UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
CORSO DI LAUREA IN INFORMATICA

STUDIO E SVILUPPO DI UN FRAMEWORK PER L'ANALISI DINAMICA DI CODICE MALIGNO IN AMBIENTE VIRTUALIZZATO

Relatore: Prof. Danilo BRUSCHI
Correlatore: Dott. Giampaolo Fresi Roglia
Correlatore: Dott. Roberto Paleari

Tesi di Laurea di
Pierre FALDA
matr. 620496

Anno Accademico 2006/2007

Ai miei genitori.

Indice

Elenco delle figure

List of Algorithms

Introduzione

0.1 Obiettivi e motivazioni

In questa tesi viene sinteticamente e parzialmente illustrato il funzionamento dell'emulatore di macchina QEMU[1] e la struttura di alcune modifiche apportategli per ottenere un framework che permettesse in maniera rapida e semplice di poter monitorare e manipolare un qualsiasi flusso esecutivo all'interno di una macchina virtuale[2]. Questo framework è stato realizzato per velocizzare e facilitare il compito di un analista durante l'analisi dinamica di codice maligno, come ad esempio un trojan od un worm. Analisi dinamica significa che il codice maligno, che da ora chiameremo *malware*, viene analizzato durante la sua esecuzione. Per questo motivo è necessario sia un ambiente completamente isolato in cui mandarlo in esecuzione, per evitare la compromissione della macchina di lavoro, sia la totale trasparenza dell'ambiente virtuale e degli strumenti di analisi, per evitare che il malware riconosca di essere sotto osservazione e cambi così il suo comportamento. Il framework è un insieme di patch al codice di QEMU che introducono varie infrastrutture per poter facilmente monitorare

[1] QEMU è un virtualizzatore ed emulatore di macchina che utilizza traduzione e compilazione dinamica. È in grado di emulare svariate CPU (x86, PowerPC, ARM e Sparc) su molteplici tipologie di host (x86, PowerPC, ARM, Sparc, Alpha, Mips). QEMU supporta un'emulazione completa del sistema nel quale un sistema operativo non modificato è fatto funzionare all'interno di una macchina virtuale.[4]

[2] Una macchina virtuale è un contenitore software isolato in cui possono funzionare un sistema operativo e le relative applicazioni come se fossero su un calcolatore fisico. Una macchina virtuale si comporta esattamente come un calcolatore reale e contiene le proprie risorse virtuali (fornite tramite software) come CPU, RAM, Hard Disk, NIC. [27]

il malware, manipolarne il flusso esecutivo o quello del sistema operativo, estendere e personalizzare alcune caratteristiche di QEMU e del framework stesso. È stato scelto di lavorare su QEMU perchè è un progetto opensource costantemente aggiornato, non richiede modifiche al software che viene mandato in esecuzione, non richiede alcun supporto hardware particolare, supporta molte architetture differenti ed è disponibile per sistema operativo Linux, Windows, FreeBSD, MacOS X.

0.2 Organizzazione dell'elaborato

Nel Capitolo 1 vedremo i concetti fondamentali necessari per poter comprendere il lavoro svolto: una parte del funzionamento e della struttura di QEMU, accenni al funzionamento della paginazione su sistemi x86 ed un'introduzione alle varie tipologie di emulatori di macchine virtuali. Nel Capitolo 2 inizieremo a parlare del framework, analizzandone prima il design con le relative motivazioni, in seguito le componenti di personalizzazione e di monitoraggio. Nel Capitolo 3 è presente la spiegazione dell'infrastruttura di iniezione di codice all'interno della macchina virtuale, mentre nel Capitolo 4 verrà illustrata la struttura ed il funzionamento di un generatore automatico di codice per rendere estremamente semplice l'utilizzo delle funzionalità di iniezione. Infine, nel Capitolo 5, la conclusione dell'elaborato dove verranno riprese brevemente le caratteristiche del lavoro svolto. Tutto quanto esposto, a meno che venga specificato diversamente, va inteso come valido su architettura IA-32[14].

Capitolo 1

Concetti Preliminari

In questo capitolo faremo una panoramica delle nozioni necessarie per poter comprendere il lavoro svolto.

1.1 Qemu

1.1.1 Terminologia

Per semplicità lessicale ed espositiva, da questo punto in poi col termine *host* intenderemo il calcolatore su cui è in esecuzione QEMU, col termine *guest* la macchina virtuale creata da QEMU ed in particolare il sistema operativo che è in funzione al suo interno. Infine, con l'espressione *CPU target* intenderemo la CPU che vogliamo emulare.

1.1.2 Traduttore dinamico

Il traduttore dinamico di cui si serve QEMU effettua una conversione, al momento di esecuzione, delle istruzioni della CPU target in istruzioni appartenenti alla ISA [1] dello host. Il codice binario risultante è conservato un una cache di traduzione cosicchè possa venire riutilizzato in caso di bisogno. Generalmente i traduttori dinamici sono

[1] Instruction Set Architecture

difficili da adattare ad ogni diversa tipologia di host perché andrebbe riscritto l'intero generatore di codice, tuttavia quello di QEMU risulta essere molto più semplice dato che anzichè convertire ogni singola istruzione si limita a concatenare blocchi di codice macchina precedentemente generati tramite l'ausilio di GCC.

Funzionamento

Il primo passo è dividere le istruzioni appartenenti alla ISA della CPU target in un numero minore di istruzioni più semplici chiamate *micro-operazioni*. Ciascuna micro-operazione è realizzata con piccoli pezzi di codice C che successivamente vengono compilati con GCC per ottenerne il corrispondente codice oggetto. Le micro-operazioni sono scelte in modo tale che il loro numero sia molto minore di tutte le combinazioni di istruzioni ed operandi della CPU target. Uno strumento chiamato *dyngen* viene richiamato al momento della compilazione sul codice oggetto generato nel passo precedente e come output produce il sorgente del *generatore dinamico di codice*. Questo generatore dinamico viene invocato durante l'esecuzione di QEMU per generare una funzione completa lato host concatenando svariate micro-operazioni. Ad esempio, se avessimo la seguente istruzione nel guest:

```
movl $0xbadc0ded,%eax
```

Verrebbe tradotta nel seguente modo:

```
movl $0xbadc0ded, %T0
movl %T0, %eax
```

Dove T0 è un registro temporaneo mappato da QEMU in un registro dello host tramite l'estensione per le variabili registro statiche di GCC. A questo punto, il generatore dinamico di codice, dati i seguenti listati delle corrispettive micro-operazioni

```
080e3ed0 <op_movl_T0_imu>:
 80e3ed0: bb 34 3c b3 09          mov  $0x9b33c34,%ebx
 80e3ed5: c3                      ret
 80e3ed6: 8d 76 00                lea    0x0(%esi),%esi
```

```
 80e3ed9: 8d bc 27 00 00 00 00    lea    0x0(%edi),%edi

080e3040 <op_movl_EAX_T0>:
 80e3040: 89 5d 00              mov  %ebx,0x0(%ebp)
 80e3043: c3                    ret
 80e3044: 8d b6 00 00 00 00     lea    0x0(%esi),%esi
 80e304a: 8d bf 00 00 00 00     lea    0x0(%edi),%edi
```

creerebbe il seguente codice host

```
80e3ed0: bb 34 3c b3 09         mov  $0xbadc0ded,%ebx
80e3040: 89 5d 00               mov  %ebx,0x0(%ebp)
```

che potrebbe essere mandato direttamente in esecuzione. Su x86 il registro T0 mappato su ebx, mentre lo stato della CPU target su ebp, di cui il eax virtuale è il primo campo.

1.1.3 Translation Block

Quando QEMU incontra per la prima volta una porzione di codice guest, lo traduce fino alla prima istruzione di *salto* od alla prima istruzione che modifica lo stato della CPU in una maniera che non è deducibile staticamente al momento della traduzione. Chiameremo questi blocchi *Translation Block*. Una cache di 16Mb mantiene in memoria i Translation Block usati piu di recente e quando è piena, per semplicità, viene svuotata. Dopo che il codice di un Translation Block è stato eseguito, QEMU utilizza il *program counter simulato* ed altre informazioni contenute nello stato della CPU target per cercare il prossimo Translation Block da eseguire sfruttando una hash table. Se questo Translation Block non è ancora stato creato, allora viene lanciata una nuova traduzione, altrimenti viene effettuato un salto a quest'ultimo. Per ottimizzare il caso più comune, ovvero dove il prossimo valore del program counter è conosciuto (ad esempio un salto condizionale), QEMU può modificare un Translation Block aggiungendo un'istruzione di salto per poter saltare direttamente al codice del prossimo Translation Block.

1.2 Indirizzi di memoria

I programmatori normalmente si riferiscono ad un indirizzo di memoria come un modo per accedere al contenuto di una cella di memoria. Quando si sta lavorando con processori della famiglia 80 x 86 dobbiamo però distinguere tre tipologie di indirizzi:

- Indirizzo logico: incluso nelle istruzioni in codice macchina per specificare l'indirizzo di un operando o di un'istruzione. Questo tipo di indirizzo incorpora la ben conosciuta architettura a segmenti propria di 80 x 86. Ciascun indirizzo logico consiste in un segmento ed in uno spiazzamento che indica la distanza dall'inizio del segmento all'indirizzo in questione.

- Indirizzo lineare[2]: un singolo intero a 32 bit[3] che può essere utilizzato per indirizzare fino a 4GB, ovvero 4,294,967,296 celle di memoria.

- Indirizzo fisico: utilizzato per indirizzare le celle di memoria all'interno dei chip. Corrisponde ai segnali elettrici inviati attraverso i pin del microprocessore al memory bus

La Memory Management Unit (MMU) trasforma un indirizzo logico in un indirizzo lineare attraverso un circuito hardware chiamato *segmentation unit*; successivamente, un secondo circuito hardware chiamato *paging unit* trasforma un indirizzo lineare in un indirizzo fisico[6].

Figura 1.1: Traduzione di un indirizzo logico

[2]Anche conosciuto come indirizzo virtuale
[3]nel caso la CPU stia lavorando in modalità a 32 bit

1.2.1 Paging Unit

Windows, Linux ed i maggiori sistemi operativi, su sistemi a 32 bit della famiglia x86, in condizioni normali[4] possono accedere fino a 4GB di memoria fisica. Questo è dovuto al fatto che il bus degli indirizzi del processore ha 32 linee, ovvero bit, e può così accedere unicamente al range di indirizzi da 0x00000000 a 0xFFFFFFFF, ovvero 4GB. Questi sistemi operativi permettono che ogni flusso esecutivo abbia 4GB di address space logico. Non tutti questi 4GB saranno utilizzabili da un flusso esecutivo per via della divisione della memoria tra kernel ed user space[5][12]. Per fare questo viene utilizzata una caratteristica messa a disposizione dai processori della famiglia x86, dal 386 in poi, chiamata *paging*. Il paging, o paginazione che dir si voglia, permette al software di utilizzare indirizzi di memoria (indirizzi logici) differenti dagli indirizzi fisici. Vediamo ora un po' più in dettaglio come funziona la paginazione su x86.

1.2.2 Paginazione

Di *prassi* il processore divide lo spazio di indirizzamento fisico in pagine di 4KB, rendendo così necessario l'utilizzo di 1Mega (1024 x 1024) di pagine per mappare 4GB. Può essere pensato come una matrice bidimensionale in cui la prima dimensione è chiamata Page Directory, mentre l'altra Page Table. A ciascuna *Page Directory Entry* sono associate 1024 *Page Table Entry*, ognuna delle quali, se la pagina è valida, contiene il *Page Frame Number* della pagina fisica che stiamo cercando. Il PFN non è altro che un indice per la memoria fisica espresso in *page size*. Un indirizzo lineare perciò, durante la sua traduzione, viene così suddiviso: i 10 bit più significativi vengono utilizzati come indice all'interno dell'array di PDE, i successivi 10 bit fungono come indice per l'array di PTE individuato dalla PDE, mentre gli ultimi 12 bit vengono utilizzati come spiazzamento all'interno della pagina di 4KB[22]. Graficamente, la struttura è rappresentata dal seguente schema:

[4]Non considerando perciò cose come il *PAE* od il supporto per la memoria alta di Linux

[5]Se così non fosse sarebbe necessario un *TLB flush* ad ogni passaggio kernel/user mode, operazione *estremamente* onerosa

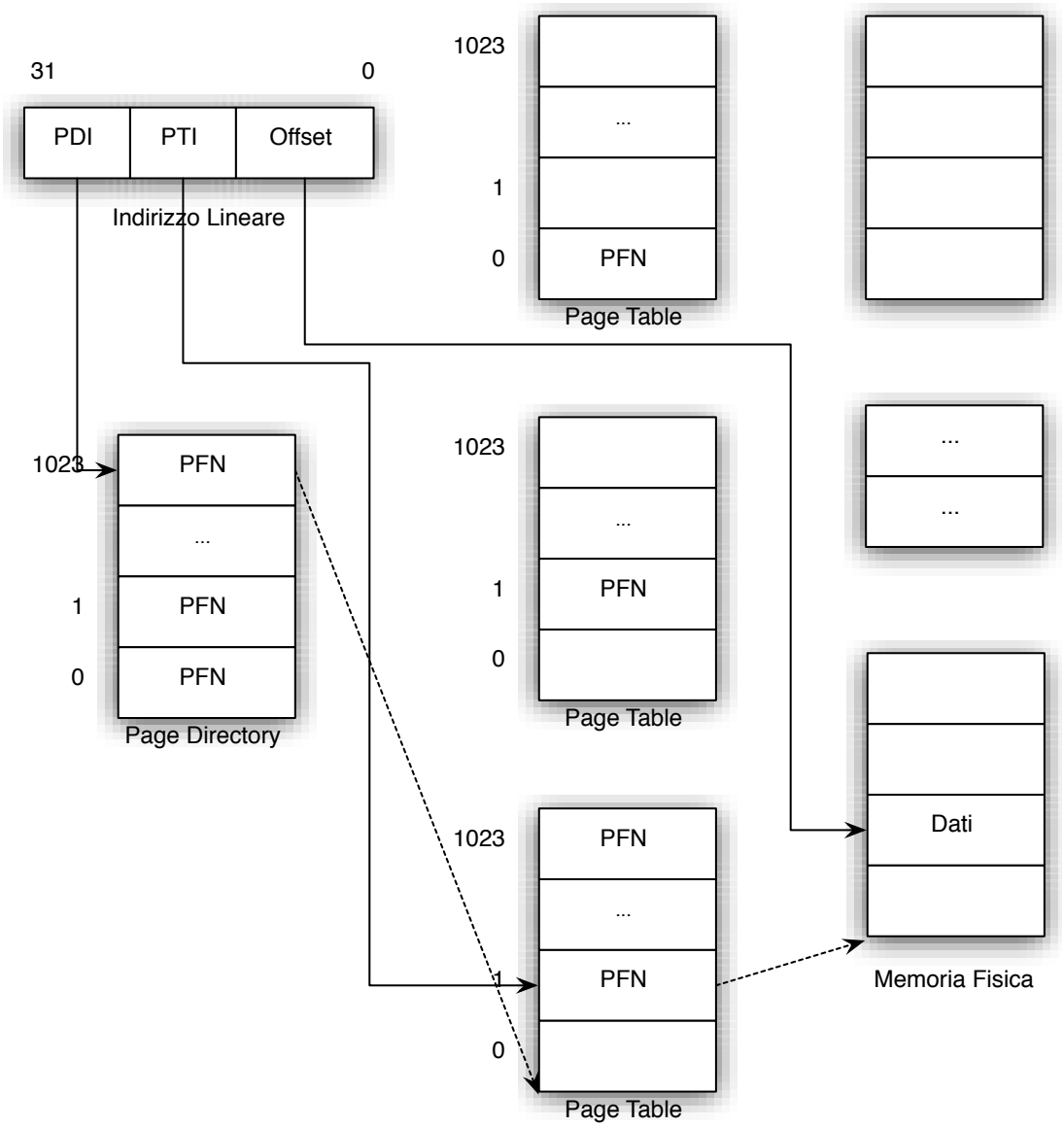

Figura 1.2: Struttura dell'indirizzo e della paginazione in caso di tabelle delle pagine a due livelli

Dato un indirizzo virtuale e la Page Directory di un flusso di esecuzione siamo in grado di localizzare in memoria fisica i dati a cui il flusso esecutivo voleva accedere. Ma dove si trova la Page Directory? Anch'essa è in memoria fisica ed il suo indirizzo (fisico ovviamente) è contenuto all'interno del *Control Register 3*[6]. Ogni flusso esecutivo ne possiede uno differente ed ad ogni *context switch* il valore contenuto nel CR3 viene aggiornato con quello relativo al flusso che sta per essere eseguito, perciò ognuno si vede assicurato uno spazio di indirizzamento di 4GB completamente indipendente e protetto da interferenze esterne.

1.3 Virtualizzazione

Esistono due tipologie di emulatori di virtual machine: *hardware-bound* (conosciuta anche come para-virtualizzazione) e *pure software* (tramite emulazione della CPU). La categoria hardware-bound può essere suddivisa in due sottocategorie: *hardware-assisted* e *reduced privilege guest* (anche chiamata ring 1 guest). Entrambe le forme di virtual machine hardware-bound si appoggiano sulla CPU reale sottostante per eseguire istruzioni non critiche nativamente e per questo motivo, quando comparate ad implementazioni pure software, risultano dare prestazioni migliori. Tuttavia, siccome eseguono istruzioni sulla CPU reale, devono effettuare qualche variazione all'ambiente di esecuzione per poter dividere le risorse hardware tra il sistema operativo guest e quello host. Alcuni di questi cambiamenti sono visibili all'interno del sistema operativo guest, se un'applicazione sa cosa controllare.

1.3.1 Emulatori di virtual machine hardware-bound

La differenza tra emulatori di virtual machine hardware-assisted e reduced privilege guest è la presenza di istruzioni nella CPU specifiche per la virtualizzazione. Gli emulatori hardware-assisted utilizzano istruzioni della CPU specifiche per mettere il sistema in modalità virtuale. Il guest funziona allo stesso livello di privilegi che avrebbe avuto se fosse stato veramente controllato dalla CPU in assenza di un emulatore di virtual machine. Le strutture dati importanti ed i registri hanno copie *shadow* che il

guest vede, ma queste copie non hanno effetto sullo host. Lo host controlla le strutture dati reali ed i registri, fornendo così una virtualizzazione quasi completamente trasparente. Lo host può direttamente istruire la CPU di notificarlo se avvengono certi eventi specifici, come un tentativo di accesso ad una particolare zona di memoria od un tentativo di richiedere le caratteristiche della CPU.

Emulatori di virtual machine reduced privilege guest

Gli emulatori della famiglia reduced privilege guest devono virtualizzare le strutture dati importanti e gli stessi registri, Il guest funziona ad un livello di privilegio più basso[6] di quello che avrebbe avuto se fosse stato realmente controllato dalla CPU. Non c'è alcun modo di impedire che la CPU notifichi allo host tutti gli eventi. Alcuni esempi di emulatori di virtual machine reduced privilege guest sono VMware[27], Xen[28] e Parallels[21].

Emulatori di virtual machine hardware-assisted

Xen 3.x e Parallels, possono funzionare anche come emulatori di virtual machine hardware-assisted. Un emulatore hardware-assisted imposta alcune strutture specifiche della CPU ed in seguito utilizza l'istruzione *VMLAUNCH*[17] su CPU Intel, oppure *VMRUN*[2] su CPU Amd per mettere il sistema operativo in uno stato virtualizzato. A questo punto, ci sono effettivamente due copie del sistema operativo esistenti, ma una (lo host) è sospesa, mentre l'altra (il guest) funziona liberamente nel nuovo stato. Ogni qualvolta avviene un evento interessante (come un'interrupt od un'eccezione) il sistema operativo host (l'emulatore di virtual machine) riprende il controllo, gestisce l'evento e si occupa di far riprendere l'esecuzione al sistema operativo guest. Ogni macchina che supporta l'esistenza di un *hypervisor* può farne partire uno in qualsiasi momento. Nè il sistema operativo nè l'utente, se ne accorgeranno. In teoria, una volta che il guest è attivo, l'emulatore di virtual machine non può essere rilevato dato che intercetta tutte le istruzioni sensibili.

[6] Ovvero più alto

1.3.2 Emulatori di virtual machine pure software

Gli emulatori di virtual machine pure software lavorano effettuando via software operazioni equivalenti ad ogni operazione della CPU da emulare. Il vantaggio principale rispetto agli emulatori hardware-bound è che la CPU software non devo corrispondere alla CPU reale sottostante. Questo fa si che un ambiente guest possa essere tranquillamente spostato tra più macchine aventi architetture differenti. Alcuni esempi di emulatori di virtual machine pure software sono QEMU e Bochs[5]. Un altro metodo di emulazione di virtual machine è spesso usata dai software anti-virus. Consiste nell'emulazione della CPU e di una parte di un sistema operativo, come ad esempio Windows o Linux. Due esempio sono Atlantis[10] e Sandbox[20]. Entrambe questi due prodotti permettono di far funzionare un file potenzialmente malizioso mentre si sta analizzando il suo comportamento in maniera completamente sicura.

Capitolo 2

Il framework

In questo capitolo verranno descritte la struttura e le funzionalità del framework soffermandoci sui dettagli implementativi più interessanti

2.1 Design

QEMU è un programma che è stato scritto avendo tra gli obiettivi primari elevate prestazioni. Partendo da quest'ottica, risulta perfettamente comprensibile la scelta dell'autore di scriverlo in linguaggio C ed Assembly, purtroppo, allo stesso tempo, il bisogno di alte prestazioni ha fatto sì che il programma assumesse una struttura non particolarmente flessibile ed espandibile. I requisiti del framework perciò erano che fosse potente, flessibile e non influisse eccessivamente sulle prestazioni. Possiamo dividere il lavoro svolto in tre aree, in base alle funzionalità che vanno ad offrire:

- area della personalizzazione
- area del monitoraggio
- area della manipolazione

2.2 Personalizzazione

Uno dei metodi più semplici ed efficaci per personalizzare un software è quello di fargli supportare plug-in esterni, pertanto opteremo per questa soluzione e nella fattispecie, i plug-in verranno forniti tramite shared library[19]. Al di la della semplicità da parte di un utente di inserire un plug-in unicamente copiando la relativa libreria in una directory specificata, è stato un modo valido di procedere perchè QEMU nonostante non abbia il supporto per farlo, ha una miriade di cose/comportamenti che potrebbero voler essere modificati ed in molti casi anche le API interne necessarie allo scopo. In sostanza, le funzioni per gestire molti comportamenti/svolgere certi task ci sono, ma è assente il modo di richiamarle a comando. Tramite una shared library si può avere accesso ai simboli[1] del programma che la carica, dando così al plug-in la possibilità di lavorare con le API e le strutture necessarie. È necessario puntualizzare che accesso ai simboli tramite shared library non significa accesso indiscriminato, le strutture e le funzioni interne che non dovrebbero essere utilizzate/modificate sono dichiarate statiche e perciò non accessibili. Con un plug-in perciò si può facilmente lavorare con API interne su strutture interne senza violare il (ridotto) incapsulamento dei dati già presente. Non essendoci nessuna funzionalità standard di QEMU che possa venire naturalmente estesa tramite plug-in senza una massiccia opera di reingegnerizzazione , è estremamente più versatile, flessibile e performante[2] permettere ad un plug-in di effettuare il proprio lavoro senza far rientrare a forza le sue funzionalità in una qualche classe di appartenenza, in modo tale da non limitarne potenzialità ed interazione a determinati ambiti. Per quelle caratteristiche invece di naturale espansione[3] o per cui un sistema di liste callback sarebbe stato consono[4], sono state create strutture ed API apposite per permettere di effettuare tutte le operazioni possibili in modo rapido, semplice e pulito.

[1] Un simbolo è un'etichetta che si riferisce ad un blocco costituente di un programma: variabili, funzioni...

[2] Riscrivere porzioni di codice in molti punti inserendo forzatamente sistemi di liste callback per permetterne la personalizzazione, avrebbe introdotto un gran numero di branch indiretti, rallentando così di fatto l'esecuzione[13]

[3] Come ad esempio l'inserimento di un nuovo comando

[4] Ne vedremo un esempio nella parte del monitoraggio

Un altro vantaggio legato all'utilizzo di shared libraries (ed anche il più immediato) è dato dal fatto che anche dal programma principale è possibile accedere ai simboli delle librerie caricate. Utilizziamo questa peculiarità in praticamente ogni programma senza pensarci, tuttavia nel nostro caso la questione è differente: normalmente le funzionalità fornite da una libreria vengono utilizzate dal programma chiamante per svolgere qualche compito, nel nostro caso invece sono le librerie che vanno a svolgere dei compiti senza che il programma chiamante ne sia al corrente. Nonostante la struttura esposta lasci ampie possibilità di personalizzazione, in casi particolari potrebbe essere necessario mandare in esecuzione una funzione fornita dal plug-in, in modo asincrono al suo caricamento/scaricamento, sostituendo quella di QEMU in un punto per cui non sono previste estensioni e per cui è previsto che venga effettuata una sola chiamata a funzione. Vediamo un esempio:

```
static int pippo(int uno,int due, char *tre) {
        int ret;
        if(due>=uno)
                ret = printf("%u >= %u\n",due,uno);
        else
                ret = printf("%u < %u\n",due,uno);
        printf("Questa funzione");
        printf(" finita\n");
        return ret;
}

int main(void) {
        pippo(1,2,"ciao");
}
```

Noi potremmo voler manipolare

- le stringhe che vengono stampate tramite la funzione printf

- le stringhe che vengono stampate tramite la funzione printf una chiamata si ed una no
- il comportamento della funzione printf solo quando sta eseguendo codice per il plug-in corrente

Senza alcun supporto per ottenere questo risultato saremmo obbligati a sostituire la libreria che fornisce la funzione printf con una scritta da noi, oppure seguire un approccio decisamente più brutale andando a modificare al volo il segmento testo. La prima opzione può non essere applicabile nel caso in cui si voglia manipolare una funzione che non è fornita da alcuna libreria, mentre la seconda non dovrebbe neanche essere presa in considerazione.

2.2.1 Customizable function

Per questo motivo, sono state introdotte in QEMU le *customizable function*. Una volta che in QEMU una funzione viene dichiarata come *customizable* tramite un'apposita macro, diventa possibile effettuare un *override* della stessa da qualsiasi plug-in. Un plug-in può effettuare l'override di qualsiasi numero di funzioni, disattivare e riattivare l'override di qualsiasi customizable function a piacere qualsiasi sia il plugin che ne ha fornita un'implementazione personalizzata ed in qualsiasi momento. Vediamo brevemente il loro funzionamento: quando viene dichiarata una customizable function

```
DECLARE_CUSTOMIZABLE_FUNCTION(int,pippo,int,int);
```

in fase di preprocessing la macro viene espansa nel seguente modo

```
#define DECLARE_CUSTOMIZABLE_FUNCTION(rettype,name, args...)
struct {
         rettype (*func_##name)(args);
         rettype (*orig_##name)(args);
         rettype (*backup_##name)(args);
} __attribute__((packed)) object_##name =
{ .func_##name=name,.orig_##name=name, .backup_##name=name };
```

per cui nel nostro caso il codice generato diventa

```
struct {
          int (*func_pippo)(int,int);
          int (*orig_pippo)(int,int);
          int (*backup_pippo)(int,int);
} __attribute__((packed)) object_pippo =
{ .func_pippo=pippo,.orig_pippo=pippo, .backup_pippo=pippo };
```

ovvero viene creata una struttura di nome object_pippo contenente tre puntatori alla funzione pippo. Attraverso tre semplici macro

```
#define CUSTCALL(name,args...)
object_##name.func_##name(args)
#define CUSTUNLOAD(name)
object_##name.func_##name=object_##name.orig_##name
#define CUSTLOAD(name)
object_##name.func_##name=object_##name.backup_##name
```

siamo in grado di richiamare la funzione principale associata all'oggetto appena creato oppure di manipolare questi puntatori per far assumere a quest'ultimo il comportamento che più ci piace. Nel caso un plug-in voglia effettuare un override di una customized function, non dovrà far altro che fornire il nome della funzione che vuole sostituire e quello della sostituta tramite le apposite API ed il sistema di gestione dei plug-in si occuper`a di modificare i campi func e backup del relativo oggetto. Con *CUSTCALL* richiamiamo una customized function, con *CUSTUNLOAD* eliminiamo l'override associato alla funzione ed infine con *CUSTLOAD* possiamo riattivare un override precedentemente disattivato tramite CUSTUNLOAD. Con questo sistema diventa possibile raggiungere un alto grado di flessibilità dell'applicativo e di granularità nella sua personalizzazione nonostante non sia stato realizzato secondo il paradigma ad oggetti nè pensato per esserlo, il tutto senza ricorrere ad un refactoring dell'intero codice e con un trascurabile impatto prestazionale dato che l'incremento di chiamate indirette introdotto con questo sistema è assolutamente limitato, controllato e controllabile.

Riassumendo, tramite plug-in forniti via shared library e customizable functions noi possiamo:

- Aggiungere funzionalità a QEMU
- Modificare funzionalità aggiunte o già esistenti
- Modificare il comportamento di alcune funzioni in qualsiasi modo si desideri
- Rimuovere funzionalità
- Sviluppare ed inserire nuove funzionalità sfruttando il codice presente in altri plug-in, favorendo così il riutilizzo del codice

2.3 Monitoraggio

QEMU integra un gdbserver, questo significa che è possibile effettuare debug del codice guest anche remotamente attraverso gdb. Più in dettaglio, è possibile fermare l'esecuzione del codice guest in qualsiasi punto ed è possibile leggere o scrivere in qualsiasi indirizzo o registro virtuale. Questa possibilità è senza ombra di dubbio molto interessante, tuttavia potrebbe non essere sufficiente nel momento in cui volessimo monitorare entità piu complesse di una dword[5]. Potremmo, ad esempio, voler monitorare ed analizzare i parametri che vengono passati ad una certa funzione o strutture eventualmente ritornate dalla stessa, arrivando persino a modificarli a nostro piacimento in certe occasioni. Questo genere di attività potrebbe essere svolto senza dubbio col solo ausilio degli strumenti fornitici da gdb, tuttavia sarebbe un lavoro estremamente lungo, tedioso e ripetitivo. Come gia anticipato prima, questo è un compito per cui una struttura a lista di callback sarebbe perfetta: ogni plug-in potrebbe registrare le proprie funzioni per l'analisi del codice guest in corrispondenza dell'inizio o della fine di un qualsiasi Translation Block od ad un certo valore del program counter

Nel primo capitolo abbiamo introdotto i Translation Block, i basic block in cui QEMU memorizza il codice host da eseguire relativo ad una porzione di codice guest.

[5]Double word = 4 bytes

Tutto il codice guest viene smembrato in Translation Block e successivamente mandato in esecuzione un frammento alla volta, non solo quello del o dei processi che intendiamo monitorare, perciò serve un modo per poter distinguere *a chi* appartiene una determinata porzione di codice che sta per essere eseguita. Affidarci a valori propri del sistema operativo ospitato nella virtual machine, come ad esempio il PID[6], oltre che ad essere potenzialmente inaffidabile[7] avrebbe fatto perdere generalità al sistema di analisi perchè il reperimento di questa informazione sarebbe dipeso dal sistema emulato, perciò la scelta è ricaduta sul valore del registro *CR3*.

2.3.1 Associazione CR3 - Flusso

Il ciclo principale di esecuzione del codice guest avviene nella funzione cpu_exec; questa funzione, a grandi linee, si occupa di:

- gestire eventuali interrupt pendenti
- trovare il prossimo Translation Block da eseguire
- eseguire il codice host del Translation Block prescelto

per cui mettendo un sistema di callback prima e dopo il punto di esecuzione del codice di un Translation Block , una funzione potrebbe essere richiamata prima e dopo aver eseguito il codice posto ad un certo valore del program counter, valore che possiamo verificare in qualsiasi momento dato che, come detto in precedenza, è memorizzato all'interno del Translation Block stesso. Il valore del CR3 è possibile ottenerlo in qualsiasi momento essendo memorizzato nello stato della CPU virtuale, il valore del program counter lo possiamo vedere dal Translation Block corrente, tutto quel che manca per poter riconoscere con certezza il/i CR3 del/dei flusso/i che vogliamo monitorare è il nome del flusso/i a cui corrispondono tutti i dati sopracitati. Con la struttura precedentemente esposta e tenendo presente che un *hook* è un punto di aggancio da cui le nostre funzioni possono essere richiamate, è possibile risolvere elegantemente il problema con la seguente tecnica nel caso stessimo analizzando un qualsiasi processo:

[6] Process IDentifier

[7] Se ad esempio stessimo monitorando un codice che va ad interferire con quelle strutture/funzionalità del sistema, avremmo seri problemi di interferenza

Algorithm 1 Monitor: TrovaCR3

```
Registra un hook relativo alla funzione responsabile della creazione dei processi del
sistema operativo
while Ho codice guest da eseguire do
  if in qualsiasi momento sto per eseguire la funzione di creazione processi del
  sistema operativo then
    manda in esecuzione la callback registrata
    attraverso la callback leggi nella memoria del guest e recupera l'indirizzo di
    ritorno della funzione monitorata
    registra dinamicamente un hook per l'indirizzo di ritorno
  end if
  if sto per eseguire il codice all'indirizzo di ritorno then
    tramite la callback associata leggi i dati del nuovo PCB appena creato[22]
    if il nome del processo appena creato corrisponde col nostro bersaglio oppure
    il CR3 del padre è nella lista dei CR3 monitorati then
      registra il relativo CR3 nella lista dei CR3 monitorati
    end if
  end if
end while
```

oppure, volendola esporre in forma grafica, eccone il flusso esecutivo:

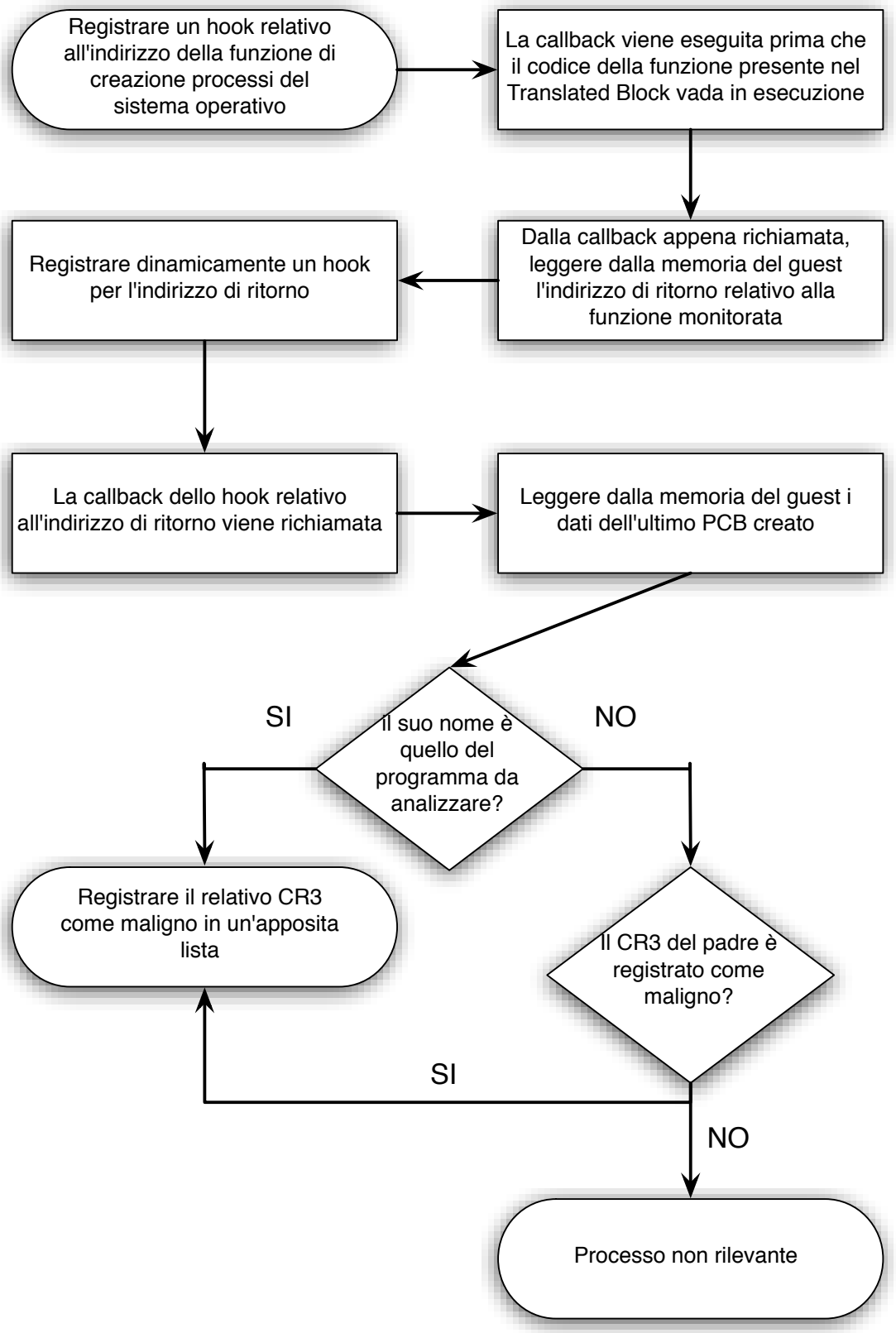

Figura 2.1: Ottenimento CR3 del processo bersaglio

il tutto in modo largamente indipendente dal sistema operativo emulato[8], assolutamente non intrusivo dato che non richiede alcuna modifica a nessun codice né componenti aggiuntivi interni e completamente invisibile dall'interno della virtual machine. Con quanto esposto finora, è possibile monitorare comodamente l'esecuzione di qualsiasi flusso, permettendo così scoprire quali funzioni utilizza semplicemente inserendo delle notifiche agli indirizzi virtuali corrispondenti, ad esempio, alle API di sistema. È inoltre possibile monitorare e manipolare dati, come gli argomenti delle funzioni, tuttavia per quest'ultima cosa sorge una piccola complicazione: quando si tenta di accedere ad un determinato indirizzo, la relativa pagina potrebbe non essere più in memoria ed essere stata soggetta a *swapping*[9] oppure non essere mai stata caricata, come nel caso della *lazy evaluation*[10] effettuata da Windows[22]. In questo caso, per reperire le informazioni presenti nella pagina, è sufficiente manipolare lo stato della CPU virtuale in modo tale da far credere al sistema guest che ci sia stato un *page fault*[11] per la pagina corrispondente all'indirizzo a cui vogliamo accedere, facendo cosi in modo che sia il *page fault handler* del sistema guest a farsi carico del suo reperimento dal disco fisso e del caricamento in memoria.

2.4 Manipolazione

Per effettuare un'analisi di alto profilo, potrebbero non essere sufficienti tutte le funzionalità illustrate fino ad ora. Potremmo avere la necessità di influenzare un flusso di esecuzione modificando la destinazione di un salto, l'esito di un test, di *inserire* delle chiamate a funzione (come ad esempio le API di sistema) oppure creare un flusso esecutivo completamente nuovo con del codice, anche complesso, fornito da noi. In sostanza, potremmo aver bisogno di:

[8] Non completamente perchè il metodo di localizzazione ed interpretazione del PCB appena creato varia in funzione del sistema operativo in esame

[9] Processo relativo alla memoria virtuale in cui una pagina di memoria non più ritenuta necessaria viene scaricata sul disco fisso

[10] Lazy evaluation significa aspettare di effettuare un task fino a che non è richiesto. Nel particolare, all'inizio della vita di un processo, le sue *page table* spesso non includono le shared library utilizzate da quel processo. Le page table sono aggiornate in seguito solamente quando la CPU referenzia la memoria all'interno delle shared library

[11] Eccezione indicante che una pagina richiesta non è presente in memoria

- *iniettare* a *qualsiasi* indirizzo un *qualsiasi* codice
- mandare in esecuzione il codice fornito, prima, dopo od addirittura al posto di un'altra porzione di codice

Vediamo qualche esempio dell'applicazione di quanto detto[3]:

- Creazione od apertura di file: la funzione delle API di Windows *CreateFile* e l'equivalente nativa *NtCreateFile* possono entrambe venire utilizzate per l'apertura o la creazione di file. Non c'è modo di differenziare se il comportamento richiesto è l'apertura o la creazione unicamente analizzando i relativi parametri. Per essere in grado di poterlo fare, dobbiamo inserire una chiamata ad una nostra funzione che controlli se il file esiste già oppure no. Questa stessa situazione la ritroviamo esaminando la Windows API *RegCreateKeyEx* dato che può essere utilizzata sia per aprire sia per creare chiavi di registro.
- File o Directory: In qualche situazione, non è possibile decidere se un nome di file si riferisce ad un file od ad una directory unicamente esaminando i parametri delle funzioni
- Comportamentale: modificare l'esito di un test o l'indirizzo a cui effettuare un salto per poter così forzare l'esplorazione di ogni possibile schema comportamentale
- Analitico: sostituendo una qualsiasi porzione di codice (anche appartenente al sistema operativo volendo) con una fornita da noi ci permette di poter effettuare un hooking assolutamente generale ed invisibile dall'interno dell'ambiente virtuale. Potremmo così modificare il comportamento di API di sistema o sostituirle completamente, sostituire funzioni o parti di funzione del codice analizzato.

Dal prossimo capitolo inizieremo ad analizzare tutte le problematiche che comporta quanto ci prefiggiamo di fare e le relative soluzioni.

Capitolo 3

L'iniettore

In questo capitolo vedremo tutte le problematiche relative alla progettazione ed alla realizzazione dell'iniettore di codice con relative soluzioni

3.1 Requisiti

I requisiti che ci imporremo per l'iniettore saranno i seguenti:

1. deve lavorare in modo assolutamente trasparente e non invasivo

2. deve essere completamente generale e perciò indipendente dal sistema operativo

3. l'analista deve poterlo utilizzare con estrema facilità

4. non deve porre alcun limite riguardo al codice da iniettare, né in fatto di dimensioni né riguardo la tipologia delle operazioni da svolgere, ovvero, dato un qualsiasi programma si dovrebbe poterlo poterlo iniettare senza problemi

Ora analizzeremo in dettaglio i vari componenti che sono stati necessari per raggiungere quanto ci stiamo prefiggendo.

3.2 Traduzione

In primo luogo, bisogna far notare che qualsiasi codice si voglia iniettare, esso deve venire eseguito nel contesto della CPU del guest. Visto che l'estrema facilità di utilizzo da parte dell'analista è parte dei requisiti, non possiamo richiedere alcuno sforzo a livello programmativo, come ad esempio produrre codice che vada a modificare lo stato della CPU virtuale: come già illustrato in precedenza, la CPU virtuale è memorizzata in una struttura presente nella memoria dello host e ciascuna porzione di codice guest viene eseguita nativamente dallo host previa traduzione, per cui il codice in questione avrebbe dovuto modificare i campi di questa struttura rappresentante la CPU virtuale per poter emulare l'esito delle istruzioni virtuali. Siamo perciò costretti a spostare il problema della generazione di codice dall'analista al framework, dobbiamo dunque poter *tradurre* in maniera automatica le istruzioni che vogliamo che il guest esegua, nel relativo codice eseguibile dallo host. Come spiegato nel primo capitolo, QEMU si avvale di un traduttore dinamico per convertire le istruzioni del sistema guest in codice eseguibile lato host, tuttavia non è possibile utilizzarlo direttamente: il traduttore si occupa di recuperare le istruzioni guest che devono essere eseguite e tradurle, ma quelle che dobbiamo fornire noi sono istruzioni presenti nella memoria dello host, non del guest. Rendere presenti le istruzioni da iniettare nella memoria del guest, magari attraverso un driver apposito, avrebbe violato i requisiti di trasparenza, invasività e generalità, in quanto questo eventuale driver avrebbe dovuto essere riscritto per ogni diverso sistema operativo. Dovremo dunque fare in modo che, a comando, il traduttore inizi a reperire informazioni da aree di memoria da noi indicate e non più dal guest.

Le principali funzioni di QEMU che si occupano della generazione del codice host/guest sono la *disas_insn* e la *dyngen_code*. La disas_insn si occupa di reperire il od i bytes delle istruzioni guest, decodificarli e creare un buffer contenente meta-istruzioni che verranno in seguito interpretate dalla dyngen_code, che si occuperà di concatenare il codice macchina delle corrispondenti funzioni-host richieste, creando così il buffer eseguibile lato host che verrà mandato in esecuzione su richiesta nel ciclo della cpu_exec. Le funzioni utilizzate dalla disas_insn e dalle relative subroutines per reperire il od i bytes necessari, vengono generate in fase di preprocessing a partire da due

funzioni modello:

```
RES_TYPE glue(glue(ld, USUFFIX), MEMSUFFIX)(ulong ptr)
RES_TYPE glue(glue(lds, USUFFIX), MEMSUFFIX)(ulong ptr)
```

In base al valore definito di RES_TYPE, USUFFIX, MEMSUFFIX e DATASIZE nel file in cui si include lo header contentente questi modelli e tramite la macro *glue*, vengono generate le relative funzioni. Vediamo un esempio:

```
#define glue(x,y)     x ## y

#define RES_TYPE      unsigned char
#define USUFFIX       ub
#define MEMSUFFIX     _code
```

in fase di preprocessing verrebbe generata la seguente funzione:
unsigned char ldub_code(ulong ptr) che, nello specifico, si occupa di leggere un *unsigned byte* di codice all'indirizzo virtuale *ptr*. Includendo più volte il file contenente il modello di funzione, ma cambiando le definizioni di USUFFIX,MEMSUFFIX e quant'altro, vengono create tutte le possibili tipologie di funzione, sia come dichiarazioni che come specializzazioni del comportamento. Modificando dunque i due modelli, è possibile far si che le funzioni vadano a reperire i dati da un buffer controllato da noi piuttosto che dalla memoria del guest, dandoci così la possibilità di sfruttare la disas_insn per la creazione di meta-istruzioni congrue al contesto della CPU virtuale e corrispondenti al codice che vogliamo iniettare. Per adesso daremo per scontato che la funzione modificata capisca autonomamente quando comportarsi come suo solito e quando invece deve andare a leggere il codice da un'altra zona di memoria, anch'essa, per il momento, sempre disponibile per le sue necessità col codice occorrente in quel momento.

3.3 Generazione dei Translation Block

A questo punto, in linea teorica, siamo in grado di poter effettuare con successo la traduzione in codice host di una singola istruzione da iniettare. L'obiettivo però è

fare in modo che qualunque tipo e numero di istruzioni possano essere iniettate ad un qualsiasi indirizzo, perciò dobbiamo tenere ben presente che:

- Una traduzione viene interrotta quando viene incontrata un'istruzione o di salto o che altera lo stato della CPU in un modo che non può essere determinato staticamente
- QEMU può modificare i blocchi inserendo codice per la gestione dei salti diretti al codice del prossimo Translation Block
- QEMU può inserire nei blocchi il codice necessario per la generazione di interrupt simulati
- QEMU può inserire nei blocchi il codice per la gestione di eventuali parametri interni dipendenti dallo stato della CPU virtuale
- Possono essere presenti istruzioni di uscita dal blocco

Modificare un Translation Block aggiungendo del nostro codice precedentemente tradotto, potrebbe essere facilmente fattibile per istruzioni e contesti particolari, ma in caso di una situazione assolutamente generale con sistema complesso di istruzioni da iniettare ed in cui vogliamo rendere superfluo l'intervento dell'analista, effettuare modifiche richiederebbe quantomeno un'analisi della semantica ed una parziale ricostruzione del codice già presente nel Translation Block per poter integrare correttamente il codice host delle istruzioni da iniettare. Tutto questo, oltre ad essere piuttosto laborioso, avrebbe un costo non indifferente anche in termini di prestazioni.

Sfruttando come traduttore quello fornito da QEMU, dobbiamo assogettarci anche ad alcune delle sue regole, in particolare all'interruzione della traduzione in caso di salto o di alterazione dello stato statico della CPU, per cui, anche se facessimo tradurre un blocco di un qualsiasi codice da iniettare, come output avremmo un numero, non automaticamente predicibile senza una precedente traduzione, di spezzoni di codice host pari al numero di interruzioni del traduttore. In parole povere, dato uno spezzone di codice *potremmo* ottenere i Translation Block che ne riproducono il comportamento nell'ambiente virtuale.

Le funzioni di QEMU che si occupano del reperimento del prossimo Translation Block richiesto dal flusso esecutivo, sono essenzialmente due: *tb_find_fast* e *tb_find_slow*. La tb_find_fast ricerca all'interno della cache dei Translation Block se è già disponibile un elemento per il valore del program counter richiesto, se non lo è viene richiamata la tb_find_slow che si occupa di generarlo ed una volta fatto la tb_find_fast penserà poi ad aggiungerlo alle liste di gestione interne ed alla cache. Dobbiamo fare in modo che la tb_find_fast faccia generare i Translation Block relativi al codice da iniettare e poi li immetta nel flusso generale di esecuzione di QEMU, cosicchè possano essere creati, ottenuti ed eseguiti in maniera estremamente pulita ed assolutamente indipendente dalla natura del codice stesso. Come detto all'inizio di questo capitolo, non possiamo memorizzare alcunchè all'interno della memoria del guest, perciò siamo costretti a memorizzare il codice in un'apposita struttura, l'*injection_node*. Le funzioni menzionate nella sezione precedente, sempre nell'ipotesi che sappiano autonomamente quando variare il loro comportamento, andranno a leggere il codice da tradurre da un buffer di iniezione presente in un *inject_point* contenuto nell'injection_node associato al processo corrente[1].

[1] Vedremo in seguito il ruolo e la struttura esatta di queste entità

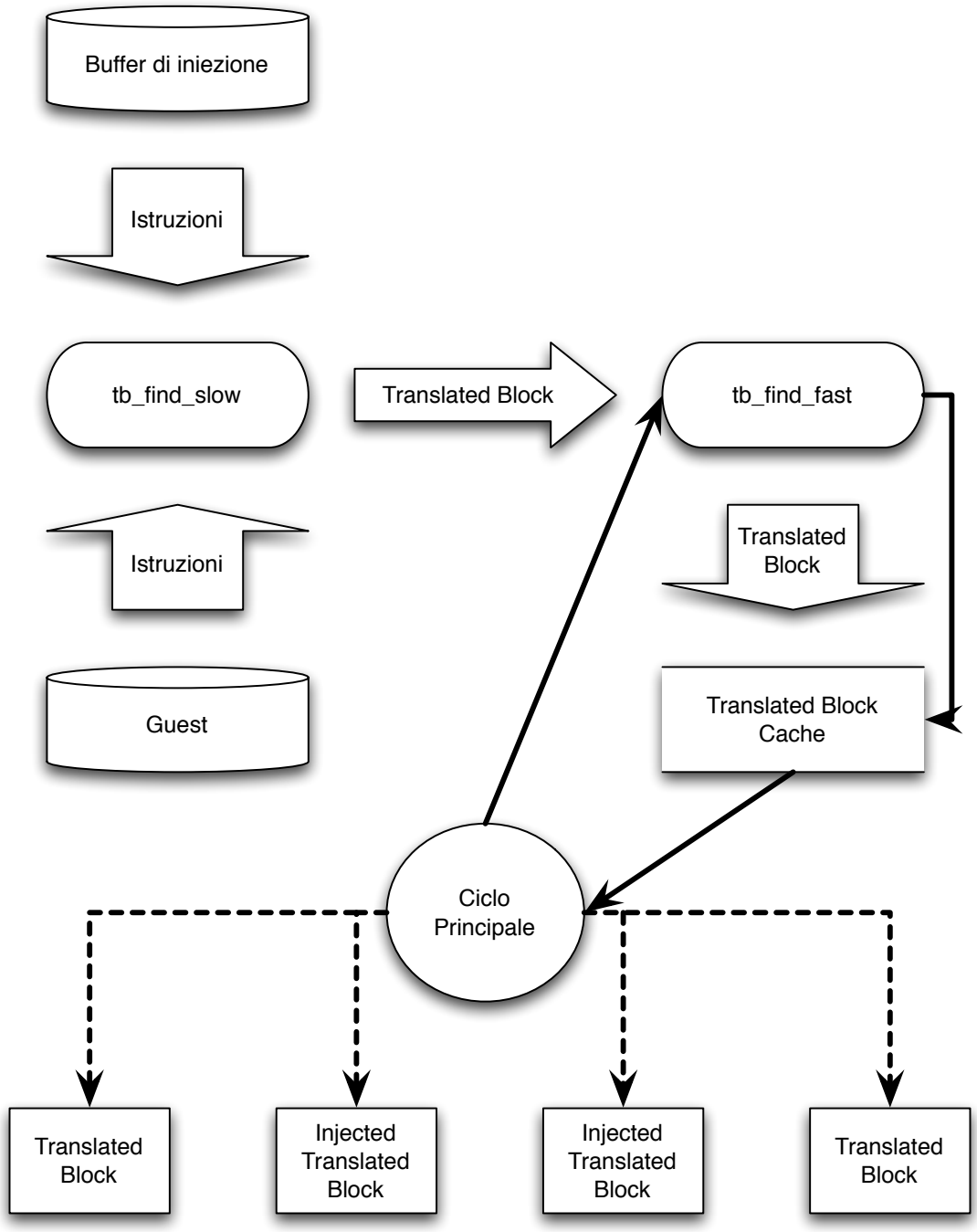

Figura 3.1: Illustrazione del sistema di reperimento ed esecuzione dei blocchi in caso di iniezione

3.4 Dirottamento della Memory Management Unit virtuale

L'unica informazione di cui necessita la tb_find_fast per poter funzionare correttamente è il valore del program counter per cui si sta richiedendo un Translation Block. Questa soluzione è senza dubbio alcuno ottima, tuttavia dal nostro punto di vista è un problema dato che il codice da iniettare, essendo assolutamente esterno alla memoria del guest e volendo noi rimanere totalmente indipendenti dal sistema operativo, non ha associato alcun indirizzo virtuale o fisico. Non potendo interferire con l'ambiente virtuale, siamo obbligati a far si che anche il nostro codice ottenga un indirizzo virtuale valido, perciò l'idea di base sarà la seguente: se un flusso esecutivo, per cui sono presenti degli inject point, raggiunge il punto in cui deve essere attivata un'iniezione, la MMU virtuale inizia a restituire dati differenti, come se ad un indirizzo virtuale corrispondessero più pagine fisiche contemporaneamente.

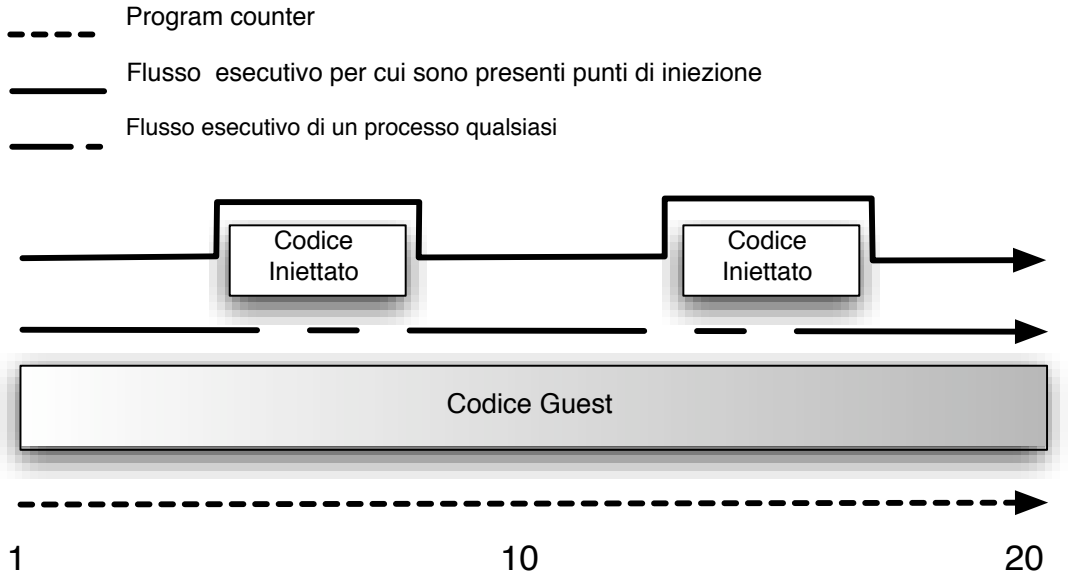

Figura 3.2: Codice diverso può essere eseguito allo stesso indirizzo in base a chi è l'esecutore

Così facendo, potremmo *riutilizzare* degli altri indirizzi virtuali ed associarli al nostro codice durante l'esecuzione. All'inizio del capitolo, abbiamo fatto una panoramica sulle funzioni di reperimento dati richiamate, direttamente od indirettamente, dalla disas_insn. In QEMU quelle funzioni fungono da MMU, per cui, dirottandole come precedentemente esposto, siamo in grado di manipolare i dati recuperati. Adesso è giunto il momento di vedere il sistema di gestione degli inject point e degli injection node; innanzitutto eccone la struttura

```
struct inject_point {

        void            *address;
        unsigned char   *ptr_to_code_to_inject;
        target_ulong    codesize;
        target_ulong    codeinit;
        target_ulong    baseaddr;
        unsigned short  gtec; /* going to exec call */

};

struct injection_node {

        struct inject_point  injections[INJECTORS_NUMBER];
        struct inject_point  *current_injection;
        tanalyzer_hlist_node    list;
        target_ulong            cr3;

};
```

Il significato dei campi dell'inject point:

- In *baseaddr* è memorizzato il primo indirizzo virtuale che andremo a riutilizzare per poter far eseguire l'inject point

- *ptr_to_code_to_inject* è un puntatore ad una zona di memoria contenente il codice macchina che vogliamo iniettare
- *codesize* ci dice la dimensione della zona puntata da ptr_to_code_to_inject
- *codeinit* ci dice a che spiazzamento dall'inizio di ptr_to_code_to_inject si trova la prima istruzione da mandare in esecuzione
- *gtec* sta per *going to exec call* e vale 1 se lo spezzone di codice iniettato che si sta per eseguire contiene una call, 0 altrimenti

e dell'injection_node:

- *cr3* il CR3 appartenente al flusso esecutivo che ci interessa manipolare
- *list* struttura per la gestione della lista di injection_node
- *current_injection* puntatore all'inject point che sta venendo processato, NULL altrimenti
- *injections* array degli inject point associati al CR3

Le funzioni che fungono da MMU possono essere controllate come segue: l'unico parametro che gli viene passato è l'indirizzo virtuale da cui leggere, tuttavia possono (ovviamente) accedere allo stato della CPU virtuale e così conoscere anche il CR3 di chi sta tentando di effettuare la lettura. Tramite un rapido controllo nella *lista degli injection_node* è possibile sapere se ve ne sono per il CR3 corrente, inoltre se il campo current_injection non è nullo, significa che effettivamente siamo in un contesto di iniezione. A questo punto l'ultima cosa che servirebbe sapere è se per l'indirizzo richiesto debba essere effettuato un dirottamento. Questo lo si può facilmente capire dall'indirizzo stesso: se il suo valore è compreso tra il baseaddr ed il baseaddr+codesize dell'inject point allora serve una lettura da ptr_to_code_to_inject e non dalla memoria guest. Più precisamente, serve una lettura al byte numero *indirizzo richiesto-baseaddr* del buffer di iniezione. Quando sta avvenendo un dirottamento della MMU, il flusso esecutivo in questione è effettivamente impossibilitato dall'accedere ai dati guest da

baseaddr fino a baseaddr+codesize, un qualsiasi tentativo di accesso a quella zona porterebbe ad un accesso al buffer di iniezione; tuttavia, come vedremo tra poco, questo fatto non comporta effettivi svantaggi, anzi, ci consente con qualche piccola modifica, di poter garantire il funzionamento del codice automodificante. Come anticipato, apparentemente potrebbe sembrare un problema non poter accedere ad alcune zone dello spazio di indirizzamento, ma bisogna tenere presente che:

1. Non tutti gli indirizzi virtualmente validi hanno *mapping* valido, ovvero corrispondono ad aree di memoria non utilizzate dal flusso

2. Non tutti gli indirizzi dello spazio di indirizzamento sono effettivamente utilizzabili per via della divisione kernel/user

3. Il dirottamento è attivo solo durante l'esecuzione del codice iniettato; una volta terminato, viene disattivata la modalità di iniezione

Questo significa che abbiamo un ampio numero di indirizzi riutilizzabili non sfruttati dai flussi e che comunque, anche se venissero utilizzati, non risconteremmo alcun problema a meno che non sia il codice iniettato stesso a tentare di accedere ai dati che vi sono presenti. Ad esempio, se da un'iniezione tentassimo di eseguire la funzione *func* del guest ed avessimo volontariamente scelto come baseaddr l'indirizzo di *func* per quell'inject point, allora verrebbe mandato in esecuzione nuovamente il codice da iniettare e non la reale *func*. In caso il baseaddr non venga scelto in modo così scriteriato, nessun range di indirizzi ci è precluso[2]. Nell'improbabile caso in cui sia richiesta una gestione più capillare dei range di indirizzi riutilizzati dai vari inject point per uno stesso flusso, al momento di capire se un determinato indirizzo per cui è stata richiesta una lettura appartenga o meno al range di un inject point ed in quello di calcolare a che spiazzamento dall'inizio del buffer di iniezione debba essere effettuata la lettura, vengono richiamate delle customizable function: tramite queste funzioni è possibile rendere virtualmente contigui, per l'iniettore, spazi di indirizzi non virtualmente contigui. Vediamo un esempio: se dovessimo iniettare del codice di dimensione

[2]L'unica eccezione sarebbe il range baseaddr/baseaddr+codesize, ma varrebbe solo nel caso in cui il punto di rientro nel codice guest sia compreso in quello spazio, ricadendo cosi nel caso di un baseaddr scriteriato

20 all'indirizzo 10, ma dal 20 al 25 non fosse possibile effettuare alcun dirottamento perchè presente del codice che vogliamo poter richiamare, basterebbe effettuare un override delle customizable function per ottenere uno schema comportamentale simile a questo:

Algorithm 2 Injector: Handle BrokenSpaces

```
if per il CR3 corrente è attivo un inject point E CUSTCALL(controlla se 10 <=
indirizzo richiesto < 20 O 25 < indirizzo richiesto <= 35) then
  if 10 <= indirizzo richiesto < 20 then
    ritorna ptr_to_code_to_inject + indirizzo richiesto - baseadder
  else
    ritorna ptr_to_code_to_inject + CUSTCALL(ricalcola offset con indirizzo e
    CR3)
  end if
end if
```

3.5 Gestione degli inject point

A questo punto, il sistema di gestione di QEMU del codice guest poggia su un nuovo strato virtuale posizionato sopra la memoria virtuale emulata e completamente controllabile da noi; potendo controllare il valore del program counter per cui la tb_find_fast richiede un Translation Block, siamo in grado di far tradurre il nostro codice correttamente, ma è necessario regolamentare il modo e l'ordine in cui vengono richiesti, tradotti ed eseguiti. Per poter fare ciò, dovremo intervenire sul ciclo della cpu_exec facendole assumere a grandi linee il seguente comportamento:

- Se il valore del program counter corrente è un inject point per il processo corrente
-
  - Svuota la cache dei Translation Block per assicurarti che venga lanciata una traduzione per reperire il prossimo
  - Crea un nuovo Translation Block per il nostro codice
  - Sostituisci lui ed il suo codice a quello precedentemente trovato

- Ricordati se stai per eseguire codice iniettato
- Esegui il codice
- Se sei nell'ambito dello stesso processo, hai eseguito codice iniettato, ora devi eseguire codice del guest e non stai effettuando una call ad una funzione del guest
- – Significa che si sta *saltando* via dal codice iniettato, perciò non è piu necessario mantenere attiva la modalità e dimenticati di avere eseguito un'iniezione
- Se hai appena eseguito una call nel codice iniettato
  - Dimenticati di averlo fatto in modo da non influenzare anche il prossimo ciclo con un'informazione relativa a quello corrente
- Se il codice iniettato non è finito e devi ancora eseguirne delle porzioni, oppure ne hai eseguita una in questo ciclo
- – Svuota la cache dei Translation Block per essere sicuro che vengano lanciate nuove traduzioni e che non vengano utilizzati spezzoni di nostro codice come elementi in cache

che viene formalizzato dal seguente algoritmo:

Algorithm 3 Injector: Infect Execution

while Ho codice guest da eseguire **do**
Richiama la tb_find_fast per ottenere il prossimo Translation Block
if Il valore del program counter relativo al Translation Block che sto per eseguire corrisponde ad un inject point per il CR3 corrente **then**
Attiva la modalità di injection per il CR3 corrente facendo puntare il campo current_injection all'inject point che dobbiamo far eseguire
Salva il valore corrente del program counter della CPU ed imposta il valore contenuto nella CPU ad un valore pari al baseaddr+codeinit contenuti nell'inject point
Svuota la cache dei Translation Block per far si che il prossimo blocco sia ottenuto tramite una nuova traduzione
Richiama la tb_find_fast e memorizza il nuovo Translation Block che ci viene restituito
Reimposta il valore del program counter nella CPU con quello salvato
Sostituisci il Translation Block restituito dalla prima chiamata alla tb_find_fast con quello appena ottenuto
Sostituisci il codice da eseguire con quello contenuto nel nuovo Translation Block
end if
if Il valore del program counter assegnato al Translation Block che sto per eseguire è un indirizzo riutilizzato **then**
ricordatelo
end if
Esegui il codice del Translation Block
if Il CR3 non è variato, hai eseguito codice iniettato, il nuovo valore del program counter non è riutilizzato e gtec vale 0 **then**
Dimenticati di aver iniettato codice
Disabilita la modalità di iniezione
end if
if gtec vale 1 **then**
resetta il suo valore
end if
if Il nuovo valore del program counter è riutilizzato oppure hai rilevato in inject point in questo ciclo **then**
Svuota la cache dei Translation Block
end if
end while

Per concludere, in fase di traduzione del codice da iniettare, avvengono tre operazioni supplementari che fino ad ora non abbiamo menzionato:

- Se si sta per lavorare su codice presente nel buffer di iniezione, disabilita il salto diretto tra blocchi
- Se il prossimo indirizzo che dovrebbe essere analizzato e tradotto, corrisponde ad un inject point, ferma la traduzione e ritorna il Translation Block, in modo tale che un nuovo Translation Block debba essere generato per finire l'operazione ed abbia come indirizzo di riferimento quello dell'inject point, dando cosi a noi la possibilità di rilevarlo nel ciclo principale della cpu_exec
- Il campo gtec dell'inject point viene posto ad 1 se si sta traducendo una call, istruzione che sappiamo essere l'ultima nel blocco per via della politica di traduzione di QEMU. Facendo eseguire il codice iniettato un blocco alla volta ed avendo disabilitato l'opzione di salto diretto tra blocchi durante la fase di traduzione, abbiamo la certezza di riuscire a seguire correttamente qualsiasi flusso di esecuzione.

Ora il sistema è in grado di inserire blocchi di codice con all'interno *qualsiasi* operazione, in *qualsiasi* punto di *qualsiasi* flusso, ciononostante, il lavoro non è ancora finito.

3.6 Supportare il codice automodificante

Fino ad ora ci siamo giustamente focalizzati sul riuscire a mandare in esecuzione il nostro codice, ma non sul garantirgli un funzionamento corretto: il nostro dirottamento della MMU è attivo nella fase di reperimento dati dello host verso il guest, ma non riguarda le operazioni si svolgono all'interno del guest. Nello specifico, traducendo il codice siamo in grado di far leggere dal buffer di iniezione come se fosse nel guest, ma se il codice che sta venendo tradotto fosse della forma

```
mov $0x1,0xdeaddead
```

l'indirizzo 0xdeaddead verrebbe interpretato come se fosse guest, costringendoci perciò ad allocare dello spazio all'interno della memoria guest per poter riporre eventuali variabili globali. Questo avrebbe contravvenuto alla regola che ci siamo imposti di indipendenza dal sistema operativo, perciò va adottata un'altra soluzione. Fortunatamente per noi, il meccanismo che dovremo utilizzare per risolvere questo problema identico a quello illustrato nelle sezioni precedenti: dovremo dirottare gli accessi alla memoria virtuale effettuati da alcune operazioni della CPU virtuale. Come visto in precedenza, la istruzioni della CPU virtuale sono implementate tramite funzioni lato host che vengono poi concatenate, perciò è sufficiente modificare il listato di queste funzioni ed inserire sia il codice per il controllo degli indirizzi riutilizzati, sia il codice per leggere/scrivere in un determinato indirizzo nel caso appartenga ad una zona di un inject point. Queste funzioni vengono generate al momento della compilazione con il medesimo meccanismo di quelle per la lettura dal guest[3], per cui basta inserire, in poche funzioni modello realizzate in assembly il codice necessario per effettuare i controlli ed eseguire le operazioni di lettura/scrittura richieste[4]. A questo punto, possiamo leggere/scrivere in zone appartenenti ad un inject point anche dal codice iniettato, permettendo così l'esecuzione di codice automodificante e l'utilizzo di variabili *globali*.

[3] ldub_code e simili per intenderci

[4] Essendo molte variabili memorizzate tramite l'ausilio dell'estensione per le variabili registro globali di gcc, questo comporta una penuria di registri disponibili al momento della compilazione, cosa che si traduce nell'impossibilità di gcc di compilare alcune funzioni. Per questo motivo le modifiche dovranno essere fatte direttamente in assembly

Capitolo 4

Il generatore di codice

In questo capitolo vedremo come realizzare uno strumento, funzionante sul sistema operativo Linux, che ci permetta di generare il codice da iniettare automaticamente avendo come input un sorgente in linguaggio C/C++

4.1 Organizzazione del codice

Alla fine dello scorso capitolo, ho detto che avremmo potuto utilizzare variabili *globali* intese come globali per il codice che andremo ad iniettare, ovvero che da qualsiasi suo punto vi si può accedere: senza avremmo avuto a disposizione solo variabili *automatiche*[1] per il nostro codice. Resta però il problema di dove posizionarle senza dover allocare della memoria all'interno del guest. Il problema è presto risolto facendo una considerazione: non essendoci alcun limite nè di dimensione nè di contenuto, potremmo far si che il nostro codice assuma una struttura a sezioni: prima una sezione libera da dati utilizzata dalle variabili globali e subito dopo la sezione codice, quella che deve essere eseguita.

[1] Allocate sullo stack in maniera automatica, le variabili locali di una funzione

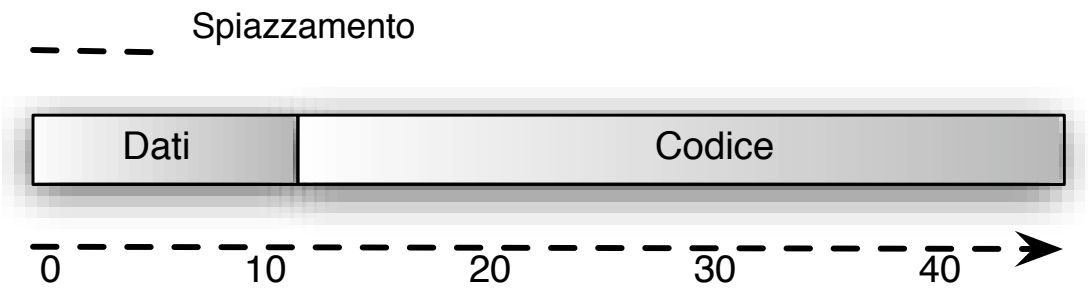

Figura 4.1: Struttura del codice da iniettare

Dato il seguente listato in linguaggio C di esempio:

```
int prova=1;
void dum(void) {
        prova =2;
}
int main(void) {
        int test=prova;
        prova=0;
        dum();
        prova=3;
        return (prova);
}
```

che dopo un normale processo di compilazione, assemblamento e linking verrebbe memorizzato nella forma

```
<sezione dati>
prova
<fine sezione dati>
...
<sezione testo>
codice delle funzioni
<fine sezione testo>
```

dovrebbe venire invece generato nel seguente modo

```
<sezione unica>
prova
codice funzioni
<fine sezione unica>
```

Con una struttura simile, acquista pieno significato il campo *code_init* dell'inject point visto in precedenza: ci indica a che spiazzamento si trova il codice da cui deve partire l'esecuzione, evitando così lo spazio riservato alla sezione dati artificiale ed il codice di altre funzioni.

4.2 Generazione del codice

Uno dei punti che ci eravamo imposti, era l'estrema semplicità di utilizzo da parte dell'analista, tuttavia, allo stato attuale delle cose, per poter iniettare del codice è necessario averne il relativo codice macchina, che deve essere precedentemente stato scritto in linguaggio assembly, compilato ed estratto. Questo potrebbe essere fattibile per piccole e semplici porzioni di codice, ma richiederebbe comunque sia una notevole quantità di tempo sia una certa abilità. L'ideale sarebbe poter convertire un file scritto in un linguaggio ad alto livello in un blocco di codice macchina già strutturato come descritto nella sezione precedente. Sfortunatamente non esistono software preconfezionati che possano effettuare questo genere di operazioni, dobbiamo progettarne e realizzarne uno noi. Dal momento che stiamo lavorando su Linux, dobbiamo prendere come riferimento la struttura dell'*ELF*[2]. Nell'ELF, la sezione contenente i dati statici inizializzati a valori *non nulli* si chiama *data*, mentre quella contenente il codice del programma si chiama *text*[8]. Noi dobbiamo far si che queste due sezioni vengano unite in una sola e sceglieremo *data*. Per far si che tutte le variabili globali/statiche vengano posizionate in quella sezione, dobbiamo assicurarci che nel sorgente che andremo a tradurre vengano inizializzate a valori non nulli, mentre per il codice che verrà posizionato nella sezione testo, dobbiamo tenere presente quanto segue:

[2]Execute and Link Format, il nome del formato di eseguibile utilizzato da linux

1. Andremo ad iniettare del codice dentro ad uno spazio di memoria in cui non possiamo assumere che sia mappata alcuna libreria

2. L'istruzione presente al baseaddr del'inject point non viene mai eseguita in nessun caso per via dell'attivazione dell'iniettore

3. Una volta terminata l'esecuzione del nostro codice, andrebbe in esecuzione il codice guest presente al prossimo valore del program counter

Per quanto concerne il primo punto la soluzione è tutto sommato semplice: non utilizzare alcuna funzione presente nelle librerie, se è richiesta una determinata funzione possiamo emularne il comportamento con una nostra e far iniettare anche quella, oppure, come nel caso si voglia chiamare una API di sistema, dobbiamo avere l'indirizzo della funzione nel guest e richiamarla tramite un puntatore a funzione, ad esempio. Per il secondo punto la soluzione è ancora piu semplice, basta posizionare alla fine del codice scritto da noi l'istruzione mancante, sempre che la si voglia far eseguire. Infine, per il terzo punto, non ci dobbiamo preoccupare: le routines che si occupano di ricevere i parametri per effettuare un'iniezione, generano dinamicamente un trampolino per far saltare l'esecuzione ad un indirizzo specificato dall'utente e lo accodano al codice da iniettare. Vediamo ora un esempio di file sorgente pronto per l'iniezione:

```
asm (".globl code_start\n\t" ".globl code_end\n\t"
         ".globl inject_init\n");

unsigned long test=1;

int prova(int val) {
        return (val * 2);
}

void inject_start(void) {
        int pippo;
        test=2;
```

```
        pippo=test;
        test=prova(pippo);
}
asm  ("inject_init:\n"
      "call inject_start\n"
      /* insert fixup code below here */
     );
```

nel primo inserto asm dichiariamo tre etichette globali di cui vedremo il significato in seguito, nel secondo invece chiamiamo la funzione principale, la *inject_start* e poi possiamo inserire, se lo desideriamo, l'istruzione non eseguita al momento dell'attivazione dell'inject point.

4.2.1 Il generatore

Per ottenere la struttura del codice che ci serve dobbiamo necessariamente intervenire nel processo di compilazione in questo modo:

1. Creiamo un file contenente una dichiarazione di sezione *data* artificiale

2. Accodiamo al file l'etichetta *code_start*. Vedremo in seguito il suo utilizzo

3. Lanciamo solamente una compilazione con gcc del file sorgente, senza perciò richiamare assemblatore e linker

4. Eliminiamo tutta la sintassi relativa alla creazione di sezioni nel listato assembly in modo tale da avere solamente un apparente blocco di codice

5. Accodiamo l'output del passo precedente e l'etichetta *code_end* al file creato in precedenza

6. Facciamo assemblare il file così prodotto

Fortunatamente il è tutto fattibile tranquillamente per mezzo di un semplice script. Procedendo in questo modo, oltre ad aver strutturato il codice del file oggetto prodotto nel modo in cui volevamo, abbiamo ottenuto un altro beneficio: nel file sono presenti anche le informazioni per la *rilocazione*[3] dei simboli contenuti. All'interno del codice prodotto, potrebbero esserci riferimenti a simboli per mezzo di indirizzi assoluti e pertanto non validi una volta iniettati all'interno del guest, dobbiamo perciò modificare questi riferimenti in base a *dove* andremo ad iniettare il codice che stiamo producendo, ovvero il baseaddr specificato al momento dell'inserimento di un inject point. Per raggiungere questo scopo abbiamo a disposizione due tipi di approccio: statico e dinamico.

Approccio Statico

L'approccio statico si basa sul far rilocare l'intera sezione data ed il suo contenuto[4] al linker al momento della compilazione.

Approccio Statico	
Vantaggi	Svantaggi
Rilocazione effettuata in toto dal linker	Serve rilinkare il codice oggetto per poterlo rilocare ad un indirizzo differente

Se si sceglie questo tipo di approccio, bisogna linkare il nostro codice oggetto, specificando di posizionare la sezione *data* all'indirizzo che più preferiamo, ad un qualsiasi altro codice oggetto contenente nella sezione *testo* codice in grado di emettere come output sia un numero che ci dica la distanza dall'inizio della sezione *data* del simbolo *code_init*, per poter impostare correttamente l'inject point, sia tutto il codice presente dall'inizio della sezione *data* fino all'indirizzo del simbolo *code_end*, che avevamo precedentemente posizionato in fondo per questo scopo, conoscere dove terminavano i nostri dati. Le etichette globali dichiarate nel file sorgente servono unicamente per risolvere problemi di simboli al momento di linkare insieme i due codici oggetto.

[3] Assegnazione ad una sezione od ad un simbolo dell'indirizzo che avrà al momento dell'esecuzione

[4] Come già detto prima, sono state memorizzate anche le informazioni per fare questo nel file oggetto

Approccio dinamico

L'approccio dinamico è un po' più complesso di quello statico e si basa sul rilocare i simboli del file oggetto manualmente. Ovviamente non andremo a replicare il funzionamento di un linker, utilizzeremo invece una tecnica empirica dai risultati assolutamente soddisfacenti.

Approccio Dinamico	
Vantaggi	Svantaggi
È possibile rilocare il codice ad un indirizzo differente senza dover rilinkare	Rilocazione da effettuare manualmente tramite euristiche
Possibilità di rilocare il codice al volo ad un indirizzo ottenuto durante l'esecuzione	*Teoricamente* non affidabile al 100%

L'approccio dinamico sfrutta anche l'etichetta *code_start* inserita in precedenza per poter conoscere l'indirizzo di inizio del codice da elaborare e così calcolarne le dimensioni. L'algoritmo ha bisogno di conoscere il punto di inizio e fine del codice e dell'indirizzo a cui vogliamo che il codice sia rilocato. Per assicurarci che tutti e solo

Algorithm 4 Generator: Relocate Code

$size = code\_end - code\_start$ per calcolare le dimensioni del codice
$relocs = 0$
$indirizzobase = indirizzobase - code\_start$
for $ptr = code\_start$ to $ptr < code\_start + size$ **do**
  $localptr = ptr$
  if $*localptr >= code\_start$ and $*localptr <= code\_start + size$ ovvero se il valore puntato da localptr è compreso all'interno degli indirizzi del codice **then**
    $*localptr = *localptr + indirizzobase$ per sommare al valore trovato la differenza tra indirizzo base e code_start
    $ptr+ = 3$ perchè abbiamo scritto 4 bytes e dobbiamo evitare di rileggere codice appena modificato
    $relocs++$ così possiamo tenere traccia di quante rilocazioni abbiamo effettuato
  end if
  $ptr++$
end for
ritorna il numero di rilocazioni

i simboli vengano modificati, dobbiamo avere un altro codice oggetto che dopo aver eseguito l'algoritmo esposto modificando così il codice precedentemente prodotto, ci restituisca il numero di rilocazioni. Linkando più volte i codici oggetto ad indirizzi base per l'eseguibile diversi, se si ottengono sempre lo stesso numero di rilocazioni possiamo avere una ragionevole sicurezza che tutto sia andato bene, potendo così provvedere a stampare il code_init ed il codice rilocato pronto per essere iniettato. Naturalmente, anche questa procedura può essere automatizzata tramite l'ausilio di uno script.

Capitolo 5

Conclusioni

Nel presente lavoro di tesi è stata presenta la struttura ed il funzionamento di un sistema che permetta ad un analista di poter monitorare e manipolare in qualsiasi modo un qualunque flusso di esecuzione, non necessariamente un processo, che sia in spazio kernel od in spazio utente. La struttura realizzata permette di ottenere un alto grado di personalizzazione del software pur mantenendo tra gli obiettivi primari l'aspetto prestazionale. È possibile registrare plug-in che consentono di aggiungere comandi a QEMU, inserire punti di osservazione, punti di manipolazione, punti di iniezione di codice senza praticamente alcun limite, sostituire alcune funzioni di QEMU per modificarne il comportamento, lavorare con API e strutture interne di QEMU, tradurre automaticamente un listato in linguaggio C/C++ in codice iniettabile direttamente nella virtual machine. Aggiungendo comandi possiamo automatizzare alcune operazioni facendo in modo che sia QEMU a farsi carico del lavoro da svolgere od inserirne di nuove magari basate sui plug-in, tramite plug-in infatti possiamo effettuare interventi su QEMU e sul codice che sta venendo eseguito nella virtual machine. Possiamo sviluppare nuovi plug-in basandoci sulle API e sulle strutture che ci vengono messe a disposizione da altri plug-in, permettendo così di ottenere alte percentuali riutilizzo del codice. Tramite le customizable functions possiamo modificare sia manualmente sia in maniera automatica il comportamento delle relative funzioni, attivando e disattivando le modifiche a piacere, con il generatore di codice invece siamo in grado, dato un sorgente scritto in linguaggio ad alto livello, di far si che venga tradotto in codice

macchina pronto per essere inserito in un flusso esecutivo della virtual machine tramite l'iniettore. Con i punti di manipolazione e monitoraggio, possiamo controllare e modificare lo stato della CPU virtuale ed il valore di qualsiasi dato presente nel guest, come gli argomenti delle funzioni od i valori di ritorno, possiamo inoltre effettuare un tracciamento ed un'analisi completa delle funzioni richiamate da un determinato flusso eliminando ogni possibile ambiguità. Infine, tramite l'iniettore, possiamo inserire qualsiasi codice all'interno di qualsiasi flusso di esecuzione all'interno del guest, eseguire operazioni supplementari o non farne eseguire altre, sostituire intere funzioni od anche solo ottenere dei wrapper assolutamente invisibili dall'interno della virtual machine. Per la realizzazione del sistema dei plug-in si è scelto di adottare un approccio basato sulle librerie condivise per questioni di semplicità, libertà di utilizzo, possibilità di interazione sia col programma principale sia con altri plug-in, cosa che ha permesso di realizzare le customizable functions. L'infrastruttura per la gestione dei punti di osservazione e di modifica è stata realizzata mediante l'impiego di liste di callback che vengono sfruttate aggiungendo/rimuovendo elementi così da poter monitorare, sostituire e manipolare ad hoc i comportamenti di analisi, reperimento e manipolazione dati. L'iniettore è stato realizzato mediate l'inserimento on demand di N layer di memoria virtuale sopra alla memoria virtuale guest. Per realizzarli è stato necessario effettuare un dirottamento della MMU del guest, per far si che quando si è nel contesto di un inject point il mapping fisico di un indirizzo virtuale guest cambi e corrisponda ad un indirizzo di memoria virtuale host in cui è presente il codice assembly da iniettare inserito dall'analista. L'iniettore si disattiva automaticamente tramite un controllo comportamentale del flusso, rilevando perciò quando si sta uscendo definitivamente dalla zona controllata dall'inject point. Per facilitare fino all'estremo il compito dell'analista è presente un generatore di codice, uno strumento che si occupa di convertire un file contenente un codice ad alto livello nel corrispettivo codice macchina pronto per essere iniettato all'interno della macchina virtuale, in qualsiasi flusso ed in qualsiasi suo punto, facendosi carico della rilocazione dei simboli e quant'altro. Nel complesso, si ritiene che il sistema creato sia perfettamente in grado di soddisfare in maniera rapida, solida, efficiente ed estremamente semplice ogni richiesta che possa essere ragionevole fare ad un framework di questa natura.

Bibliografia

[1] A. V. Aho and J. D. Ullman. *Principles of Compiler Design*.

[2] Amd. *AMD64 Architecture Programmer's Manual volume 2: System Programming*. Amd, September 2006.

[3] U. Bayer, C. Kruegel, and E. Kirda. Ttyanalyze: A tool for analyzing malware.

[4] F. Bellard. Qemu, a fast and portable dynamic translator.

[5] Bochs. http://bochs.sourceforge.net/.

[6] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2005.

[7] T. Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*, May 1995.

[8] D. Elsner, J. Fenlason, and friends. Using as the gnu assembler.

[9] P. Ferrie. Attacks on virtual machine emulators.

[10] P. Ferrie. http://pferrie.tripod.com/#atlantis.

[11] G. P. F. S. Foundation. http://gcc.gnu.org/.

[12] M. Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2004.

[13] Intel. *IA-32 Intel Architecture Optimization Reference Manual*. Intel Corporation, June 2005.

[14] Intel. *IA-32 Intel Architecture Software Developer's Manual vol 1: Basic Architecture*. Intel Corporation, June 2005.

[15] Intel. *IA-32 Intel Architecture Software Developer's Manual vol 2: Instruction Set Reference*. Intel Corporation, June 2005.

[16] Intel. *IA-32 Intel Architecture Software Developer's Manual vol 3: System programming guide*. Intel Corporation, June 2005.

[17] Intel. *Intel® Virtualization Technology Specification for the IA-32 Intel® Architecture*. Intel Corporation, April 2005.

[18] Kernighan and Ritchie. *Il Linguaggio C*.

[19] Levit. *Linkers & Loaders*. Morgan Kaufmann, 2000.

[20] Norman. http://www.norman.com.

[21] Parallels. http://www.parallels.com/.

[22] M. E. Russinovich and D. A. Solomon. *Microsoft® Windows® Internals*. Microsoft Press, 2004.

[23] W. Stallings. *Computer Organization and Architecture*. Prentice Hall, 2003.

[24] A. S. Tanenbaum. *Structured Computer Organization*. Prentice Hall, 1998.

[25] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2001.

[26] A. S. Tanenbaum. *Operating Systems Design and Implementation*. Prentice Hall, 2006.

[27] Vmware. http://www.vmware.com.

[28] Xen. http://www.xen.org/.