Abuso dell'Hardware nell'Attacco al Kernel di Linux

Pierre Falda, Antifork Research Inc.
17 Ottobre 2006

Sommario

In questo documento mi prefiggo di esporre il design a livello teorico e pratico di un linux kernel malware che, sfruttando le peculiarità dell'hardware sottostante e del kernel di linux, abbia una portabilità ed un livello di occultamento molto elevati, tali da sfuggire a qualsiasi strumento di rilevamento deterministico esistente.

I. INTRODUZIONE

RIGUARDO la possibilità di sfruttare caratteristiche proprie dell'hardware per la scrittura di malware complesso da rilevare od, in modo equivalente, di utilizzare queste stesse caratteristiche per la costruzione di strumenti di rilevamento che mirino ad essere altrettanto potenti, si è già parlato in questi anni[1]. Alcuni lo hanno ipotizzato, spingendosi in alcuni casi fino a descrive il design di applicazioni simili, senza però che una implementazione pubblica e completamente funzionante fosse fornita. In questo documento, noi vedremo il design di un malware operante a kernel space per il sistema operativo linux che sfrutti funzionalità dello hardware per nascondersi, analizzeremo alcune porzioni di codice per mostrare come sia possibile il funzionamento di quanto descritto nella pratica, fornendo inoltre del codice perfettamente operativo e completo, cosi che il lettore interessato possa studiare nel dettaglio quanto esposto e non solo a livello macroscopico.

Nella seconda parte, dopo un breve ripasso di alcune nozioni di sistemi operativi, faremo un salto indietro nel tempo di qualche anno e guarderemo alcuni dettagli implementativi dei principali malware delle vecchie generazioni[2] per analizzarne pregi e difetti architetturali, infine concluderemo con qualche cenno sul rilevamento dei malware a kernel space. Nella terza sezione esamineremo in maniera sommaria il funzionamento di parte dell'infrastruttura di debug dei processori appartenenti alla famiglia IA-32[3] e di come linux la supporti, mentre nella quarta parte utilizzeremo quanto appreso nelle due sezioni precedenti per creare il modello di funzionamento del malware, poi osserveremo come la flessibilità di linux ci permetta di applicare il tutto in maniera rapida ed estremamente pulita. E' altamente consigliata la lettura di [2], [3] e [4] nonché avere dimestichezza col linguaggio C ed assembly per una piena comprensione di questo documento.

II. SYS CALL TABLE HACKING

Possiamo definire le syscall come le funzioni che ci vengono offerte dal kernel per interfacciarci con le sue funzionalità. Richiamandole possiamo richiedere al kernel l'erogazione di risorse o servizi: se, ad esempio, noi volessimo duplicare il processo corrente, dovremmo richiedere questo servizio attraverso la relativa syscall, ovvero la sys_fork.

La sys call table, che da qui in poi chiamerò SCT, è un array di puntatori a funzione residente in memoria kernel in cui ogni elemento punta alla relativa syscall.

Un concetto simile a questo è quello di Interrupt Descriptor Table, che chiamerò IDT: un interrupt è un evento sincrono od asincrono che altera la sequenza di istruzioni eseguita dal processore, e l'IDT è un array residente in memoria kernel i cui elementi contengono svariate informazioni, tra cui quelle riguardanti la routine di gestione del corrispondente interrupt.

Vediamo ora piu nel dettaglio cosa succede quando invochiamo una syscall da un nostro programma funzionante in user mode, tenendo come riferimento la figura 1.1 che ora andremo a commentare.

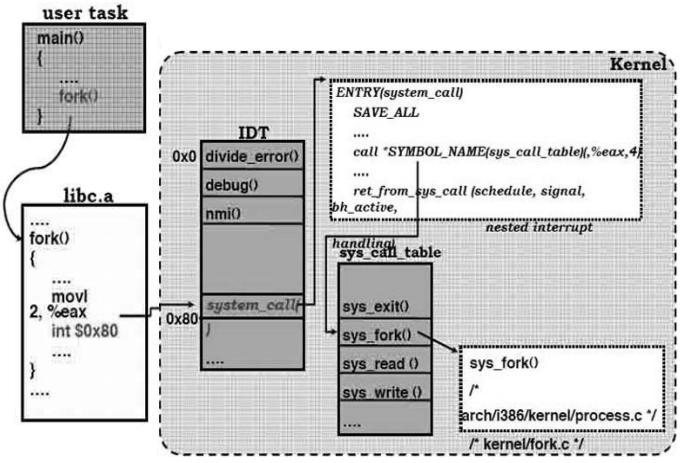

Figura 1.1 Il flusso di esecuzione quando si tenta di eseguire una syscall

Questa ci mostra quello che succede nel caso di una chiamata alla sys_fork:

dal nostro programma in user mode noi chiamiamo la funzione di libreria fork(), a questo punto il corrispondente codice all'interno delle librerie di sistema verrà mandato in esecuzione occupandosi di impostare opportunamente i registri (in questo caso inserisce il valore numerico 2 all'interno del registro %eax, ovvero il numero identificativo della sys_fork) e generando il software interrupt 0x80. A questo punto il controllo passa al kernel che una volta rilevato ed identificato l'interrupt (anche se nello specifico è una exception[3]) si occupa di mandare in esecuzione la funzione di gestione appropriata, il cui indirizzo è contenuto all'interno del corrispondente elemento nell'IDT. Come ci mostra la figura, l'elemento che ci interessa è quello che occupa la posizione 0x80 nell'IDT, ovvero quello contenente l'indirizzo della funzione system_call, che viene mandata in esecuzione. Questa funzione si occupa principalmente di svolgere due compiti: salvare il contenuto dei registri tramite una sequenza PUSH e richiamare la syscall richiesta, come possiamo vedere nel riquadro denominato "nested interrupt".

A SYMBOL_NAME(sys_call_table) sarà sostituito l'indirizzo in memoria della sys_call_table, perciò la riga call *SYMBOL_NAME(sys_call_table)(,%eax,4)

avrà questo significato: "chiama la funzione il cui indirizzo si trova nella locazione di memoria sys_call_table+%eax*4". A questo punto la syscall vera e propria sarà mandata in esecuzione libera di svolgere il suo compito, nel nostro caso libera di duplicare il processo corrente.

```
c010308c <system_call>:
c010308c:       50                      push   %eax
c010308d:       fc                      cld
c010308e:       06                      push   %es
c010308f:       1e                      push   %ds
c0103090:       50                      push   %eax
c0103091:       55                      push   %ebp
c0103092:       57                      push   %edi
c0103093:       56                      push   %esi
c0103094:       52                      push   %edx
c0103095:       51                      push   %ecx
c0103096:       53                      push   %ebx
c0103097:       ba 7b 00 00 00          mov    $0x7b,%edx
c010309c:       8e da                   movl   %edx,%ds
c010309e:       8e c2                   movl   %edx,%es
c01030a0:       bd 00 e0 ff ff          mov    $0xffffe000,%ebp
c01030a5:       21 e5                   and    %esp,%ebp
c01030a7:       66 f7 45 08 c1 01       testw  $0x1c1,0x8(%ebp)
c01030ad:       0f 85 c1 00 00 00       jne    c0103174 <syscall_trace_entry>
c01030b3:       3d 26 01 00 00          cmp    $0x126,%eax
c01030b8:       0f 83 2a 01 00 00       jae    c01031e8 <syscall_badsys>

c01030be <syscall_call>:
c01030be:       ff 14 85 c0 46 41 c0    call   *0xc04146c0(,%eax,4)
c01030c5:       89 44 24 18             mov    %eax,0x18(%esp)

c01030c9 <syscall_exit>:
c01030c9:       fa                      cli
c01030ca:       8b 4d 08                mov    0x8(%ebp),%ecx
```

Figura 1.2 Listato in assembly che ci mostra il codice della funzione system_call

Ora che abbiamo rivisto questo meccanismo, possiamo abbandonarci a qualche considerazione di carattere pratico: appare ovvio che riuscire a controllare il comportamento delle syscall corrisponde a poter manipolare a proprio piacimento il comportamento degli applicativi funzionanti in user mode qual ora richiedano l'intervento del kernel per l'erogazione di un qualsiasi servizio o risorsa. Su questa base nacquero i primi malware per il kernel di linux: erano LKMs (Loadable Kernel Modules) che agivano in modo tanto semplice quanto potente andando a modificare i puntatori contenuti all'interno della SCT, facendo cosi in modo che le syscall che venivano effettivamente mandate in esecuzione non fossero piu quelle originali, ma dei wrapper installati dal malware stesso. Tramite il wrapper era possibile richiamare la syscall originaria ed in caso di bisogno modificarne l'output, cosi da eliminare eventuali dati scomodi per l'attaccante.

Questo approccio permette di controllare il sistema in maniera estremamente semplice e soprattutto molto portabile tra le varie versioni di kernel dato che lavorando unicamente con le funzioni di interfaccia e sul loro filtraggio non si corre il rischio di incappare in problemi derivanti da un cambio anche radicale della loro implementazione. Il punto di forza di questa tecnica è appunto questo, portabilità e facilità di implementazione, tuttavia si deve pagare lo scotto della facilità di rilevamento dato che un controllo ai valori contenuti nella SCT rivelerebbe immediatamente la presenza del malware.

Una generazione piu recente di malware[4] ha fornito una nuova implementazione a questi concetti per ottenere un occultamento maggiore in caso di tentativi di rilevamento oltre che un ulteriore aumento alla portabilità: anziché utilizzare LKMs il codice maligno viene iniettato direttamente in /dev/kmem od equivalenti, ovvero un char device che fornisce un'immagine della memoria virtuale del kernel[2]

Questo fa si che non sia piu necessario che il kernel vittima abbia il supporto per i moduli abilitato per poter essere infettato, inoltre uno stesso malware binario può essere installato su piu kernel, senza bisogno di ricompilazione, cosa che poteva essere necessaria per i moduli.

L'aumento di occultamento viene ottenuto mediante una lieve modifica dell'approccio della generazione precedente: anziché modificare il contenuto della SCT viene creata in memoria una SCT nuova contenente puntatori alle funzioni wrapper del malware, poi l'indirizzo di questa nuova tabella viene sostituito all'interno della funzione system_call a quello della SCT originale. Basandoci sull'esempio di gestore riportato nella figura 1.2 il malware sovrascriverebbe l'indirizzo presente nella riga 0xc01030be, facendo cosi in modo che la call avvenga si con uno spiazzamento di %eax*4, ma partendo da una base diversa, quella della SCT del malware.

Chiaramente con questa tecnica tutti i controlli sulla SCT non darebbero alcun esito attendibile in merito alla rilevazione, in quanto non siamo piu in presenza del dirottamento di una syscall, ma della SCT stessa.

Per rilevare malware di questa tipologia[4] sono presenti svariati strumenti[5], ma molti hanno una debolezza intrinseca derivante della loro struttura: sono anch'essi applicativi funzionanti in user mode. Come ho gia detto prima, non possiamo fidarci delle syscall in un kernel infettato, per cui questi tool vengono ingannati molto facilmente da una qualsiasi modifica 'anomala' o ben congegnata, ossia mirata a far fallire questi strumenti: se ad esempio un applicativo in user mode cercasse di leggere dalla memoria kernel, via kmem

ad esempio, il contenuto della SCT per verificarne l'integrità, dovrebbe necessariamente farlo tramite syscalls, esponendosi cosi ad una interferenza da parte di un malware che potrebbe tranquillamente fargli leggere informazioni non veritiere.

Altri tool, anche se operanti a kernel space e pertanto non soggetti ad un'eventuale modifica delle syscall, hanno un altro problema in comune con quelli ad user mode, ovvero basano la rilevazione su dei pattern di riconoscimento o di controllo: in questo caso una modifica apportata a zone non controllate (ad esempio, modificare il gestore dell'interrupt 0x80 al posto della SCT) pregiudicherebbe il rilevamento. Bisogna tener presente che un malware potrebbe modificare qualsiasi zona del segmento testo (e non solo) del kernel, pertanto dei controlli solo sulle zone piu soggette agli attacchi non sono sufficienti.

Per evitare anche questo problema esiste una soluzione molto semplice, effettuare un hash del segmento testo del kernel e ripetere l'operazione in maniera asincrona da parte di una funzione operante in kernel mode, in modo che anche in presenza di modifiche non note o non riconducibili ad una determinata tipologia il rilevamento dia esito positivo, inoltre non si fornirebbe nessun appiglio ai malware per interferire coi controlli.

Nella prossima sezione inizieremo a vedere le basi per creare la struttura di un malware che ovvi anche ad una soluzione simile, un malware estremamente portabile che prenda il controllo delle syscall senza modificare il segmento testo del kernel e senza ripiegare sul poco portabile virtual file system[2].

III. L'INFRASTRUTTURA DI DEBUG

IA-32 fornisce meccanismi per la correzione del codice che sono un valido aiuto per il debug di:

Applicativi
Software di sistema
Sistemi operativi multiprogrammati

Si accede al supporto per il debugging tramite l'utilizzo di 8 *Debug Registers* (da dbr0 a dbr7) , dei registri specifici modello (*MSRs*) e dell'istruzione di breakpoint (int 3).

Esaminiamo brevemente la funzione ed il funzionamento dei debug registers:

i registri da dbr0 a dbr3 contengono l'indirizzo lineare di un breakpoint. Una debug exception viene generata quando avviene in tentativo di accesso all'indirizzo del breakpoint.

Il registro dbr6 riporta le condizioni che erano presenti quando una debug o breakpoint exception è stata generata, mentre dbr7 permette di specificare le forme di accesso che causeranno la generazione della debug exception al raggiungimento del breakpoint.

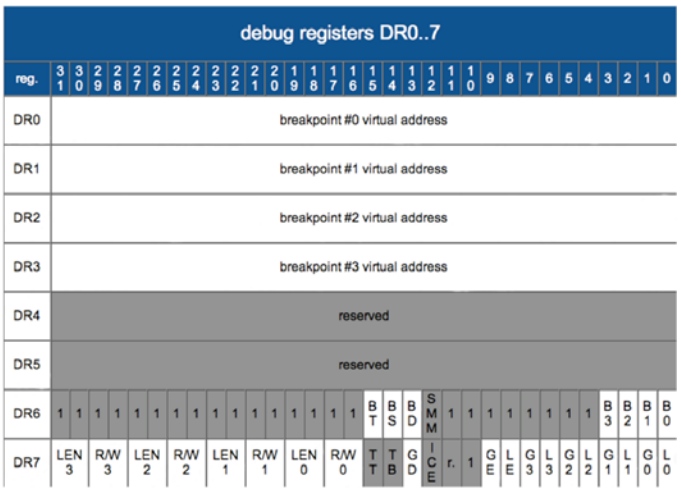

Figura 2.1 Schema riassuntivo degli 8 debug registers

Piu in dettaglio, le modalità di accesso che possono essere specificate tramite l'opportuna impostazione del registro dbr7 possono indicare:

Break alla sola esecuzione
Break alla sola scrittura
Break alla scrittura/lettura di I/O
Break alla lettura/scrittura di dati

Questo vuol dire che possiamo far generare una debug exception quando la CPU tenta di eseguire del codice posto in una qualsiasi locazione di memoria, anche a kernel space.

All'interno di linux la funzione *principale* atta alla gestione delle debug exception, come possiamo vedere dalla trap_init in poi all'interno dei file traps.c ed entry.S, è la

*void do_debug(struct *pt_regs, int errorcode)*

perciò possiamo dirottare qualsiasi flusso di esecuzione del kernel verso la do_debug senza modificare alcunché, unicamente impostando in maniera opportuna due registri.

```
struct pt_regs {
        long ebx;
        long ecx;
        long edx;
        long esi;
        long edi;
        long ebp;
        long eax;
        int  xds;
        int  xes;
        long orig_eax;
        long eip;
        int  xcs;
        long eflags;
        long esp;
        int  xss;
};
```

Figura 2.2 La struttura pt_regs

Analizzando la struttura pt_regs, riportata nella figura 2.2, abbiamo una panoramica dei registri di cui possiamo

conoscere il valore relativo al momento della generazione della debug exception e, cosa ancora piu interessante, volendo siamo liberi di modificare questi valori. Il nostro obiettivo è riuscire a modificare il contenuto del campo eip, corrispondente all'indirizzo di ritorno della procedura debug.

```
KPROBE_ENTRY(debug)
        cmpl $sysenter_entry,(%esp)
        jne debug_stack_correct
        FIX_STACK(12, debug_stack_correct, debug_esp_fix_insn)
debug_stack_correct:
        pushl $-1                       # mark this as an int
        SAVE_ALL
        xorl %edx,%edx                  # error code 0
        movl %esp,%eax                  # pt_regs pointer
        call do_debug
        jmp ret_from_exception
```

Figura 2.3 Listato del gestore della debug_exception

Prima ho detto che la do_debug era la funzione *principale* per la gestione delle debug_exceptions, ma la funzione il cui indirizzo è presente nell'elemento corrispondente alla debug_exception all'interno dell'IDT è la debug (Figura 2.3). Questa si occupa di fare tre cose:

- Assicurarsi che lo stack sia impostato in maniera corretta, e nel caso attraverso la macro FIX_STACK sistemarlo
- Inserire il valore dei registri da %es (xes nella nostra figura) ad %ebx nello stack tramite la macro SAVE_ALL ed impostare %edx ed %eax per un corretto passaggio dei parametri alla do_debug
- Chiamare la do debug

A questo punto, dal puntatore alla struttura pt_regs che viene passato come parametro alla do_debug abbiamo visto che possiamo accedere agli elementi da ebx ad xes (Ragionando dall'alto verso il basso sulla figura 2.2, ma dal basso verso l'alto nello stack). Torniamo ad osservare la figura 2.3 ora, prima della SAVE_ALL abbiamo un ulteriore PUSH, il cui valore secondo il ragionamento precedente verrà a trovarsi nel campo orig_eax, ma poi nessun altro valore viene salvato. Dunque il campo successivo della struttura punterà al valore precedentemente inserito nello stack, ovvero l'indirizzo di ritorno della funzione debug che era stato inserito in maniera automatica dalla CPU al momento della chiamata del gestore. Riassumendo, tramite il campo eip della struttura pt_regs che viene passata come argomento alla do_debug siamo in grado di sovrascrivere l'indirizzo di ritorno della funzione debug, dirottando cosi l'esecuzione del kernel in un qualsiasi punto a nostra discrezione.

IV. ABUSO DI LINUX

Rientrando nell'ottica di dover creare un malware e supponendo di avere il controllo della do_debug, possiamo quindi creare una procedura maligna a cui far saltare l'esecuzione dopo una debug exception ed una volta terminata, potremmo far risaltare l'esecuzione al punto giusto per far riprendere il naturale flusso di esecuzione.

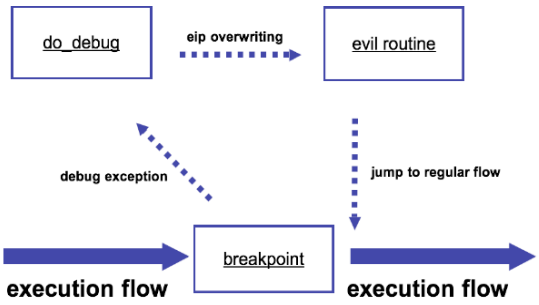

Figura 3.1 Schema di funzionamento del malware

In base a quanto esposto nella prima sezione, con questo approccio potremmo creare una nuova tipologia di malware che si comporti, ad esempio, come la prima versione di SucKIT[4], ovvero modificando l'indirizzo della SCT all'interno della system_call, solo che al posto di modificare ci basterebbe inserire l'indirizzo dell'istruzione CALL SYMBOL_NAME(...) in un nostro motore che utilizzi l'infrastruttura di debug come esposto nella sezione precedente, cosi da prendere il controllo dell'esecuzione prima della chiamata ad una qualsiasi syscall tramite una routine maligna attivata dalla debug exception che si occupi di chiamare i nostri wrapper (Figura 3.1)

Dobbiamo dunque prendere il controllo della do_debug o quantomeno operare all'interno di essa per avere accesso alla struttura pt_regs che le viene passata come parametro e per farlo abbiamo due strade:

Dirottarla con la tecnica del salto[2]
Abusare della flessibilità di linux

Dirottarla implicherebbe modificare parte del segmento testo e quindi renderci vulnerabili in caso di un hash, quindi dobbiamo giocoforza trovare un'altra strada ed è qui che linux ci viene incontro. Proviamo a guardare il listato della do_debug (Figura 3.2)

```
fastcall void __kprobes do_debug(struct pt_regs * regs, long error_code)
{
        unsigned int condition;
        struct task_struct *tsk = current;

        get_debugreg(condition, 6);

        if (notify_die(DIE_DEBUG, "debug", regs, condition, error_code,
                        SIGTRAP) == NOTIFY_STOP)
                return;
```

Figura 3.2 Parte del listato della do_debug

ed a seguire il parametro regs. Come possiamo vedere viene passato come argomento alla notify_die (Figura 3.3)

```
static inline int notify_die(enum die_val val, const char *str, struct pt_regs *regs, long err,
                             int trap, int sig)
{
    struct die_args args = {
                .regs = regs,
                .str = str,
                .err = err,
                .trapnr = trap,
                .signr = sig
        };

    return notifier_call_chain(&i386die_chain, val, &args);
}
```

Figura 3.3 Listato della notify_die

che lo inserisce all'interno di una struttura args di tipo die_args prima di passarlo alla notifier_call_chain (Figura 3.5) assieme ad un puntatore alla struttura i386die_chain di tipo notifier_block (Figura 3.4)

```
struct notifier_block
{
        Int (*notifier_call)(struct notifier_block *self, unsigned long, void *);
        struct notifier_block *next;
        int priority;
};
```

Figura 3.4 Struttura notifier_block

```
int __kprobes notifier_call_chain(struct notifier_block **n, unsigned long val,
                                  void *v)
{
        int ret=NOTIFY_DONE;
        struct notifier_block *nb = *n;
        while(nb)
        {
                ret=nb->notifier_call(nb,val,v);
                if(ret&NOTIFY_STOP_MASK)
                {
                        return ret;
                }
                nb=nb->next;
        }
        return ret;
}
```

Figura 3.5 Listato della notifier_call_chain

All'interno della notifier_call_chain (Figura 3.5) viene scorsa la lista di cui i386die_chain è il primo nodo e richiamata la funzione memorizzata all'interno di ogni elemento passandogli come argomento anche la struttura in cui era stato inserito il puntatore al parametro originario della do_debug. Se riuscissimo ad inserire un nostro nodo all'interno della lista verremmo comodamente chiamati dal kernel ad ogni debug exception permettendoci cosi di dar inizio al funzionamento di tutto il meccanismo descritto prima.

```
int register_die_notifier(struct notifier_block *nb)
{
        int err = 0;
        unsigned long flags;
        spin_lock_irqsave(&die_notifier_lock, flags);
        err = notifier_chain_register(&i386die_chain, nb);
        spin_unlock_irqrestore(&die_notifier_lock, flags);
        return err;
}
EXPORT_SYMBOL(register_die_notifier);
```

Figura 3.6 Listato della register_die_notifier

Come mostra la figura 3.6 esiste una funzione esportata che si occupa di registrare una qualsiasi struttura di tipo notifier_block all'interno della lista, fornendoci cosi l'ultimo tassello per il nostro puzzle.

V. CONCLUSIONI

La tecnica esposta è portabile su tutti i sistemi dotati di CPU intel-compatibile avente le estensioni per il debug (DE flag) ed un kernel appartenente alla serie 2.6. Per quelli della serie 2.4 è possibile dirottare la do_debug modificando il segmento testo, spostando cosi il punto in cui si va a modificare e cosi diminuendo le possibilità di un rilevamento, anche se sarebbe pur sempre possibile monitorare un accesso in lettura al suo indirizzo tramite un altro registro di debug e cosi tentare di ingannare uno scanner. In un contesto reale inoltre, su un kernel 2.4, sarebbe meglio dirottare per intero gestore dell'interrupt 0x80 dato che la struttura del sistema di tracing delle syscall ci obbligherebbe a monitorare un altro indirizzo in cui è presente una CALL alla sys call table. Un ulteriore sviluppo dell'implementazione proposta[6] per entrambe le famiglie di kernel potrebbe essere il supporto per le vsyscalls[3]

VI. RINGRAZIAMENTI

Voglio ringraziare tutte le persone che mi hanno dato supporto durante lo studio e l'implementazione di questa metodologia di attacco, in particolare (alfabeticamente) Attilio Rao, Enrico Perla e Massimiliano Oldani per le nostre conversazioni ricche di spunti e consigli.

VII. RIFERIMENTI

[1] Joanna Rutkowska – http://www.invisiblethings.org/papers/chameleon_concepts.pdf - 2003
Amit Vasudevan and Ramesh Yerraballi – http://www.acsac.org/2005/papers/72.pdf - 2005

[2] Pierre Falda - http://darkangel.antifork.org/publications/lkepd.html - 2004
PRAGMATIC - http://www.thc.org/papers/LKM_HACKING.html - 1999

[3] Intel Arch Manuals
http://www.intel.com/design/pentiumii/manuals/243190.htm
http://www.intel.com/design/pentiumii/manuals/243191.htm
http://www.intel.com/design/pentiumii/manuals/243192.htm

[4] sd - http://www.phrack.org/archives/58/p58-0x07 - 2001

[5] Vari strumenti di rilevamento:
http://www.chkrootkit.org
http://www.rootkit.nl/projects/rootkit_hunter.html
http://osiris.shmoo.com
http://la-samhna.de/samhain

[6] Pierre Falda – Mood-NT - http://darkangel.antifork.org/codes.htm - 2006