System bietet eine Reihe von DLLs, die ermöglichen, Programme zu erstellen. Dabei kümmern Sie sich nicht um die darunter liegenden Systemaufrufe, die bei jeder neuen Windowsversion modifiziert werden können. Wie sind die in einem Programm verwendeten, in einer DLL definierten Funktionen angeordnet (Windows bietet eine Menge DLLs)?

Während der Software-Initialisierung wird seine IAT durchlaufen. Diese IAT hat eine Liste aller in der Software verwendeten Funktionen und kennt den Namen der DLLs, die sie enthalten. Der Anwendungs-Lader wird somit die Adresse der Funktionen im Speicher suchen und diese Information in die IAT der Software packen. Wenn die DLL nicht im Speicher ist, wird sie geladen. Jedes Mal, wenn die Software den Code einer in einer DLL definierten Funktion ausführen will, wird sie einen Sprung in ihrer IAT zu dem Platz machen, wo die Adresse der gewünschten Funktion ist.

Listing 2. *Wie man dank eines WMI-Skripts eine Liste der aktiven Dienste erhält*

```
'---------------------------------------------------------------
'This script has been written with WMI Code Creator
'from Microsoft Labs
'---------------------------------------------------------------
Dim i
i = 0
strComputer = "."
Set objWMIService = GetObject("winmgmts:\\" & strComputer & "\root\CIMV2")
Set colItems = objWMIService.ExecQuery( _
    "SELECT * FROM Win32_Process",,48)
For Each objItem in colItems
'    Wscript.Echo "-----------------------------------"
'    Wscript.Echo "Win32_Process instance"
'    Wscript.Echo "-----------------------------------"
    Wscript.Echo "Name: " & objItem.Description
     i = i + 1
Next
Wscript.Echo i
```

Wir können, wenn wir OllyDbg benutzen, die IAT jeder Anwendung lokalisieren. Für die, die nicht mit dem PE-Dateiformat vertraut sind, werden wir noch darüber sprechen. Lassen Sie uns damit beginnen, *PEDump.exe* von Matt Pietrek zu benutzen, um diese Sektion zu lokalisieren.

Wir lokalisierten den Beginn der Importtabelle. Um das zu überprüfen, können wir OllyDbg öffnen. Wir haben gerade ein neues technisches Wort gesehen, das Sie kennen sollten: RVA (Relative Virtual Address). Dieses Konzept erlaubt uns, die Position eines Elements (wie Tabellen) in den PE-Dateien (wie die EXE, DLLs), beginnend mit der Basisadresse der PE-Datei, zu erfahren. Was auch immer die Position des Dateibeginns im Speicher ist, dank dem RVA ist es immer möglich, ein Symbol zu finden. Sagen wir z.B., dass die PE-Datei in den Speicher an der virtuellen Adresse 0x10000 geladen ist. Der RVA der IAT sei 00001000. Somit können wir die Position der Tabelle im Speicherabbild finden, da letzteres sich an der folgenden Adresse befindet: 0x01000000 + 0x00001000 = 0x01001000.

Das IAT-Hooking wird daraus bestehen, die IAT-Einträge eines Programms, welches unsere Funktionen ausführt (in einer DLL unseres Rootkits implementiert), zu modifizieren. Dem gehackten Programm verbieten wir, gültige Windows-Funktionen auszuführen.

Export Address Table (EAT)

Obwohl das IAT-Hooking einfach aufzusetzen und ziemlich mächtig ist, hat es beachtliche Nachteile. Es ist sehr einfach zu entdecken. Wenn die Software entscheidet, die Adresse einer Funktion nicht während des Starts zu suchen, sondern erst, wenn Sie sie verwendet, wird das IAT-Hooking ganz einfach nicht funktionieren.

Um die Adresse einer Funktion zu finden, wird oft die `GetProcAdress`-Funktion verwendet. Das Ziel des EAT-Hookings ist, diese Funktion zu hijacken. Wenn eine Software (jede Software!) eine spezifische Funktion (die wir zuvor ge-hijackt haben) aufruft, wird jedes mal anstelle der gültigen Windows-Funktion eine Funktion aufgerufen, die in der DLL unseres Rootkits implementiert ist. Es ist eine Alternative zum IAT-Hooking, die ziemlich mächtig, aber ebenfalls feststellbar ist.

Listing 3. *Wie man Prozesse von einer Ersatz-GINA startet*

```
int LaunchApp(){
    int VaLid = -1;
    // Zur Information: die folgende Struktur wird von CreateProcess-ähnlichen
    // Funktionen verwendet um das Fenster des neuen Prozesses zu spezifizieren
    // (Erscheinungsbild...)
    STARTUPINFO si;
    // Zur Information: die folgende Struktur wird von CreateProcess-ähnlichen
    // Funktionen verwendet um Informationen über den neuen Prozess zu erhalten
    // (wie Prozess und erster Thread PID, handle..)
    PROCESS_INFORMATION pi;
    BOOL Retour = FALSE;
    wchar_t szProcess[] = L"C:\\smartcard.exe";
    wchar_t szCmdLine[] = L"";
    int WhatIsClicked;
    int WhatIsChoose;
      WhatIsClicked = MessageBox( NULL, "Do you want to use your smart card
         for authentication?", "SmartCard Reader", MB_YESNO );
   if ( (VaLid = ParseDumpFile("C:\\ pubfile.hex")) == 0 ){
      remove("C:\\ pubfile.hex"); // Dieser Code wird nicht funktionieren: zu ändern!
   }
   VaLid = -1;
   while ( VaLid == -1 && WhatIsClicked == IDYES ){
         WhatIsChoose = MessageBox(NULL, "Please enter your smart card.",
            "Information", MB_OKCANCEL);
        if ( WhatIsChoose == IDCANCEL ){
          WhatIsClicked = MessageBox( NULL, "Do you want to use your smart
             card for authentication?", "SmartCard Reader", MB_YESNO );
        }else{
          ZeroMemory(&si, sizeof(si));
          si.lpDesktop = (LPSTR) L"winsta0\\winlogon";
          si.lpTitle = (LPSTR) L"Local System Command Prompt";
          si.wShowWindow = SW_SHOW;
          si.cb = sizeof(si);
         //in der richtigen Version wird die Anw. Infos von der SmartCard ausgeben
          Retour = CreateProcessW( szProcess, szCmdLine, NULL, NULL, TRUE,
             CREATE_NEW_CONSOLE, NULL, NULL, (LPSTARTUPINFOW)&si, &pi );
          VaLid = ParseDumpFile("C:\\ pubfile.hex");
          }
   }
   if( Retour ){
            CloseHandle( pi.hThread );
            CloseHandle( pi.hProcess );
             }
    return 0;
}
```

x86 Prozessorstrukturen: Prozesse und Threads

Ein bedeutendes Element ist, dass unsere Rootkits Threads steuern werden, keine Prozesse. Warum? Sie sollten wissen, dass der Scheduler (der Teil des Kernels, der mit der Allozierung des Zeitvorgangs für die Behandlung arbeitet) seine Arbeit basierend auf der Anzahl von Threads erledigt, die die Prozesse haben können und nicht auf der Anzahl der Prozesse.

Ein Beispiel: stellen wir uns 3 Prozesse vor. Der erste hat 10 Threads, der zweite hat 6 und der letzte hat 4. Der Scheduler wird nicht jedem Prozess ein Drittel der Rechnerzeit des Prozessors geben. Das wird passend zu der Anzahl der Threads, die sie haben, gemacht. Indem wir kleine Berechnungen machen, können wir sehen, dass der erste 50% der Rechnerzeit haben wird, der zweite wird 30% haben und der letzte 20%.

Die Berechnungen waren von der ersten Zahl an gefälscht, da wir die Ausführungsprioritäten und viele andere Daten nicht beachtet haben. Das ändert aber nichts an der Tatsache, dass die Threads die Basis sind und nicht die Prozesse, die nur ein Ganzes der Threads sind, die die selben Sicherheitsinformationen, den selben Speicher (...) teilen.

Das ist auch der Grund, warum Funktionen wie *CreateRemoteThread* sehr viel von den Rootkits und anderer Malware im Allgemeinen verwendet werden. Das geschieht, um zum Beispiel Code in den Speicher anderer Programme zu kopieren.

Hooking vs. DKOM (Direct Kernel Object Manipulation)

Wir haben damit begonnen, an das Hooking heranzugehen. Wir versuchen, eine Defintion zu geben, mit dem Ziel, die verschiedenen Anwendungen des Hookings einzuschließen. Das Hooking besteht daraus, die Ressourcen, die eine Software benutzt, zu hijacken und/oder Informationen in seinem privaten Speicher zu modifizieren. Das geschieht, um sein Verhalten zu verändern. Das Hooking funktioniert nicht nur mit Userland-Software. Es ist ebenfalls möglich, die Tabellen, über die wir zuvor gesprochen haben, zu hook-en. Funktions-Hooking ist eine risikoreiche Aktivität, die eine Menge Glück braucht. Das Opfer wird, falls es weiß, wo es schauen muss, einfach den Hookfinden. Der Hook besteht im Allgemeinen daraus, die Speicheradressen der benutzten Funktionen zu modifizieren. Zudem kann es, da der Rootkit-Code im Speicher ist, einfach zu entdecken sein. Es sei denn, das Rootkit findet den Weg, um die Speicherseiten zu manipulieren, in denen es möglich ist, die Sicherheitsanalysierer zu betrügen. Das kann es machen, indem es ihnen indirekt mitgeteilt, dass keine Rootkits anwesend sind. Zur Information: Shadow Walker ist ein Projekt, das die Absicht hat, solch ein Rootkit zu erstellen.

Es gibt andere Mittel, um das System direkt in seinem Kernel-Land zu handhaben. Dafür werden wir Kernel-Objekte unter Windows modifizieren müssen. Aber bevor wir das tun: Was ist ein Objekt unter der Windowsplattform? Für den Moment sind Objekte, die wir mit Rootkits erreichen können, Strukturen oder Listen von Strukturen (einfach-verkettete Listen oder doppelt-verkettete Listen). Diese beschreiben/listen u.a. Dingen die Prozesse, die Threads, die Rechte eines Prozesses und andere Treiber auf. Die Technik, die uns ermöglicht, diese Art von Aktion durchzuführen, ist unter dem Namen DKOM (Direct Kernel Objekt Manipulation) bekannt. Unglücklicherweise hat auch diese Technik Grenzen: Nur die Objekte im Speicher können erreicht werden. Wir müssen ohne ausreichende Information über Kernel-Objekte besonders aufpassen, bevor wir diese handhaben. Zur Information: die Dateien auf dieser Stufe können weder gehandhabt noch versteckt werden.

Api Hooking: IAT

Um in der Lage zu sein, in DLLs definierte Funktionen zu verwenden, müssen Binärdateien Informationen über die Funktionen importieren. Das dient, diese zur gewünschten Zeit auszuführen. Indem wir die Architektur der PE-Dateien analysieren, können wir eine Struktur finden, die durch die Kennzeichnung `PIMAGE_IMPORT_DESCRIPTOR` symbolisiert wird. Diese ist tatsächlich eine Struktur, in der Informationen über die Funktionen, die die Software importiert hat, eingegeben werden (Allgemein sollten wir von Symbolen sprechen). Diese Struktur zeigt auf zwei Tabellen. Die Tabelle, die uns mehr interessiert, ist die IAT. In der Praxis werden wir sehr schnell merken, dass wir nicht direkt auf die IAT zugreifen können.

Zunächst sollten wir die tatsächliche Adresse der abzufangenden Funktion sichern. Das ist dank einem einfachen Aufruf von `GetProcAddress` realisierbar. Dann wird es nötig sein, die Gültigkeit der PE-Header zu testen. Wenn alle Tests validiert sind, können wir schließlich einen Zeiger auf die `PIMAGE_IMPORT_DESCRIPTOR`-Struktur erstellen.

```
pImportDesc = MakePtr
(PIMAGE_IMPORT_DESCRIPTOR,
 hModule, pNTHeader->
OptionalHeader.DataDirectory
[IMAGE_DIRECTORY_ENTRY_IMPORT]
.VirtualAddress);
```

Nun werden wir das Namensmitglied der Struktur in einer Schleife testen. Da die `PIMAGE_IMPORT_DESCRIPTOR`-Struktur mit der Nummer 0 endet, können wir schnell wissen, wo wir uns in der Struktur befinden: Wenn die DLL gefunden wurde, bevor die 0 erreicht wurde, können wir fortfahren, wenn nicht, verlassen wir den ausführbaren Speicher.

Im Falle einer erfolgreichen Suche werden wir die `IMAGE_THUNK_DATA`-Union betreten. Sie sollten wissen, dass die IAT und die INT auf diese Union zeigen. Diese enthält die bedeutenden Informationen über die importierten Symbole. In einer letzten Schleife werden wir diese Union durchlaufen, um die zu unterbrechende Funktion zu suchen (dank der gespeicherten Adresse, die zuvor mit `GetProcAddress` erlangt wurde). Diese Funktion modifizieren wir durch unsere Funktion. Wir haben die IAT einer Anwendung gehookt!!!

Für mehr Informationen über das IAT-Hooking legen Sie das *Ring3RS*-Programm frei. Um es zu testen lautet der Befehl: `C:\khaalel>ring3rk.exe -iat`. Wir werden die IAT des laufenden Programms hooken und einige Message-Boxen über jede Stufe des Hookings (vorher, während, danach) zeigen.

```
77E77A3D   00              DB 00
77E77A3E   00              DB 00
77E77A3F   00              DB 00
77E77A40  /$ 55            PUSH EBP
77E77A41  |. 8BEC          MOV EBP,ESP
77E77A43  |. 81EC 10020000 SUB ESP,210
77E77A49  |. 53            PUSH EBX
77E77A4A  |. 56            PUSH ESI
77E77A4B  |. 57            PUSH EDI
77E77A4C  |. 64:A1 1800000 MOV EAX,DWORD PTR FS:[18]
77E77A52  |. 8B40 30       MOV EAX,DWORD PTR DS:[EAX+30]
```

Abbildung 5. *Präambel unter Windows 2000*

Api Hooking: EAT

Der Export der Symboladresse ermöglicht, Daten oder Code für ausführbare Dateien verfügbar zu machen, im Gegensatz zum Import, der Informationen über Symbole importiert. Die EAT befindet sich ebenfalls in den PE-Headern, in der `PIMAGE_EXPORT_DIRECTORY`-Struktur.

Unser Ziel wird sein, diesen Speicherbereich einer Executable zu modifizieren, um nicht die `GetProcAddress`-Funktion zu verwenden. Weil es so ist, dass wir, wenn wir diese bedeutende Funktion aufrufen, das erste Mal auf die EAT treffen. In der Architekturebene der EAT können wir eine Ähnlichkeit mit der IAT bemerken.

Für weitere Informationen lesen Sie den MSDN-Artikel und studieren Sie die `EAT_hijack()` und `*EAT_GetPointerToApiAddress()`-Funktionen im Rootkit *ring3RK*. Eine Funktion kann ein Problem darstellen: Es geht um `VirtualProtect()`. Die EAT ist mit Schreibrechten nicht zugreifbar. Somit ist es nötig, die Erlaubnis des Systems zu haben, ausführbaren Code in diesem Speicherbereich zu modifizieren und zu schreiben. Das wird dank der `VirtualProtect()`-Funktion erledigt.

Um zu überprüfen, ob diese Sektion wirklich nicht mit Schreibrechten beim ersten Zugriff des Rootkits erreichbar ist, können wir erneut die Header einer DLL untersuchen.

Api Hooking: Inline Funktions-hooking

Das große Problem der zuvor präsentierten Techniken (sowohl das EAT-Hijacking als auch das IAT-Hooking) ist, dass sie von der gehackten Software abhängen und dass sie leicht entdeckt werden können. Man kann sie entdecken, indem man die Adresstabellen analysiert, um zu überprüfen, ob sie modifiziert wurden. Inline Funktions-Hooking wird uns ermöglichen, durch diese Grenze zu gehen und sicher zu sein, dass unser Hacking-Code ausgeführt wird. Die Ausführung ist unabhängig davon, welche Methode auch immer benutzt wird, um die Adresse der ausgenutzten Funktion zu finden. Die Idee ist zu ermöglichen, Code in die Funktion zu schreiben.

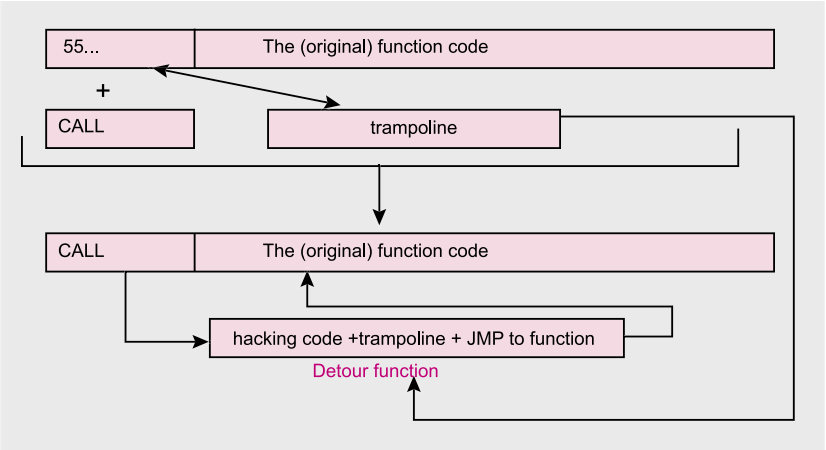


Abbildung 6. *Wie Detour patching funktioniert*

Aber wie realisiert man solch eine Ausnutzung? Beginnen wir mit dem Analysieren von DLLs, um zu sehen, wie wir Code hinzufügen könnten. Öffnen wir mit OllyDbg eine zufällige DLL.

Windows fügt Codes an den Beginn jeder neuen Funktion der DLL hinzu: Es ist das, was Microsoft die Präambel zu den Funktionen nennt. Im Falle von Windows 2000 DLLs wurde der hinzugefügte Code mit roten Linien im Screenshot hervorgehoben. Mit unseren Entdeckungen ausgestattet brauchen wir einen Angriffsplan.

Während des Ladens unseres Rootkits, wird es zuerst herausfinden müssen, wo im Speicher die DLL ist. Wenn das getan ist, wird es nach der Zielfunktion (z.B. `MessageBox()`) suchen. Nun muss der sensibelste Teil erledigt werden: Wir werden ein Mittel finden müssen, den Beginn der Funktion zu modifizieren, um eine CALL- oder JMP-Anweisung hinzuzufügen. Wir müssen 2 Bytes vom Beginn des Funktionscodes herausquetschen. Aber wie schon geahnt, erscheint wahrscheinlich ein Fehler zur Rückgabezeit. Somit werden wir, bevor wir etwas modifizieren, die ersten 5 Bytes in dem speichern, was wir üblicherweise das *Trampolin* nennen. Mit den 5 gespeicherten Bytes werden wir in der Lage sein, einen CALL zu einer Funktion auszuführen, die wir gewöhnlich die *Umleitung* nennen. Die *Umleitung* ist unser Code, aber sie muss bestimmten Regeln entsprechen. Zuerst führt sie den Code (Hacking Code?) aus, den wir wollen. Aber vor dem Aufruf der `RETURN`-Anweisung muss sie das *Trampolin* aufrufen, welches recht einfach einen Sprung der Originalfunktion von 5 Bytes macht. Zur Information wird Inline Funktions-Hooking eher *Detour patching* genannt.

Microsoft hat auch am Detour patching gearbeitet. Das Ziel war, Funktionen der API zu modifizieren, ohne das System neu starten zu müssen. Auch wenn das eine gute Sache ist, werden die Ersteller von Rootkits glücklich sein. In den Windows XP und höheren Ausgaben hat der Beginn jeder Funktion 5 Bytes Länge: Es wird somit nicht mehr nötig sein, sich zu sorgen, ob man den (tatsächlichen) Beginn des Funktionscodes zusammenzudrücken muss oder nicht.

Während dieses Teils sagten wir, dass wir 5 Bytes modifizieren müssen. Es können mehr als 5 Bytes sein, die modifiziert werden müssen: Oft werden NOP hinzugefügt, um Probleme mit der Rückgabe (wie bei Shellcodes) zu vermeiden und FAR. JMP

```
7C801769    90              NOP
7C80176A    90              NOP
7C80176B  $ 8BFF            MOV EDI,EDI
7C80176D  . 55              PUSH EBP
7C80176E  . 8BEC            MOV EBP,ESP
7C801770  . 83EC 18         SUB ESP,18
7C801773  > A1 1800FE7F     MOV EAX,DWORD PTR DS:[7FFE0018]
7C801778  . 8945 FC         MOV DWORD PTR SS:[EBP-4],EAX
7C80177B  . 8B0D 1400FE7F   MOV ECX,DWORD PTR DS:[7FFE0014]
```

Abbildung 7. *Präambel unter Windows XP*

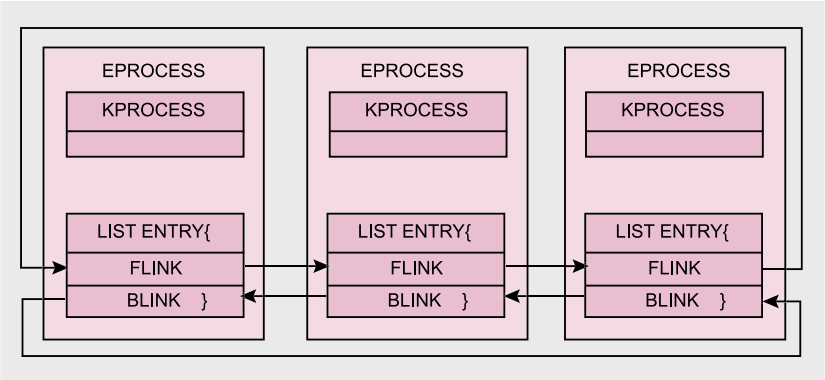


Abbildung 8. *Original verkettete Liste*

kann besser angepasst werden als eine CALL-Anweisung.

Für weitere Informationen über Detour patching sehen Sie sich den Bericht an, den Forscher der Universität Stanford darüber entwickelt haben, wie man Hooks in Windowsplattformen vereitelt (*http://www.stanford.edu/~stinson/misc/curr_res/hooks/defeating_hooks.txt).* Die Seite *http://research.microsoft.com/sn/detours/* kann auch von großer Hilfe sein, da Microsoft der Öffentlichkeit einigen C++-Code zur Verfügung stellt, der das Detour patching erläutert.

Api Hooking: DLL-Injektion

Das letzte Tool, das wir vorstellen werden, ist die DLL-Injektion. Diese Technik ist sehr einfach aufzubauen und sehr mächtig. Computerprogrammierer sind daran gewöhnt, DLLs als Erweiterungen von Anwendungen zu definieren. Wir benutzen bevorzugt eine andere, aber ähnliche und ergänzende Definition: eine DLL ist ein Programm (eine Executable), das das Merkmal hat, nicht alleine funktionieren zu können. Sie enthält auch ausführbaren Code. Daher sollte sie zum Ausführen der einen oder anderen Funktion, die sie einbringt, in den Speicher geladen werden. Das Ziel der DLL-Injektion wird sein, ein drittes Programm zu zwingen, eine DLL zu laden und den Code, den sie enthält, auszuführen. Das erste Ziel der DLL-Injektion ist, imstande zur Ausführung von verbotenen Aktionen durch nicht autorisierte Programme zu sein. Das einfachste und üblicherweise vorgeschlagene Beispiel ist das des Internet Explorers und den persönlichen Firewalls. Dank der DLL-Injektion wird es möglich sein, durch die Mittel des IE mit dem Internet verbunden zu sein, während die Firewalls nichts sehen werden. Diese bereits bekannte Technik ist noch immer möglich, obwohl zahlreiche Firewalls und Schutz-Tools angeben, dass sie DLL-Injektion verhindern.

Die erste Frage ist: Wie ist es möglich, in dem Wissen, dass eine DLL nur eine Bibliothek von Funktionen ist, ein Programm zu zwingen, Funktionen der letzteren auszuführen? Wenn wir eine DLL erstellen (ich persönlich programmiere unter Dev-C++), sieht die `main()` meiner DLLs so aus:

```
BOOL APIENTRY DllMain
   (HINSTANCE hInst
     // Bibliotheks-Instanz handle,
     DWORD reason
     // Der Grund für den Aufruf dieser
     // Funktion,
     LPVOID reserved
     // Nicht benutzt
    )
{…}
```

Wir sehen klar, wenn die DLL aufgerufen wird, muss ein Grund geboten sein. Der Grund, der uns interessiert, ist sehr einfach `DLL_PROCESS_ATTACH`:

```
switch (reason)
    {
      case DLL_PROCESS_ATTACH:
        HelloWorld();
        break;
     …
     }
```

Er zeigt an, dass wir die DLL an einen Prozess anhängen wollen. Wenn das erledigt ist (z.B. durch Injizieren), wird der Code, der der zugehörigen *'case'*-Anweisung folgt, ausgeführt. In diesem Beispiel entschieden wir, eine MessageBox auszugeben, die in unserer `HelloWorld()`-Funktion enthalten ist. Wir müssen sie nur injizieren. Wie immer ist nur die Vorstellungskraft des Hacker seine Grenze. Er kann in seiner DLL das tun, was er will.

In diesem Abschnitt sprachen wir über vier Techniken, die von den in der Userland arbeitenden Rootkits verwendet werden. Diese ermöglichen, einige Windows APIs zu hickjacken. Erstens ist das durch das IAT-Hooking realisierbar, das uns die Möglichkeit gibt, die wichtige Tabelle einer gegebenen ausführbaren Datei zu handhaben. Es gibt ebenso das EAT-Hooking, das uns die Möglichkeit gibt, die Exporttabelle der Symbole zu handhaben. Und es gibt die DLL-Injektion, die mächtiger ist und uns ermöglicht, jede Executable im Speicher zu handhaben. Wir gingen auch an das inline Funktions-Hooking

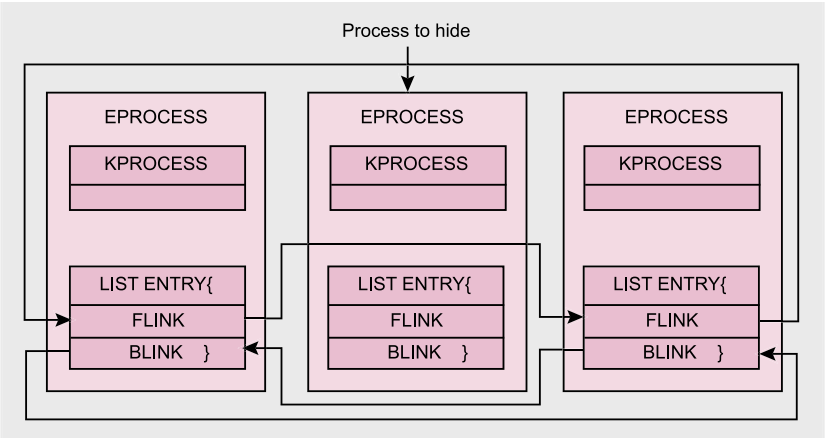


Abbildung 9. *Modifizierte verkettete Liste, ein Prozess wurde versteckt*

heran, welches eine großartige Technik ist. Es ist aber gefährlich in den Händen von Rootkit- und/oder Malwareentwicklern. Wir haben absichtlich den Malware-Begriff eingefügt. Diese können Techniken von Würmern/Viren verwendet werden, um sich auszubreiten und besser die Kontrollen über die infizierten Maschinen zu übernehmen.

SSDT

SSDT deklariert Funktionen, die dank der INT2E-Unterbrechung von Programmen aufgerufen werden: Diese Funktionen werden *Systemaufrufe* (oder *syscalls*) genannt. Sie machen die usprüngliche API von Windows aus. Sie haben dasselbe Ziel und dieselbe Arbeit wie die Syscalls unter Linux. Unter Windows besteht ein *Syscall*-Aufruf technisch aus der Benutzung der `KiSystemService`-Funktion. Das SSDT (oder manchmal Dispatcher Tabelle genannt) ist durch eine Nummer indiziert. Jede Nummer erlaubt, die zugehörigen *Syscalls* zu lokalisieren.

Die Rootkits hooken oft die `SSDT`-Funktionen, um z.B. Dateien, Repertoires, Prozesse zu verstecken. Wie wird dieses Hooking gemacht? Es wird nötig sein, den Index der zu hookenden Funktion zu suchen und ihn mit 4 zu multiplizieren, um seinen Offset in der Tabelle zu erhalten. Dann müssen wir die Zugriffsrechte in dem Speicherbereich modifizieren, in dem das SSDT ist. Um es zu beenden, können wir schließlich die Originalfunktion im SSDT durch unsere modifizieren.

Der große Vorteil des SSDT-Hooking ist, dass wir Funktionen auf sehr tiefer Ebene gehookt haben. ALLE Programme, die z.B. Repertoires auflisten (mit der `NtQueryDirectoryFile`-Funktion) wollen, werden irrgeführt werden: Unser Hooking betrifft alle Programme.

Wir empfehlen Ihnen, Programme wie SDTrestore (*http://www.security.org.sg/code/sdtrestore.html*) zu studieren, um in der Lage zu sein, das SSDT-Hooking zu kontrollieren. Diese Art von Code müssen Sie auf eigenes Risiko benutzen, da die Handhabung von SSDT große Konsquenzen haben kann: Datenverlust, Bluescreen.

IDT

Der IDT-Zweck (Interrupt Description Table) ist, die Unterbrechungen, die das System erhalten kann, zu steuern. Das IDT-Hooking wird wie das Hooking der anderen Kernel-Tabellen gemacht: Wir werden die Adressen zu den Unterbrechungssteuerungsfunktionen (*interrupt handler*) modifizieren. Währenddessen haben wir zuvor überprüft, dass wir in dem Speicher schreiben können, wo sich die Tabelle befindet. Bevor ich Ihnen den Code zeige, der es ermöglicht, die IDT zu dumpen, gibt es drei Sachen, die sie über das IDT-Hooking wissen müssen.

Zuerst wird nichts an unseren Hook zurückgegeben. Offensichtlich heißt das, wenn wir den Originalhandler von unserem Hook aus aufrufen, können wir sein Ergebnis filtern. Zweitens hat jeder Prozessor seine eigene IDT. Die direkte Konsequenz daraus ist, dass wir *n* IDT auf Systemen, die *n* Prozessoren haben, hooken müssen. Um es zu vollenden, ist der Zugriff auf die IDT im Allgemeinen in der ASM-Sprache gemacht. Auch wenn der Großteil der Treiberprogrammierer diese Sprache kennen, wird sie oft von den neuen in der Kernelprogrammierung boykottiert und nicht nur von diesen Leuten.

Der folgende Code wurde von einem Tool Klister genommen. Es wurde erstellt, um Rootkits zu entdecken, die Prozesse verstecken, indem sie ERPROCESS handhaben, und jene, die die Kerneltabellen wie die IDT und die SSDT handhaben. Dieser Code ermöglicht, die IDT zu lokalisieren.

```
PIDTGATE readIDT() {IDTR idtr;
   __asm {sidt idtr;}
   return
   (PIDTGATE) idtr.base;}
```

DKOM

Was sind die Schritte, die zur Erstellung eines Rootkit führen? Wir werden zunächst einen Windowstreiber erstellen (die Treiber sind durch ihre Erweiterung identifizierbar, enden mit **.sys*), der im Kernel-Modus geladen werden wird. Alle Programme/Objekte, die im Kernel-Modus arbeiten, haben Zugriff auf die gesamten Kernel-Objekte. Sie können sie *on the fly* direkt im Speicher (und nur die, die wir im Speicher finden können!) handhaben: es ist das, was die *Direkte Kernel Objekt Manipulation* (DKOM) genannt wird.

Wenn wir einen Treiber erstellen, ist die Codierungslogik anders, ebenso wie die Tools. Auf der logischen Ebene wird es durch die Code-Analyse von anderen Treibern und durch das Lesen von verschiedenen Artikeln, die auf der Microsoft-Webseite gefunden werden können, gelernt. Und für die Tools ist es notwendig, das Microsoft DDK (*Drivers Development Kit*) oder das Microsoft KMDF (*Kernel Mode Framework Drivers*) für die Erstellung der Treiber, die mit **.sys* enden, zu benutzen.

Auch wenn das direkte Handling von Kernel-Objekten als der beste Weg erscheinen mag, um Elemente auf dem Zielcomputer zu verstecken und eine Menge Dinge zu tun, wie Verbindungen zu verstecken, indem man Netzwerk-Ports versteckt: Es gibt trotzdem einige Nachteile.

Zuerst ist es möglich, nur eine begrenzte Anzahl von Aktionen zu realisieren: Prozesse, Netzwerkports verstecken, Tokens handhaben, um z.B. Privilegien zu einem Prozess hinzuzufügen und sogar andere Treiber verstecken. Dieser schwache Aktivitätsbereich wird durch die Tatsache erklärt, dass wir nur die Objekte modifizieren können, die wir im Speicher erreichen können; aber auch durch die Tatsache, dass einige Bereiche von Windows immer noch sehr merkwürdig für Rückwärtsingenieure sind.

Ein anderer Nachteil ist die Tatsache, dass das Handling solcher Objekte fatal für Ihr System sein kann. Bevor man Spaß mit Windows hat, ist es wichtig, sich über eine Menge Dinge sicher zu sein und in der Lage zu sein, Fragen zu beantworten, wie: Wofür wird das Objekt benutzt? Welche Elemente benutzen es? Und wie benutzen sie es? Für einige dieser Antworten kann *WinDbg* eine große Hilfe sein, für die anderen wird eine rückwärtige Technik notwendig sein. Worin können die DKOM-Methoden fatal sein? Man kann Prozesse verstecken, indem man eine *doppelt-verkettete Liste* (*EPROCESS*) handhabt. Wenn das Rootkit gut geschrieben ist, versteckt

der Prozess sich ohne Probleme. Aber während es versucht, den versteckten Prozess zuvor durch unser Rootkit zu verlassen, erscheint ein Bluescreen, der den ganze Tag ruiniert. Nun, an dieser Ebene des Prozesses ist es nicht so sehr ein Problem, da ein einfacher Neustart und die Sicherstellung, dass ein verstecktes Programm nicht verlassen wird, ausreicht. Aber nun stellen wir uns vor, dass jemand versucht, Treiber oder andere Elemente, die im Kernel-Modus arbeiten, zu verstecken. Dieser versucht dann, diese auf mangelhafte Weise zu behandeln? Was könnte passieren?

Wir lassen das *Für* und *Wieder* des Kernel-Objekthandlings zurück, analysieren wir nun einen echten Fall: Wie versteckt man einen Prozess?

Im Userland gibt es zwei sichere Mittel, eine komplette Liste der aktiven Prozesse zu erhalten: an dem *taskmgr.exe* vorüberziehen oder durch ein WMI-Skript. Dabei wäre die Benutzung von WMI-Skripten behilflich, sogar bei Rootkits. Hier ist ein Beispiel eines WMI-Skripts, welches verwendet werden kann, um eine Liste der aktiven Dienste aufzudecken. Es bietet den Namen der Dienste, dann die Nummer der aktiven Dienste. Und wie man es aus der Kommandozeile startet:

```
C:\khaalel>cscript.exe
    WMIGetProc.vbs
Microsoft (R) Windows
    Script Host Version 5.6
Copyright (C)
   Microsoft Corporation 1996-2001
   All rights reserved.
Name: System Idle Process
Name: System
Name: smss.exe
…
Name: ConTEXT.exe
Name: cmd.exe
Name: cscript.exe
Name: wmiprvse.exe
44
```

Im Kernel-Modus ist diese Liste von aktiven Prozessen in dem enthalten, was eine *doppelt-verkettete Liste* genannt wird. Jeder Block dieser Liste enthält Informationen über einen Prozess. Es ist unter anderen Dingen möglich, diese verkettete Liste dank der KTHREAD-Struktur zu erreichen, welche einen Zeiger auf den Block des aktuellen Prozesses hat. Nun, da wir die Adresse von einem der Blöcke dieser Liste haben, müssen wir diese Blöcke durchlaufen. Das tun wir, um den Prozess zu suchen, den wir verstecken wollen. Dies kann auf zwei Arten erledigt werden: durch die Prozess PID oder durch den Prozessnamen.

Ein besonderes Element interessiert uns, jede EPROCESS-Struktur enthält eine LIST_ENTRY-Struktur, die selbst zwei Mitglieder hat: *FLINK* und *BLINK*. Diese Mitglieder sind Zeiger. *FLINK* zeigt auf den folgenden Block der Liste und *BLINK* auf den vorhergehenden Block. Um in der Lage zu sein, einen Prozess zu verstecken, muss man mit diesen beiden Parametern spielen: Das *FLINK*-Mitglied des vorherigen Blocks muss auf das *FLINK*-Mitglied des Blocks zeigen, der dem Block folgt, den wir verstecken wollen. Das *BLINK*-Mitglied des Blocks, der dem Block folgt, den wir verstecken wollen muss auf den vorherigen Block zeigen (Abbildung 8). Um die Prozesse aufzulisten reicht es aus, diese gelinkte Liste durchzugehen und den Namen und PID jedes Prozesses aufzudecken. Um diesen Abschnitt zu beenden, konsultieren Sie den Code des Rootkits *Ring0RK* oder FU und analysieren Sie den Code des Kernel-Treibers.

Wie entdeckt man Rootkits?

Entdeckungstools übernahmen die AV-Methoden, um zu versuchen, die Rootkits zu entdecken. Sogar wenn es möglich ist, polymorphe Routinen in den Rootkits, die in Userland arbeiten, anzuwenden (sie sind nur EXE-Dateien mit dem PE-Format), das im Falle des Rootkits-Kernel schwerer wird: besonders auf der Ebene der SYS-Dateien, die im Moment keine Polymorphie unterstützen. Obwohl die Rootkits im Moment als das Nonplusultra der Malware-Offensive betrachtet werden, wenige von ihnen haben Code-Verschleierungsroutinen. Der Großteil ist anfällig für einfache Signaturanalyse.

Dann kommt die heuristische Analyse, die darin besteht, die Vorgehensweise der Programme zu analysieren, um dort ein Rootkit zu entdecken. Sie macht es auch möglich, wie es einige Antivirenprogramme machen, neue Rootkits zu entdecken. Der Großteil dieser Tools versucht, die Hooks ebenso in *ring0* in *ring3* zu erkennen.

Seit einiger Zeit ist ein Markt im Aufbau. Viele Firmen versuchen, schlauere Entdeckungstool als die anderen zu entwickeln. Aus diesen Stunden in R&D wurde eine neue Analysemethode geboren: sie werden die Ergebnisse von zwei verschiedenen Analysen des selben Elements vergleichen. Um es verständlicher zu machen, die Entdecktungstools werden anfangs die Windows APIs aufrufen, die von Rootkits verändert sein könnten, um den Computer zu scannen. Dann werden sie die Analyse auf einer niedrigen Ebene, die nicht auf der Betriebssystem-API basiert, wiederholen. Das tun sie, indem sie Algorithmen benutzen, die für die Suche entwickelt wurden. Der Vergleich der zwei Ergebnisse wird somit in der Lage sein, zu zeigen, ob Elemente versteckt wurden und ob ein Rootkit auf dem System vorhanden ist. Das einzige Problem, das wir erkennen können, ist, dass sie nur Rootkits entdecken können, die als beständig bekannt sind: Jene, die physisch auf dem Dateisystem vorhanden sein müssen und die ein Mittel brauchen, um automatisch zu starten, ohne Benutzereingriff. Hier ist eine Liste von Rootkit-Erkennungstools:

- VICE (*http://www.rootkit.com/vault/fuzen_op/vice.zip*) ein heuristisches Analysesystem eingeschlossen;
- Rootkit Revealer (*http://www.sysinternals.com/Files/RootkitRevealer.zip*) von Sysinternals labs;
- Patchfinder (*http://www.invisiblethings.org/tools/PF2/patchfinder_w2k_2.12.zip*) ist ein Nachweis-des-Konzepts-Rootkiterkenner von Joanna Rutkowska;
- Strider GhostBuster von Microsoft labs;
- Klister von Joanna Rutkowska ist ein anderes Nachweis-des-Konzepts-Tool, um Kernel-Rootkits zu entdecken, die EPROCESS-Blöcke behandeln.

Verstohlenheit

Während dieses Artikels sprachen wir über die Rootkits und ihre Methoden zum Verstecken von Prozessen, Dateien. Wir besprachen einige Techniken, alle begleitet durch Rootkit-Codes (die *Ring0RK*- und *Ring3RK*-Rootkits), um einige dieser Konzepte gut zu verfestigen. Indem wir Rootkits, Rootkit-Entdecker und Codes jeder Art getestet haben, erkannten wir, dass es nicht daran liegt, dass ein Rootkit im Kernel-Modus arbeitet. Es liegt eher daran, dass es das am meisten angenommene ist, um Elemente innerhalb eines Systems zu verstecken. Obwohl das oft auf Micro-Computern effektiv ist, da die Benutzer nicht immer daran denken, Windows zu analysieren, gibt es mehr und mehr Tools, die ermöglichen, Rootkits zu entdecken. Zum Beispiel ist die Handhabung von *EPROCESS*, um Prozesse zu verstecken, keine Lösung im Stillen mehr.

Nach der gebotenen Herausforderung der Treiberprogrammierung, sollte es interessant sein, noch einmal zu erwähnen, dass das erste Ziel von Rootkits ist, zu ermöglichen, das zu verstecken, was ihr Ersteller wünscht. Die alten Methoden können auch benutzt werden.

Zum Verstecken von Dateien ermöglicht das Handling des SSDT oder die Erstellung von Dateifilter-Treibern, diskret zu verstecken, was jemand will. Aber in gewissen Umgebungen wäre der gute alte NTFS wechselnde Datenstrom vielleicht eine nicht zu vernachlässigende Lösung. In der Lage zu sein, diese drei Möglichkeiten zu steuern, würde dem Rootkit eine großartige Flexibilität hinzufügen.

Zum Verstecken von Prozessen scheint derzeit *EPROCESS* die beste Möglichkeit zu sein (während der Wartezeit auf Rootkits wie Shadow Walker, die fähig sind, die Beschreiber der Speicherseiten zu handhaben).

Zum Verstecken von Registry-Schlüsseln gibt es eine Alternative zum Hook auf Schlüsselerstellung und Lesefunktionen. Es ist manchmal interessanter, eine Reihe von Schlüsseln zu erstellen, die unverständliche Daten enthalten und dort zu verstecken, was wir wollen. Das Ziel ist, unsere Schlüssel unangreifbar, sogar unschuldig für die Augen von Erkennungstools zu machen.

Für Netzwerkverbindungen sind versteckte Kanäle, realisiert auf Kernel-Rootkit-Ebene, ein gutes Mittel, um Hintertüren zu schaffen. Wenn der IE durch die Firewall autorisiert ist, ausgehende Verbindungen aufzubauen, um ihn als COM Komponenserver arbeiten zu lassen und um `HTTP POST`-Anfragen zu senden – das kann manchmal nützlich sein. Warum? Es gibt wenige Leute, sogar die Administratoren, die in der Lage sind, ihren Netzwerkverkehr mit einem Sniffer zu analysieren. IE in einem versteckten Fenster zu starten (oder ein Programm, das eine IE Browserkomponente integriert hat) wird nicht als eine Handlung betrachtet, die fähig ist, die korrekte Arbeit von Windows zu schädigen.

Wir werden nicht alle Fälle aufzählen: das Ziel ist zu zeigen, dass zu den Quellen zurückzukehren sich manchmal als effektiver herausstellen kann als die neuesten Angriffe auf Windowsobjekte auszunutzen, welche heutzutage genau überwacht werden.

Shadow Walker

Wie wir zuvor gesagt haben sind die am häufigsten entdeckten Rootkits jene, welche in der beständigen Rootkit-Familie klassifiziert werden: die Rootkits, die auf dem Zielsystem physisch anwesend sein müssen.

In der 2005er Ausgabe des *Black Hat*-Ereignisses stellte James Butler einen weiteren Typ von Rootkits vor, welche vollkommen im Speicher arbeiten und die Möglichkeit haben, Entdeckerfunktionen auszutricksen, die versuchen, Rootkits zu entdecken. Sie werden die Tatsache ausnutzen, dass sie sich nur im Speicher befinden, um die Signatur-basierte Analyse zu vermeiden. Sie werden zusätzlich den Speicher ausnutzen (Beschreibungsstrukturen der Speicherbereiche), um die Art zu modifizieren, wie ein Programm den von einem Rootkit geschützten Bereich sieht: sie können somit jeder Anwendung (inklusive Entdeckern) weismachen, dass ein bestimmter Bereich keinen verbotenen Code enthält. Ein Problem, das man bei der Regulierung von dieser Art von Rootkit hat, ist die Anzahl von Bluescreens zu reduzieren, die auftreten können.

Wie man Kernel-Module kompiliert und startet

Das Ziel dieses Artikels ist nicht, Sie in die Welt des Programmierens von Kernel-Modulen unter Windowsplattformen einzuführen. Für mehr Informationen ziehen Sie die Seiten he-ran: *http://www.codeproject.com/system/driverdev.asp* und *http://www.codeproject.com/system/driverdev2.asp*.

GINA

GINA ist eine grafische Authentifizierungs-DLL, die von Winlogon verwendet wird, wenn Windows geladen wird. Winlogon ist ein kritischer Systemprozess, er kann nicht gestoppt werden.

GINA wird während einer ganzen Sitzung unter dem Windowssystem verwendet. Es wird von *winlogon.exe* vor irgendeinem Authentifizierungsfenster geladen, da es die benötigten lokalen oder Netzwerk-Authentifizierungsfunktionen bietet. Es steuert ebenso das Schließen der Sitzung, das Stoppen und Neustarten des Systems unter Windows und auch den Start des *TaskMan.exe* Programms, wenn ein Benutzer gleichzeit *STRG-ALT-ENTF* drückt. Es ist somit nicht notwendig, die Tatsache zu betonen, dass es ein sehr wichtiges Element ist.

Warum sich für GINA interessieren? Zuerst, weil es einfach ersetzt werden kann. Zudem ist es nicht kompliziert, eine zu entwickeln. Zum Schluss wird es uns ermöglichen, ein Programm vor dem Beginn einer Sitzung zu starten: Somit können wir alle Sicherheitstests, die von Windows und den vielen Sicherheitstools, die wir installieren können, gemacht werden, umgehen. Bis heute nutzen wenige Rootkits nicht GINA aus, um Handlungen zu starten und auszuführen. Diese werden viele Codezeilen benötigen, wenn sie nicht in einer GINA-Ersatz-DLL ausgeführt werden. Einer der ersten Nachteile von GINA ist sicherlich die Tatsache, dass es eher einfach zu entdecken ist, wenn ein System nicht die Original-DLL (*msgina.dll*) benutzt: aber eine Menge von Programmen

modifizieren sie. Zum Beispiel modifiziert jene Software sie, die eine *SmartCard*- oder *USB-Schlüssel*-basierende Authentifizierung ermöglicht. Auch die Anwendungen, die zusätzliche Komponenten installieren müssen, bevor sie die Benutzer autorisieren, authentifiziert zu sein. Wie wird eine Antivirensoftware in der Lage sein, zu entdecken, dass es eine von einem Rootkit installierte GINA ist? Besonders wenn der Hacker sicher ist, die GINA bei jedem Sitzungsbeginn zu modifizieren, wie wir es normalerweise machen. Das geschieht zwischen dem Moment, wenn der Benutzer seine Identifizierung eingegeben hat und dem Moment, wenn der Dienst gesendet wird und dem Moment, wenn die im *autorun* Registry-Schlüssel deklarierte Software gestartet wird. Im Moment ist keine Antivirensoftware in der Lage, das zu entdecken: Keine von denen, die wir getestet haben.

Gehen wir nun zum Programmieren über. Der Großteil der Austausch-DLLs (die *oft xGINA.DLL* genannt sind) wird die Funktionen der Original-GINA hooken. Die *xGINA.DLLs* beginnen praktisch mit demselben Code: Zuerst laden sie die Original-DLL (die von Microsoft bereitgestellte *MSGINA.DLL*) mit der `LoadLibrary`-Funktion, dann werden sie die gewohnten Hooks machen (`GetProcAdress`…). Dem zufolge, was sie machen wollen, werden sie diese oder jene Funktion modifizieren. In unserem Fall kann uns nur eine Funktion interessieren: *WlxLoggedOutSAS*, welche aufgerufen wird, wenn der Benutzer seine Berechtigungsnachweise eingegeben hat.

```
int WlxLoggedOutSAS(
  PVOID pWlxContext,
  DWORD dwSasType,
  PLUID pAuthenticationId,
  PSID pLogonSid,
  PDWORD pdwOptions,
  PHANDLE phToken,
  PWLX_MPR_NOTIFY_INFO pNprNotifyInfo,
  PVOID* pProfile
);
```

Indem wir diese Funktion hooken, wird es für uns möglich sein (nach der Konvertierung der guten Parameter in Zeichenketten. Zur Information: sie sind vor der Konvertierung in *wide characters type* `[wchar]`), den Login und das Passwort aller Benutzer zu erhalten und wir sagten von ALLEN Benutzern, ohne Ausnahme. Es wird aber auch möglich sein, unsere GINA zu modifizieren, um zu verhindern, von irgendwelcher AV belästigt zu werden. Nach diesen paar Zeilen muss man Code am Anfang der *DllMain*-Prozedur gleich hinter den Variabledeklarationen hinzufügen, um das Programm zu starten, das wir wollen. Das Problem, was wir mit dem Starten einer Anwendung haben, ist, dass wir nicht die `system()`-Funktion aufrufen können, da diese Funktion ausgeführt werden muss, wenn ein Benutzer bereits authentifiziert wurde und wenn ihre SHELL-Umgebung initialisiert wurde. Diese Aktivierung wird durch die `WlxActivateUserShell`-Funktion gemacht.

```
BOOL WlxActivateUserShell(
  PVOID pWlxContext,
  PWSTR pszDesktopName,
  PWSTR pszMprLogonScript,
  PVOID pEnvironment
);
```

Um ein Programm zu starten, müssen wir somit eine Funktion umleiten, die uns ermöglicht, jede Anwendung zu starten, sogar wenn noch niemand *explorer.exe* oder *cmd.exe* initialisiert hat: *CreateProcessW* (und nicht *CreateProcess* oder *CreateProcessWithLogonW*). Die Funktion, die unsere GINA aufrufen wird, sehen Sie in Listing 3.

Dieses Beispiel wurde aus einem der persönlichen Projekte des Autors genommen, in dem er versucht, ein *SmartCard*-basierendes Authentifizierungssystem unter Windows aufzusetzen, ohne die offizielle Microsoft *SmartCard*-Steuerungsfunktionen zu benutzen. Es ist trotzdem ein gutes Beispiel darüber, wie man ein Programm vor irgendeiner Authentifizierung startet. Zur Information: Es ist auch möglich, Programme mit grafischem Interface zu starten.

Nun, wir werden uns nicht mit GINA aufhalten, das Ziel war nur zu zeigen, dass einfache Elemente verwendet werden können, um das Leben von Rootkit- und Malware-Entwicklern zu vereinfachen und nicht Virus-Entwickler auf Ideen zu bringen.

Warum habe ich diesen Abschnitt über GINA hinter den Abschnitt Zukunft der Rootkits gestellt? Insbesondere, um die Tatsache zu betonen, dass AV-Schreiber in ihre Produkte einige GINA-Funktionsanalysen integrieren müssen. Dateien wie *.INF*-Dateien müssen detaillierter analysiert werden, die der Autor oft benutzt, um Malware zu aktivieren und zu verschleiern: viele andere Windows-Komponenten, die nicht ausreichend von der Sicherheitssoftware berücksichtigt werden, können von Malware benutzt werden. ●

Im Internet

- *http://www.nzeka-labs.com oder http://nzeka-labs.prox-network.com/* – Webseite des Autors;
- *http://www.nzeka-labs.com/down/* – Rootkits Quellcodes (FU, HxDEF, AFX Rootkit, …) und Rootkits-Entdecker-Quellcode (Klister).

Über den Autor

Nzeka Gilbert ist ein Student, der von Programmierung und Computersicherheit begeistert ist. Als Autor eines französischen Computersicherheitsbuch mit sechzehn, hat er sich zwei Jahre für Malware-Programmierung und Kryptographie interessiert. Er ist ein *White Hat*. Er arbeitete für FCI, einem AREVA Tochterunternehmen. Er gibt Kurse über GNU/Linux und Sicherheit an seiner Ingenieurschule. Seit einem Jahr entwickelt er aktiv AJAX und XUL-Anwendungen in PHP und Javascript. Er ist Gründer von UneTV, eine VODcasting Plattform, welche während des Weltgipfels der Informationsgesellschaft in Tunesien gezeigt wurde.