Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Triennale in Informatica

Studio e realizzazione di un meccanismo per la sospensione, riattivazione e migrazione di processi per il sistema operativo Linux

Tesi di Laurea in Sistemi Operativi

Relatore:
Renzo Davoli

Presentata da:
Michele Alberti

Seconda Sessione
Anno Accademico 2008/2009

Indice

1 Quadro generale **5**

1.1 Checkpoint/Restart Systems in Linux . . . 5

1.2 Tipologie di implementazione . . . 6

1.3 Il file di *checkpoint* . . . 9

1.4 Scenari di utilizzo e benefici . . . 15

1.4.1 In ambito Cluster . . . 15

1.4.2 In ambito Desktop . . . 16

1.5 Alcune precisazioni . . . 17

2 CryoPID, a process freezer for Linux **19**

2.1 Il progetto . . . 19

2.2 Caratteristiche progettuali . . . 20

2.3 Il design in generale . . . 23

2.4 La fase di *checkpoint* in CryoPID . . . 24

2.4.1 Gli “strumenti del mestiere” . . . 24

2.4.2 Operazioni preliminari . . . 27

2.4.3 Le informazioni da “congelare” . . . 27

2.4.4 Alcuni trucchi in *cryopid* . . . 34

2.4.5 Operazioni finali . . . 37

2.5 Il layout del file immagine . . . 38

2.5.1 Il file *stub* . . . 39

2.5.2 La scrittura delle informazioni “congelate” . . . 41

2.5.3 Una `malloc()` alternativa . . . 42

2.5.4 Il “trampolino” . . . 44

2.6 La fase di *restart* in CryoPID 46
2.6.1 Operazioni preliminari 47
2.6.2 Ripristino delle informazioni “congelate” 48
2.6.3 Operazioni finali 53
2.7 La prima “patch” 54
2.8 Attuali limitazioni 55

3 Scherzare la *glibc* con un hack! 60
3.1 Un semplice test... 60
3.2 `getpid()` caching 62
3.3 Il problema 63
3.4 Un *hack* come soluzione 65
3.5 `getpid()` hack in CryoPID 67
3.6 `LD_PRELOAD`, una soluzione alternativa 74
3.7 Proof-of-concept: “byword” 76

4 Sviluppi futuri 78

5 Conclusioni 80

Introduzione

Tutti i lettori di questo documento avranno dovuto, almeno una volta nello loro vita, sospendere momentaneamente una attività per poi riprenderla in un secondo momento, magari anche diversi giorni dopo. Chiaramente tutto questo risulta conveniente soltanto se si riesce a riprendere il lavoro dal punto in cui lo si era lasciato.

Negli ultimi tempi si è cercato di trasportare tale filosofia in campo informatico spinti dalle necessità sorte con l'uso massiccio di sistemi virtuali quali VPS (Virtual Private Server) e dai sistemi di HPTC (High Performance Technical Computing).

Questa idea ha dato luogo ad implementazioni note col nome di "Checkpoint/Restart Systems", sistemi software capaci di salvare su dispositivi di memoria secondaria lo stato di una applicazione in esecuzione in modo tale da poterla riesumare in un secondo momento. Risultato di tale operazione è un file detto di "checkpoint", il cui contenuto informativo deve permettere una corretta operazione di ricostruzione dell'applicazione.

In generale il file deve racchiudere tutte le informazioni necessarie al ripristino delle risorse comunemente utilizzate dai processi in sistemi UNIX-like. Tra queste le ovvie riguardano lo stato dello stack, dello heap, della CPU (i registri general purpose, quelli floating point e tutti gli altri registri disponibili) fino ad arrivare alla gestione delle risorse utili per IPC (e.g., segnali, pipe, etc.), dei socket di rete e dei file (cfr. 1.3).

Va sottolineato che il fine ultimo di tali "Checkpoint/Restart Systems" non constiste soltanto nella ricostruzione dell'applicazione in sé ma soprattutto nel ripristino della sua esecuzione dal punto in cui ne era stata salvata l'immagine.

Le possibilità offerte da tali sistemi sono facilmente intuibili, alcune delle qua-

li sono particolarmente riconducibili agli ambienti in cui fault-tolerant e load balancing sono problematiche molto sentite[1]. Ne sono una prova i numerosi progetti di ricerca universitari e non sviluppati durante gli ultimi anni; va però puntualizzato che sono pochi i progetti sufficientemente completi e ancora meno numerosi quelli potenzialmente inseribili in un contesto professionale, almeno per quanto riguarda l'ambiente di riferimento di questa tesi, ovvero quello del software libero.

Nel capitolo 1, *Quadro generale*, vengono descritte in maggiore dettaglio le caratteristiche dei "Checkpoint/Restart Systems" e loro campi di applicazione. Inoltre si presentano la descrizione e il confronto tra i vari approcci implementativi riportando rispettivamente esempi di progetti effettivi.

Nel capitolo 2, *CryoPID - A process freezer for Linux*, viene descritto in dettaglio il progetto ispiratore del lavoro discusso in questa tesi. Viene anche presentata la prima "patch" implementata per il codice dello stesso.

Nel capitolo 3, *Scherzare la glibc con un hack!*, sono illustrati i passaggi che hanno portato al completamento della seconda "patch" al codice del progetto per ovviare all'attuale implementazione della system call `getpid()` da parte della *glibc*[2].

Nel capitolo 4, *Sviluppi futuri*, viene riportata per sommi capi la "roadmap" futura del progetto, ovvero un elenco di idee che il team di sviluppo (di cui il sottoscritto fa parte attivamente) ha pensato per il miglioramento complessivo dello stesso.

Infine nel capitolo 5, *Conclusioni*, si analizzano i risultati ottenuti e l'intero processo lavorativo che ha permesso la produzione di questo elaborato.

Si è scelto di prendere in considerazione soltanto software libero, lo stesso progetto *CryoPID* è rilasciato sotto licenza BSD-like ed è comunque liberamente modificabile e redistribuibile.

In effetti questo lavoro di tesi, consistente per lo più in una serie di modifiche al sorgente originale di *CryoPID*, non sarebbe stato possibile se il progetto non fosse software libero.

---

[1]Si pensi ad esempio ai computer clusters, in special modo agli High-Availability clusters.

[2]GNU C library: http://www.gnu.org/software/libc/

Capitolo 1

Quadro generale

1.1 Checkpoint/Restart Systems in Linux

I "Checkpoint/Restart Systems" (d'ora in poi CRS) sono sistemi software capaci di salvare in un file la descrizione di una applicazione in esecuzione (fase di *checkpoint*) e in seguito di utilizzare lo stesso per ripristinarla dal punto in cui ne era stata salvata l'immagine (fase di *restart*).

Va detto che l'utilità di *checkpoint/restart* sarebbe da attribuire al sistema operativo, ne sono un esempio i meccanismi implementati da SGI per il sistema operativo IRIX ed IBM per quello AIX (entrambi a codice chiuso).

L'attuale versione stabile del kernel Linux[1] non presenta tale caratteristica. In realtà, la comunità di sviluppatori si sta già mobilitando nell'implementare tale mancanza, portando nel kernel diverse novità quali "namespaces" e "containers". È in sviluppo un insieme di meccanismi, noti come control groups (cgroups), in grado di aggregare e partizionare insieme di task, in modo gerarchico, assegnandogli particolari comportamenti.

Tra i sottosistemi implementati vi è l'interessante "freezer-subsystem" in grado di "congelare" e poi riattivare uno o più cgroups. La parte mancante, perché lo si possa considerare effettivamente un CRS, è la fase di memorizzazione in un file, prevista per il futuro prossimo.

[1] Kernel Linux versione 2.6.30.4, http://kernel.org/

La grande diffusione dei sistemi GNU/Linux[2], soprattutto in campi quali HPTC (High Performance Technical Computing) e Computer Clusters, ha spinto nel frattempo la ricerca universitaria e non a concepire e in seguito sviluppare soluzioni al problema.

1.2 Tipologie di implementazione

Le implementazioni per *checkpoint/restart* più diffuse si raggruppano in quattro categorie principali:

1. **Interna all'applicazione stessa.**

   Implementazioni di questo genere non rientrano propriamente fra i CRS proprio perché realizzate internamente e customizzate per una singola applicazione. In generale, queste implementazioni hanno lo scopo di salvare soltanto lo stato interno delle strutture dati dell'applicazione, in grado di fornirne una base di partenza al successivo avvio.

   Grosso limite di questo tipo di *checkpoint/restart* sono le restrizioni sulla libertà del momento del salvataggio dell'immagine. Tipicamente, il file di immagine può essere scritto soltanto in presenza di basso carico dell'applicazione e può capitare che passi diverso tempo prima che la richiesta di memorizzazione dell'immagine venga eseguita. Altro serio problema di tali implementazioni, è la mancanza di un meccanismo comune di *restart*.

   Dalla loro però hanno le alte prestazioni; infatti, il programmatore dell'applicazione conosce esattamente quali informazioni salvare e quali scartare, limitando inoltre al minimo la dimensione del file risultante dall'operazione di *checkpoint*.

   In conclusione, queste realizzazioni sono utilizzabili solo in quegli applicativi che non prevedono l'uso di particolari servizi forniti dal sistema operativo sottostante.

2. **Attraverso una libreria.**

[2] GNU/Linux è l'ambiente preso in considerazione in questa dissertazione.

Implementazioni di questo genere sono tra le più diffuse. Si distinguono in due sotto-categorie: le prime sono quelle che necessitano il linking dell'applicazione obiettivo alla libreria stessa (tipicamente per richiederne i servizi), le seconde non presentano questa forte limitazione ma hanno bisogno di particolari meccanismi di preloading tramite l'impostazione della variabile d'ambiente `LD_PRELOAD`[3].

Dal punto di vista utente, le realizzazioni di meccanismi di *checkpoint/restart* basati su libreria risultano scomode e, considerando soprattutto il primo approccio, spesso non utilizzabili in pratica. Si pensi a quelle situazioni in cui si dispone soltanto dei file binari degli applicativi.

A livello tecnico, il processo di memorizzazione dell'immagine è tipicamente implementato come gestore di uno specifico segnale. In certi casi, il processo di *checkpoint* potrebbe fallire in quanto l'applicazione obiettivo potrebbe semplicemente mascherarne il segnale tramite le system call `signal()` e `sigprocmask()`.

Nella maggior parte dei casi, vengono eliminate le limitazioni relative alla temporizzazione riscontrate nella prima metodologia implementativa (cfr. punto 1). Ulteriore vantaggio dei CRS basati su libreria è la presenza, il più delle volte, di un processo comune di *restart*.

Una delle caratteristiche più interessanti di questo tipo di realizzazioni è la possibilità di eseguire operazioni di *checkpoint* ad intervalli regolari, utile negli scenari in cui fault-tolerant è d'obbligo.

CKPT[4] e il meccanismo di *checkpoint/restart* del progetto Condor[5] sono esempi di implementazioni basate su questa filosofia.

3. **Applicativo autonomo.**

   Fanno parte di questa categoria tutte le implementazioni di *checkpoint/-restart* autonome e a livello utente.

[3] Meccanismo che permette di ridefinire un insieme arbitrario di chiamate di libreria (e.g., libreria standard *libc6*).

[4] http://pages.cs.wisc.edu/~zandy/ckpt/

[5] http://www.cs.wisc.edu/condor/

I CRS di questa categoria sono stati pensati per mantenere le stesse potenzialità di quelli basati su libreria cercando di risolverne gli inconvenienti (cfr. punto 2). In particolare, implementazioni di questo tipo risultano trasparenti e le meno invasive dal punto di vista dell'utente.

La trasparenza è una proprietà evidente dato che i CRS in questione non necessitano di alcun tipo di re-linking o modifica dell'applicativo sorgente (e.g., per richiedere i servizi della libreria), cosa che invece accade nelle implementazioni basate su libreria.

Quello della poca invasività è probabilmente il punto di maggior forza di questo genere di realizzazioni, soprattutto in confronto a quelle che richiedono modifiche al kernel del sistema operativo (cfr. punto 4).

CryoPID[6], oggetto primario di questo lavoro di tesi, appartiene a questa categoria di CRS (cfr. 2.1).

4. **Supporto kernel del sistema operativo.**

   I CRS implementati secondo questa metodologia sono i meno numerosi; freno principale è lo sviluppo software a livello kernel, notoriamente più complicato per scrittura e debugging.

   Gli approcci implementativi possibili sono lo sviluppo di moduli kernel oppure una serie di modifiche al kernel stesso. Entrambi permettono flessibilità e potenza notevolmente maggiori rispetto a tutte le altre categorie implementative. Il primo però consente agli utenti di utilizzarlo senza dovere applicare "patch", ricompilare e riavviare il proprio kernel.

   In generale le informazioni necessarie alla fase di *restart*, reperibili attraverso complicati meccanismi e operazioni non convenzionali nelle altre tipologie realizzative, sono nei CRS a livello di sistema rintracciabili molto facilmente. Semplicemente leggendole dalle rispettive strutture dati del kernel!

   Se da una parte risulta semplice e "pulito" reperire le informazioni necessarie dalle strutture dati del kernel, dall'altra questo comporta maggiori difficoltà e lunghi tempi di porting verso altri ambienti e piattaforme rispetto

[6] http://sharesource.org/project/cryopid/

agli altri approcci. Non solo, gli aggiornamenti interni ad un kernel sono giornalieri a differenza degli standard riguardanti le API (Application Programming Interface). Ciò rende necessariamente più complesso mantenere il codice aggiornato e funzionante a mano a mano che si susseguono nuove versioni del kernel.

I progetti effettivi sono tra i più completi e professionali in circolazione; tra questi i più famosi sono BLCR[7] e Zap[8], entrambi basati su moduli kernel (in realtà, presentano una parte minoritaria eseguita a livello utente).

1.3 Il file di *checkpoint*

Si è già accennato in precedenza dell'importanza fondamentale del file di *checkpoint* contenente tutte le informazioni necessarie ad un corretto ripristino dell'applicazione obiettivo. Si rende ora necessaria una descrizione dettagliata sulle risorse oggetto delle fasi di memorizzazione e di ripristino.

CPU (Central Processing Unit)

Ovviamente lo stato dei registri della CPU deve essere salvato per poi essere ripristinato. Quest'ultimo comprende instruction pointer, stack pointer, registri general purpose, registri floating point e tutti gli altri forniti dall'architettura.

In generale, i CRS implementati a livello kernel riescono facilmente avendo la possibilità di accedere direttamente alle strutture dati disponibili. Al contrario, le altre categorie implementative ricorrono a metodologie più complicate, quali l'uso massiccio della system call `ptrace()`. Quest'ultima fornisce un meccanismo di tracciamento di processi capace, tra le tante opportunità, di accedere ai registri della CPU permettendone così la memorizzazione e ripristino.

Naturalmente i CRS eseguiti a livello kernel del sistema operativo segnano le migliori prestazioni; non hanno il peso dei cambi di contesto, obbligati agli altri approcci implementativi, dovuti al meccanismo delle system call.

[7] https://ftg.lbl.gov/CheckpointRestart/CheckpointRestart.shtml
[8] http://www.ncl.cs.columbia.edu/research/migrate/

Spazio degli indirizzi

Lo spazio degli indirizzi contiene la maggiore quantità di informazioni riguardanti ogni programma in esecuzione. Tipicamente è costituito da una serie di regioni di memoria virtuale dedicate ad ogni processo riguardanti i dati inizializzati e non, lo heap, lo stack e tutte quelle zone mappate per utilizzi specifici (e.g., "memory mapped I/O"). Le informazioni più rilevanti, per ogni regione, sono il range degli indirizzi, la dimensione, i permessi d'accesso ed infine l'i-node del file o device mappato.

La memorizzazione e successivo ripristino dello stato delle risorse in questione non sono problemi particolarmente complessi per i CRS realizzati a livello kernel del sistema operativo; anche in questo caso si "interrogano" le strutture dati del kernel, quelle dedicate alla gestione della memoria virtuale. Contrariamente, i sistemi software basati su libreria oppure autonomi devono ricorrere ad espedienti che ne compromettono spesso anche le possibilità di adattamento alle diverse piattaforme. Il metodo più utilizzato in ambiente GNU/Linux è l'analisi dettagliata delle informazioni kernel esportate a livello utente dal file system virtuale `/proc`.

TLS (Thread-Local Storage)

I sistemi operativi moderni supportano più thread all'interno dello stesso processo, cioè la possibilità di eseguire concorrentemente due o più sequenze di operazioni. Questa caratteristica è ormai da tempo supportata anche dal kernel Linux.

I thread appartenenti allo stesso processo condividono il medesimo spazio degli indirizzi; ciò significa che, ad esempio, le variabili globali sono visibili a tutti i thread. Se da un lato le possibilità offerte sono notevoli, anche dal punto di vista delle prestazioni[9], dall'altro vi sono situazioni in cui questa "visione" globale non è adatta, anzi reca numerosi problemi.

Ne è un lampante esempio la gestione della variabile `errno`, contenente l'ultimo codice d'errore generato dall'ultima funzione che utilizza tale funzionalità. Se in un ambiente orientato al processo quest'ultima può essere gestita come va-

[9] Si ricorda che un cambio di contesto fra thread dello stesso processo risulta molto più veloce che tra processi distinti.

riabile globale, in uno orientato ai thread questa concezione risulta sbagliata. La soluzione attualmente utilizzata è la presenza di una variabile `errno` per ogni thread di ogni processo.

Tutte le risorse che vengono allocate per thread, vengono mantenute in una zona di memoria detta Thread-Local Storage.

Potenzialmente, l'applicazione obiettivo potrebbe avvelersi delle funzionalità offerte dai thread. Compito dei CRS è quello di carpirne struttura e risorse locali (e.g., stack, gestore dei segnali, etc.) per poi essere in grado di ricostruire l'applicativo originale.

Il supporto ai thread è una caratteristica non comune nei CRS attuali, soprattutto nel caso degli approcci a livello utente perché risulta di difficile implementazione.

Segnali

I segnali sono interrupt software. La maggior parte delle applicazioni non triviali necessitano della loro gestione. I segnali forniscono un meccanismo per la gestione di eventi asincroni, non di meno sono utilizzati come sistema di IPC (Inter-Process Communication) tra processi o thread.

Un buon sistema di *checkpoint/restart* deve essere in grado di gestirli in modo appropriato, riuscendo nella memorizzazione e successivo ripristino dei gestori dei segnali nonché dello stato dei segnali pendenti.

Tipicamente, i CRS implementati a livello utente riescono a memorizzare e ripristinare lo stato dei gestori dei segnali installati, attraverso le system call `signal()` e `sigaction()`. La lista dei segnali pendenti è invece gestibile tramite l'utilizzo della system call `sigpending()`. Va fatto notare che, soprattutto nel caso dei CRS autonomi, l'implementazione di questa caratteristica non risulta immediata; infatti, non esistono metodi formali e "puliti" per servirsi delle system call sopra citate per estrapolare informazioni riguardanti processi che non siano il chiamante. In generale, questo significa che i CRS devono "forzare" gli applicativi obiettivo ad eseguire tali funzioni di sistema per poi farsene restituire i risultati[10].

[10] Nella sottosezione 2.4.4 viene presentato lo stratagemma implementato in CryoPID.

Come per le precedenti risorse, i sistemi realizzati per lavorare a livello kernel, possono gestire tali informazioni servendosi direttamente delle adeguate strutture dati.

File (regolari) e File Descriptor

La gestione dei file e dei file descriptor pone diverse sfide per i software di *checkpoint/restart*. Queste due diverse astrazioni sono di utilizzo comune nella gran parte degli applicativi. Gli stessi sistemi di input/output sono costruiti sui file e file descriptor. Il loro ruolo centrale nei sistemi UNIX-like, pone forti obblighi di una corretta gestione nei CRS moderni.

I file sono gli "oggetti" più visibili agli utenti, notariamente la risorsa del sistema operativo più conosciuta da questi ultimi. Questo spesso equivale a situazioni soggette a molte variazioni, anche in tempi brevi.

Scenario tipico è la modifica di un file nel tempo che intercorre tra l'operazione di creazione del *checkpoint* e il rispettivo *restart*. Ancora peggio, non esistono metodi diretti per capire le possibili interazioni con il file system avvenute tra due o più immagini "catturate". Se un file è stato cancellato non vi sono metodologie in grado di recuperarne il contenuto.

Fino ad ora sono state implementate soltanto soluzioni parziali al problema, spesso risultanti in comportamenti sbagliati in determinate situazioni. Le soluzioni più comuni prevedono il salvataggio di copie nascoste dei file impiegati dall'applicazione obiettivo oppure il salvataggio del contenuto degli stessi nel file di *checkpoint*. In quest'ultimo caso andrebbero salvate altre importanti informazioni, quali il nome del file stesso, i relativi permessi, etc., per poi essere in grado di ricreare il file originale.

I file descriptor sono l'unico collegamento tra qualsiasi processo e i propri file e non solo. Quelli associati a file regolari dovrebbero essere riassegnati al momento del *restart*. Stessa cosa per quelli associati ai terminali, ad esempio per la gestione dell'input/output.

Le informazioni importanti al riguardo sono i valori dei flag, dei permessi di accesso e del file offset. In generale, i CRS basati su libreria usano il sistema di redirezione delle system call, come `stat()` e `fcntl()`, per memorizzarne

internamente una copia. In modo analogo operano i sistemi di *checkpoint/restart* autonomi. Le stesse system call `open()`, `close()`, `dup()` e `dup2()` sono poi utilizzate nella fase di ripristino dell'applicazione obiettivo, insieme a quelle citate in precedenza.

I CRS implementati a livello kernel del sistema operativo accedono direttamente alle strutture dati pertinenti ai moduli del file system.

Directory

La memorizzazione e il ripristino delle directory interessate costituiscono un problema non banale; se possibile risulta ancora più difficile che nel caso dei file, concepirne una politica di gestione. Alcune informazioni riguardanti le directory devono però essere salvate e ripristinate, su tutte quelle che riguardano la "current working directory".

Attualmente la quasi totalità dei CRS non supporta la gestione delle directory.

Socket (di rete)

Anche la gestione dei socket di rete risulta tra le più complicate, tanto che quasi la maggioranza dei CRS li ignora completamente.

Quando supportati, le implementazioni sviluppate presentano una disomogeneità marcata. Le implementazioni basate su libreria permettono, nella maggior parte dei casi, la chiusura dei socket attivi e la loro riattivazione mediante l'esecuzione di callbacks da parte dell'applicazione riesumata.

Tipicamente, durante la fase di *restart*, i CRS debbono ristabilire il collegamento con il server (l'altro capo della connessione) chiedendo di ricominciare la comunicazione dagli ultimi dati ricevuti correttamente dall'applicazione obiettivo.

I sistemi sviluppati a livello kernel sono, ovviamente, facilitati in questo compito. Ne è un esempio il sistema presente nel modulo kernel CRAK[11] per i socket TCP/IP. In fase di *checkpoint*, i socket vengono posti nello stato `TIME_WAIT`[12]

[11] http://www.ncl.cs.columbia.edu/research/migrate/crak.html

[12] Stato seguente la "session shutdown" del socket. Permette di trattare dei pacchetti di dati presenti sulla rete come validi se viene ricreata, in tempi adeguati, la stessa tupla che specificava la connessione chiusa.

per evitare la chiusura della connessione. Al momento del *restart*, CRAK crea nuovi socket modificandone, a livello kernel, la struttura per farli connettere al server remoto. Quest'ultimo poi viene fatto "indirizzare", tramite il meccanismo dei dati out-of-band, verso i nuovi socket.

Pipe e FIFO (named pipe)

Le pipe sono la forma più datata di System IPC (Inter-Process Communication) presente in tutti i sistemi UNIX-like. Sono sistemi con due limitazioni fondamentali; sono half-duplex (i.e., i dati scorrono soltanto in una direzione) e possono essere utilizzati soltanto fra processi che hanno un "antenato" in comune. La loro corretta gestione è fondamentale per il *restart* degli shell script, importante classe di applicazioni sopratutto nei sistemi batch.

Tipicamente, l'approccio dei CRS in circolazione nei confronti delle pipe è molto simile a quello per i socket di rete. Alcuni sistemi basati su libreria permettono l'esecuzione di callbacks all'applicativo obiettivo delegandogli il ripristino delle pipe coinvolte. Altri ancora memorizzano il valore dei file descriptor assegnati alle pipe per poi ricrearle, lasciando però all'applicazione obiettivo il compito di recuperare la comunicazione antecedente il *checkpoint*.

Le FIFO, conosciute anche col nome di named pipe, sono sistemi simili alle pipe ma ne risolvono la seconda limitazione. Le named pipe sono file veri e propri che permettono a processi non correlati di scambiarsi dati.

Le FIFO, per i sistemi di *checkpoint/restart*, presentano grosse difficoltà di gestione perché, essendo autentici file, uniscono le problematiche proprie delle pipe e dei file. La quasi totalità dei CRS sviluppati non le supportano.

Molte altre sono le informazioni che un completo CRS dovrebbe supportare, tra cui compaiono quelle riguardanti "system environment", "accounting", "resource limits", sessioni, credenziali dei processi, socket UNIX e System V IPC. Queste non sono state descritte in dettaglio perché alcune non presentano particolari problematiche per i CRS attuali (è il caso delle prime tre); altre perché troppo complesse e spesso non considerate affatto dai sistemi implementati finora (è il caso delle restanti quattro).

1.4 Scenari di utilizzo e benefici

In questa sezione vengono presentati alcuni scenari di utilizzo e relativi benefici derivanti dall'utilizzo di sistemi di *checkpoint/restart*.

1.4.1 In ambito Cluster

Alcuni interessanti utilizzi provengono dal mondo del "Cluster Computing", di seguito ne vengono proposti tre.

"Gang" Scheduling

La possibilità di eseguire il *checkpoint* ed in seguito il *restart* di una serie di processi paralleli facenti parte della stessa applicazione (gang) permette massima flessibilità di scheduling e maggiore utilizzo dell'intero sistema.

Gli amministratori di sistema hanno la possibilità di decidere politiche di scheduling differenti nell'arco dell'intero giorno (e.g., pesanti, lunghe computazioni di notte e piccole, interattive di giorno), semplicemente eseguendo il *checkpoint* degli applicativi obiettivo per poi ripristinarli al successivo cambio di politica.

Il *checkpoint/restart* consente anche il mantenimento di un sottoinsieme dei nodi oppure dell'intero cluster senza troppe interferenze con il lavoro degli utenti. Senza tale utilità le soluzioni sarebbero drastiche; terminare tutte le applicazioni utente (con perdita delle risorse utilizzate fino a quel momento) oppure impedire l'esecuzione di nuove applicazioni ed attendere la fine delle computazioni già attive (col sistema che non utilizza tutte le risorse disponibili).

Migrazione dei processi

Un buon CRS dovrebbe offrire la possibilità del *restart* di processi non solo nel nodo originale su cui si è eseguito il *checkpoint*, ma anche in un qualsiasi altro nodo del cluster (o in generale in un sistema diverso).

In generale, la migrazione di processi risulta una caratteristica essenziale in tutte quelle situazioni in cui si hanno informazioni su un imminente guasto di un nodo (e.g., ventola della CPU, problemi col disco locale, etc.) per cui risulta

necessario eseguire il *checkpoint* di tutte le applicazioni obiettivo e il seguente *restart* su nodi sani del cluster.

L'idea della migrazione, per lo più "a caldo", non è nuova. Progetti come OpenVZ[13] e Xen[14] permettono in generale la migrazione rispettivamente di VPS (Virtual Private Server) e di intere macchine virtuali; il vantaggio dato dai CRS è la possibilità di far migrare singole applicazioni.

Rollback della computazione

Infine, i sistemi di *checkpoint/restart* risultano molto utili in quelle situazioni in cui una lunga esecuzione è terminata per un errore mai presentatosi o comunque non deterministico (come infrequenti errori di sincronizzazione) e non si vuole perdere tutto la computazione eseguita fino a quel momento. Senza i CRS, l'unica soluzione possibile sarebbe ricominciare il lavoro dall'inizio. Contrariamente, disponendo di un sistema di *checkpoint/restart* eseguito ad intervalli regolari, la computazione può essere ripresa dall'ultimo file di checkpoint scritto. Non solo, vi è anche la possibilità di scegliere da quale punto della computazione ripartire.

1.4.2 In ambito Desktop

Tra i CRS implementati fino ad ora, vi sono anche quelli pensati per un uso quotidiano; quelli orientati alle normali applicazioni. Di seguito vengono proposti alcuni casi d'utilizzo e rispettivi benefici.

Salvare/Ripristinare la sessione di lavoro

Molti ambienti di sviluppo (ma non solo) non prevedono il salvataggio della sessione di lavoro obbligando l'utente alla sua ricostruzione ad ogni utilizzo. Grazie ai CRS ciò non è più necessario, basta eseguire il *checkpoint* dell'applicazione e il corrispettivo *restart* nel momento del bisogno.

[13] http://wiki.openvz.org/Main_Page

[14] http://www.xen.org/

Sostituto di “Prelink”

Prelink[15] è un meccanismo di prelinking delle applicazioni sotto Linux; permette un più veloce tempo di inizializzazione quando vengono invocate numerose e pesanti librerie dinamiche. Il limite maggiore di Prelink è che deve essere aggiornato ai cambiamenti dell’architettura di Linux.

Eseguire il *restart* di qualsiasi applicazione di cui si ha un *checkpoint* risulta di notevole velocità rispetto all’avvio “da zero” della stessa, soprattutto in quei casi in cui non si dispone di Prelink.

Debug

Le immagini di checkpoint risultano di notevole utilità in situazioni di debug di una applicazione. Non solo, eseguendo l’operazione di checkpoint ad intervalli regolari si riesce a ricostruire la “storia esecutiva” dell’applicativo in studio. Quest’ultima possibilità risulta ancora più utile se impiegata in combinazione con il rollback dell’applicazione stessa.

Studio dei dati di computazione

Esistono applicazioni (soprattutto nel campo della simulazione) che richiedono una prima fase di computazione particolarmente pesante seguita da una seconda di analisi di dati. La prima fase è generalmente eseguita su un dispositivo ad elevate prestazioni o su un cluster di computer, mentre la seconda può essere eseguita su di un normale calcolatore dopo il lavoro di *restart* di un qualsiasi CRS.

1.5 Alcune precisazioni

Per tutto il capitolo si è parlato di “applicazione” o “applicativo” obiettivo come l’oggetto sul quale eseguire l’operazione di *checkpoint* per poi ripristinarla in un secondo momento grazie alla fase di *restart*. Va chiarito che con applicazione si intende qualunque forma di programma in esecuzione. I lettori debbono

[15]Pacchetto per Debian Sid: http://packages.debian.org/unstable/admin/prelink

tenere presente che si tratta di un concetto completamente diverso da quello di "processo". Una applicazione software non è un processo!

Questo significa che potrebbe essere composta da, in generale, un "albero" di processi i quali potrebbero presentare una struttura a singolo o multi-thread. Non solo, l'applicazione potrebbe involvere numerosi file, socket di rete e non.

Tutto questo per precisare che, è vero, nessun CRS attuale è completo per qualsiasi situazione, ma è anche vero che il problema del *checkpoint/restart* è obiettivamente complesso da affrontare in ottica generale.

Capitolo 2

CryoPID, a process freezer for Linux

2.1 Il progetto

CryoPID[1], il progetto ispiratore di questo lavoro di tesi, è stato creato in ambito accademico dall'australiano Bernard Blackham.

Il nome del progetto vuole ricordare, in senso ironico ed evocativo, l'obiettivo di pensare di applicare la "criogenia"[2] al mondo informatico; in particolare CryoPID è un CRS, per ambienti GNU/Linux, eseguito interamente a livello utente. È capace di "congelare" lo stato di un processo in un file e, come per tutti i sistemi di *checkpoint/restart*, quest'ultimo può essere in seguito utilizzato per riesumare il processo dal punto in cui ne era stata salvata l'immagine, anche dopo un reboot o addirittura su un'altra macchina.

Va precisato che, attualmente, CryoPID è in grado di eseguire il *checkpoint/-restart* di singoli processi, oltretutto, costituiti da un solo thread di esecuzione.

[1] http://www.cryopid.org/

[2] Branca della fisica che si occupa dello studio, della produzione e dell'utilizzo di temperature molto basse (sotto i -150 °C) e del comportamento dei materiali in queste condizioni.

2.2 Caratteristiche progettuali

CryoPID è stato sviluppato riconoscendo le limitazioni più comuni nei CRS realizzati e volendolo implementare interamente a livello utente, si sono studiati e analizzati soprattutto i difetti riscontrati in quelli basati su libreria (cfr. 1.2).

Di seguito vengono descritte e valutate le linee guida che tutt'ora caratterizzano il progetto.

Applicativo autonomo

CryoPID nasce con l'idea di implementare un sistema di *checkpoint/restart* semplice, leggero ma potente allo stesso tempo e in grado di essere utilizzato negli scenari più disparati. Proprio per queste ragioni CryoPID è un CRS di tipo "applicativo autonomo", interamente a livello utente, che *non necessita* quindi di:

- **Modifiche al kernel.**

  Spesso gli utenti (o "utonti" ?!) finali di prodotti software non sono in grado, per motivi burocratici o meramente tecnici, di applicare "patch" o configurare il kernel Linux direttamente.

  CryoPID non necessita di tali operazioni e perciò risulta non invasivo ed utilizzabile nei più svariati contesti.

- **Modifiche al codice dell'applicazione.**

  Molti sistemi di *checkpoint/restart* basati su libreria richiedono la modifica del codice sorgente dell'applicativo obiettivo per permettergli l'utilizzo, tipicamente tramite chiamate a funzione, delle potenzialità del CRS. Questa possibilità molto spesso è difficile da sfruttare anche nel campo del software libero; risulta infatti non banale modificare il codice sorgente scritto da altri, soprattutto se la mole da studiare non è esigua. Inoltre, in molti ambiti professionali, non sempre si ha la possibilità di lavorare con software libero.

  CryoPID riesce nel suo scopo anche negli scenari sopra illustrati risultando utilizzabile in un numero di situazioni di gran lunga superiore rispetto ai CRS realizzati come libreria.

- **Re-linking dell'applicazione.**

  Il re-linking di un programma è sicuramente meno impegnativo rispetto alla modifica del codice sorgente dello stesso. Risulta comunque un problema irrisolvibile in quelle occasioni in cui si dispone soltanto dei file binari dei programmi in esame.

  Ancora una volta, CryoPID risulta utilizzabile evidenziandone una delle sue migliori caratteristiche: la trasparenza.

- **Utilizzo di `LD_PRELOAD`.**

  CryoPID, proprio perché applicativo autonomo, non necessita dell'impostazione della variabile `LD_PRELOAD` nell'ambiente in cui si esegue l'applicativo obiettivo. Il processo di *checkpoint/restart* risulta quindi molto comodo e veloce da utilizzare.

Privilegi di root.

Attualmente il progetto non necessita di particolari privilegi per eseguire il proprio compito; in generale, qualsiasi utente può "congelare" soltanto i processi che gli appartengono, a meno che l'utente in questione non sia "root".

Estendere la possibilità di utilizzo a qualsiasi utente costituisce una caratteristica implementativa fondamentale per ogni sistema di *checkpoint/restart*, di cui CryoPID è dotato.

Quest'ultima caratteristica fa di CryoPID un potente e versatile strumento, soprattutto se si pensa che nella maggioranza dei casi il soggetto "normale utente" è distinto da quello di "amministratore".

Multiarchitettura.

CryoPID è una applicazione multiarchitettura, ovvero supporta ben quattro architetture hardware diverse, quali x86, x86-64, Alpha e SPARC. Chiaramente, mantenere la compatibilità per un numero elevato di architetture non risulta semplice, ma permette una maggiore diffusione del progetto.

CryoPID è scritto, in maggior parte, nel linguaggio ANSI C perché riconosciuto come il linguaggio più adatto allo scopo e tra i più portabili. Sono presenti anche piccole sezioni scritte nel linguaggio assembly dell'architettura obiettivo per eseguire operazioni non ottenibili direttamente in C.

Attualmente vengono aggiornati e sviluppati soltanto le versioni riguardanti le architetture x86 e x86-64 perché accreditate come le più diffuse.

Supporto kernel.

CryoPID supporta le versioni del kernel Linux più utilizzate e ancora attualmente sviluppate, cioè le serie 2.4 e 2.6.

Facendo un semplice censimento all'interno della comunità degli utilizzatori del progetto si è notato, con moderato stupore, come sia ancora rilevante la percentuale di quelli che adoperano CryoPID in ambienti Linux con kernel 2.4.x.

Attualmente gli sviluppatori si stanno focalizzando maggiormente sul supporto alle versione 2.6[3] del kernel; proprio per la ragione sopra citata si sta cercando però, quando possibile, di applicare le modifiche critiche anche alla versione prevista per la serie 2.4, volendo oltretutto mantenere funzionante una delle caratteristiche più singolari del progetto: la possibilità di migrare processi tra queste due versioni del kernel Linux.

Migrazione dei processi.

CryoPID permette di migrare i processi "congelati" tra macchine potenzialmente molto diverse. L'immagine può essere semplicemente spedita (e.g., come allegato ad una email) alla macchina destinataria e qui utilizzata per riesumare il processo obiettivo.

Attualmente il supporto è ancora instabile e troppo "acerbo" per essere presentato come punto di forza del progetto.

[3] La versione 2.6.x del kernel Linux è quella attualmente in costante sviluppo, nonché la più utilizzata nelle maggiori distribuzioni GNU/Linux.

2.3 Il design in generale

Il progetto è diviso in due macro aree funzionali, ricalcando le principali operazioni offerte da ogni CRS:

1. **"freezer"**

   L'eseguibile che realizza tale operazione è chiamato "cryopid"[4]. È la parte designata alla sospensione del processo obiettivo e al salvataggio della sua immagine in un file, la quale viene anche compressa tramite strumenti quali "gzip", "lzo", etc., per mantenerne le dimensioni moderate.

   Il codice che esegue il "congelamento" risulta il più complesso e interessante, nonché la parte più consistente.

   L'applicativo *cryopid* si aggancia al processo obiettivo, il cui identificatore (PID - Process IDentifier) deve essere passato da riga di comando, diventandone il tracciante[5]. Il processo tracciato viene momentaneamente fermato, ne vengono recuperate le informazioni necessarie al suo ripristino, le quali sono poi salvate nel file di *checkpoint* il cui nome deve essere fornito dall'utente.

   A questo punto, a seconda di come si è invocato *cryopid*, il processo obiettivo viene fatto continuare[6] oppure fatto terminare[7].

   Si legga la sezione 2.4, per la descrizione dettagliata dei passaggi riguardanti la fase di *checkpoint* in CryoPID.

2. **"resumer"**

   Il ripristino del processo precedentemente "congelato" avviene in modo particolare; diversamente da molti CRS che prevedono un eseguibile che implementi il "resumer", in CryoPID quest'ultimo non esiste. È lo stesso file di *checkpoint* che contiene al suo interno il meccanismo utile al ripristino!

[4] Il lettore, lo confessi, non se lo aspettava, vero?

[5] Comportamento ottenibile tramite la system call `ptrace()`

[6] `PTRACE_DETACH` prima di far ripartire il processo tracciato, lo disaccoppia da quello tracciante.

[7] `PTRACE_KILL` invia un segnale di `SIGKILL` al processo tracciato, causandone la terminazione forzata.

Si legga la sezione 2.5 per i dettagli riguardanti il formato e il contenuto del file di *checkpoint*.

Il codice di *restart* interno al file immagine legge, di volta in volta, le informazioni ivi salvate cercando di ripristinarle e di ricostruire il processo originale. Si legga a tal proposito la sezione 2.6.

Le descrizioni tecniche che seguono si riferiscono, nei particolari, all'implementazione del progetto per l'architettura x86 e sono soggette a minime differenze rispetto alle altre.

2.4 La fase di *checkpoint* in CryoPID

L'operazione di *checkpoint* è fondamentale in ogni CRS, e senza una sua buona implementazione non vi possono essere speranze di un corretto ripristino dell'applicazione o processo obiettivo.

L'eseguibile *cryopid* che realizza tale fase è l'unico prodotto dal processo di compilazione del progetto in esame. È una applicazione utente che fa utilizzo delle diverse infrastrutture implementate nel kernel Linux per esportare ed accedere in "user space" informazioni altrimenti inaccessibili.

2.4.1 Gli "strumenti del mestiere"

Di seguito vengono descritti gli strumenti più utilizzati in questa fase, ovvero la system call `ptrace()` e il famoso file system virtuale `/proc`.

`ptrace()`.

La dichiarazione di `ptrace()` si trova nel file `sys/ptrace.h`, mentre il suo prototipo è il seguente:

*long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);*

Come riportato nel manuale per sviluppatori, `ptrace()` è una system call pensata per il tracciamento di processi "figli" da parte di un processo "padre".

In generale, `ptrace()` permette di intercettare le system call e di ispezionare (con la possibilità di modificare) l'ambiente d'esecuzione del processo tracciato. Infatti, il principale campo di utilizzo di tale direttiva è il debugging delle applicazioni.

Due sono le metodologie utilizzabili per iniziare il tracciamento di uno o più processi "figli" da parte di un processo "padre". Un primo modo prevede la chiamata di `ptrace()` col valore `PTRACE_TRACEME` nel parametro *request*. In questo caso è il processo designato in qualità di "tracciato" a richiedere il servizio di tracciamento. Il secondo modo, al contrario, prevede che sia il processo designato come "tracciante" a richiedere il servizio, diventando il padre (anche se solo virtuale) del processo obiettivo. In questo secondo caso, la system call `ptrace()` viene chiamata con `PTRACE_ATTACH` come valore per il parametro *request*. A seguito di una di queste due chiamate il processo tracciante è in grado di recuperare dati dalla memoria del tracciato tramite chiamate `PTRACE_PEEKTEXT` o `PTRACE_PEEKDATA` (attualmente, in Linux non vi sono spazi di indirizzamento separati per testo e dati, perciò queste due richieste sono identiche) passando a `ptrace()` un indirizzo specifico nel parametro *addr*. Non solo, tramite chiamate `PTRACE_POKETEXT` (o `PTRACE_POKEDATA`, vale lo stesso discorso fatto prima) il processo tracciante ha la possibilità di inserire *data* all'indirizzo *addr* nella memoria del tracciato potendo, potenzialmente, modificarne il comportamento.

Un ulteriore esempio della versatilità e potenza della system call `ptrace()` sono le chiamate `PTRACE_GETREGS` e `PTRACE_SETREGS,` che consentono di recuperare e di impostare, rispettivamente, i registri general purpose del processo tracciato.

Il progetto CryoPID fa un uso intensivo della direttiva `ptrace()` sfruttandola non tanto per tracciare le system call del processo da "congelare" quanto per accedere ad importanti informazioni utili al suo ripristino, come appunto il valore dei registri della CPU in un determinato momento.

procfs.[8]

Questo è uno pseudo-file system, generato dinamicamente dal kernel, che non fa riferimento ad alcun dispositivo fisico ma presenta in forma gerarchica di directory e file alcune delle strutture interne del kernel stesso.

Tipicamente, questo file system virtuale è montato presso la directory `/proc`.

L'implementazione presente in Linux ricorda molto quella realizzata nel sistema operativo "Plan 9"[9]; sotto `/proc` sono presenti tante directory quanti sono i processi in esecuzione, inclusi quelli kernel. In particolare, per ognuno è prevista una directory `/proc/`*`pid`* che contiene numerose informazioni riguardanti le risorse assegnate al processo stesso; ne sono esempi i file relativi alle regioni di memoria virtuale mappate (`/proc/`*`pid`*`/maps`) e i file descriptor (`/proc/`*`pid`*`/fd`). Quello seguente è un frammento di esempio del contenuto tipico del file "maps" associato ad un processo.

```
08048000-08053000 r-xp 00000000 08:03 2451540 /bin/cat
08053000-08054000 rw-p 0000b000 08:03 2451540 /bin/cat
08054000-08075000 rw-p 08054000 00:00 0 [heap]
b7b2f000-b7c18000 r--p 0019c000 08:03 991247 /usr/lib/locale/locale-archive
b7c18000-b7e18000 r--p 00000000 08:03 991247 /usr/lib/locale/locale-archive
b7e18000-b7e19000 rw-p b7e18000 00:00 0
```

Figura 2.1: Output del comando "cat /proc/self/maps"

Inoltre, procfs fornisce un modo per ottenere ed impostare informazioni legate al sistema stesso. Infatti, viene spesso utilizzato come alternativa, se non addirittura come metodologia consigliata, all'utilizzo di particolari system call; è questo, ad esempio, il caso della `sysctl()`, direttiva che permette la lettura e l'impostazione dei parametri del sistema.

Il fatto interessante e particolare è la possibilità di manipolare, trattandosi di directory e file, le strutture dati interne al kernel tramite semplici operazioni di input/output.

Il kernel si occupa di generare al volo il contenuto ed i nomi dei file all'interno

[8]Si legga anche la sezione 5 del `man` riguardante `/proc`

[9]http://plan9.bell-labs.com/plan9/index.html

di procfs, e questo ha il grande vantaggio di rendere accessibili i vari parametri a qualunque comando di shell e non solo.

2.4.2 Operazioni preliminari

Come già anticipato, per accedere alle informazioni riguardanti il processo obiettivo, *cryopid* utilizza le possibilità offerte dalla system call `ptrace()` diventando, come operazione preliminare, il padre "virtuale" del processo obiettivo tramite la `PTRACE_ATTACH`. Effetto collaterale di quest'ultima chiamata è l'invio del segnale di `SIGSTOP` al processo da tracciare, causandone la sospensione temporanea. Questo comportamento risulta necessario affinché il processo tracciante sia in grado di "ispezionare" il tracciato per poi magari riavviarlo.

In seguito all'aggancio, *cryopid* procede al salvataggio dei registri della CPU del processo da congelare tramite la `PTRACE_GETREGS`. Questo ne garantisce il normale riavvio una volta terminata l'intera operazione di *checkpoint*; in generale, è necessario che il processo obiettivo sia riavviato senza che il suo "ambiente" risulti minimamente cambiato, altrimenti potrebbero generarsi situazioni di errore. Per intendersi, è similare a quello che usualmente capita ad ogni context switch nel kernel.

2.4.3 Le informazioni da "congelare"

È giunto il momento di addentrarsi nei dettagli riguardanti quali informazioni vengono salvate e come.

Di seguito vengono descritte, nell'ordine di comparsa nel codice sorgente, le tipologie di risorse gestite attualmente da *cryopid*.

1. **TLS (Thread Local Storage).**

   La TLS è una risorsa molto importante nella corretta gestione di un processo di *checkpoint/restart* (cfr. 1.3).

   Il codice di *cryopid* prevede la sua memorizzazione mediante l'utilizzo della `PTRACE_GET_THREAD_AREA` che consente di reperire un elemento all'interno della TLS passandone l'indirizzo tramite il parametro *addr*.

2. **Spazio degli indirizzi (VMA).**

   La memorizzazione delle zone di memoria assegnate al processo (e librerie associate) è certamente cruciale. Come detto in precedenza, CryoPID è un progetto a livello utente e perciò, soprattutto in questo frangente, fa un utilizzo massiccio delle informazioni kernel che vengono esposte a livello user tramite il file system virtuale `/proc`. In particolare, viene preso in considerazione il file `/proc/`*`pid`*`/maps` che riporta, riga per riga, informazioni descrittive delle zone di memoria virtuale assegnate al processo obiettivo, il cui PID deve essere fornito dall'utente al momento del lancio di *cryopid*.

   Il codice designato alla memorizzazione dello stato delle VMA analizza uno dopo l'altro i campi costituenti le linee di testo riportate nel file "maps", in modo da poterne salvare il valore in strutture dati del tipo riportato in figura 2.2.

```
struct cp_vma {
        unsigned long start, length;    /* vma start and its length */
        int prot;       /* access rights */
        int flags;      /* special flags: PRIVATE or SHARED vma */
        int dev;        /* device associate */
        long pg_off;    /* offset into the file/whatever */
        int inode;      /* i-node of that device */
        char *filename; /* filename of the file */
        char have_data; /* determinate if this struct stores real data */
        char is_heap;
        unsigned int checksum;  /* checksum of the real data */
        void* data; /* real data, if stored */
};
```

Figura 2.2: Struttura dati "cp_vma" (cpimage.h)

   Il contenuto vero e proprio della VMA non viene sempre salvato, come invece si penserebbe. Debbono essere salvate, sicuramente, le informazioni contenute in quelle zone di memoria definite "private" i cui permessi di accesso prevedono anche l'opzione di scrittura. Queste zone di memoria sono, tipicamente, utilizzate per la memorizzazione di dati, quali variabili globali, variabili locali, memoria allocata tramite `malloc()` (e similari) per finire con i cosiddetti "mmapped file" (file mappati in memoria RAM).

In generale, il fine di CryoPID è la memorizzazione dello stretto necessario; non viene quindi considerato il contenuto informativo di quelle zone di memoria riconducibili a codice eseguibile (e.g., le librerie dinamiche) perché ricaricabili, a meno di particolari situazioni o scenari, dalla memoria secondaria durante la fase di *restart* del processo "congelato".

Il comportamento appena descritto si rivela completamente sbagliato quando si vuole migrare l'immagine di quest'ultimo fra macchine potenzialmente diverse; si pensi, ad esempio, alle incongruenze di versione della *glibc* usate sulla macchina mittente e quella destinataria o addirittura a quei frequenti casi di una o più librerie mancanti. L'ovvio risultato sarebbe un eloquente "Segmentation Fault" in fase di *restart*. Per ovviare a tali problemi, la soluzione è richiedere a *cryopid*, in fase di lancio, di memorizzare tutto il contenuto dello spazio degli indirizzi, comprese quindi le VMA assegnate al codice delle librerie e non solo[10].

La memorizzazione dei dati contenuti nelle zone di memoria dedicate al processo tracciato avviene, ancora una volta, mediante l'utilizzo di procfs; in particolare viene sfruttato il file `/proc/`*`pid`*`/mem` per accedere alle VMA tramite le usuali system call `open()`, `read()` e `lseek()`.

3. **File e file descriptor.**

   CryoPID prevede il supporto, seppur molto limitato, di *checkpoint/restart* per i file e i file descriptor. La struttura dati pensata a tale scopo è visibile in figura 2.3.

   Anche in questo caso, *cryopid* fa un massiccio utilizzo delle agevolazioni fornite dal file system virtuale `/proc`. Infatti, la sottocartella `/proc/`*`pid`*`/fd` contiene un elemento per ogni file che il processo in esame ha aperto, di nome pari al numero del file descriptor associato, il quale è un link simbolico al vero file.

   Di ognuno viene salvato il numero del file descriptor e di quest'ultimo, tramite `lstat()` e `fcntl()`, se ne ricavano le informazioni riguardanti la

[10]Comportamento ottenibile specificando l'opzione `-l` (oppure, `--libraries`).

```
struct cp_fd {
        int fd;         /* file descriptor number */
        int mode;       /* access rights */
        int close_on_exec;      /* file descriptor flags */
        int fcntl_status;       /* file status flags */
        off_t offset;
        int type;
        union {
                struct cp_console console;
                struct cp_file file;
                struct cp_fifo fifo;
                struct cp_socket socket;
        };
};
```

Figura 2.3: Struttura dati “cp_fd” (cpimage.h)

protezione e i file descriptor flag. Segue poi la memorizzazione dei file status flag e del file offset per poi riconoscere, mediante `stat()`, la tipologia del file in questione. Come osservabile facilmente dalla struttura dati sopra riportata, i file gestiti da *cryopid* sono i device a caratteri (in realtà, soltanto quello associato al processo obiettivo), i socket, i file regolari ed infine le FIFO/pipe.

- **Device a caratteri (console).** Viene memorizzata la struttura dati “termios” osservabile nel file `<termios.h>`, soltanto quella relativa al terminale associato del processo obiettivo. Il valore del device è ottenuto dal file `/proc/`*`pid`*`/stat`. La struttura mantiene le diverse impostazioni configurabili nell’interfacciamento con un terminale, dai metodi di input/output ai caratteri di controllo.
- **Socket.** Attualmente il supporto è limitato ai socket TCP e UNIX. I primi presentano un’ulteriore restrizione; il kernel deve essere equipaggiato (mediante patch) del progetto “tcpcp”[11] (TCP Connection Passing). Quest’ultimo progetto permette ad applicazioni cooperanti di scambiarsi il controllo delle connessioni TCP instaurate tra un host Linux ed un altro. Attualmente però, anche se non ufficialmente, non

[11] http://tcpcp.sourceforge.net/

è più in sviluppo (l'ultima release risale al 2005, il kernel Linux è *leggermente* cambiato da allora).
Anche nel caso dei socket, `/proc` si rivela uno strumento fondamentale. In particolare, la directory `/proc/net` presenta un insieme di file che riportano numerose informazioni di stato riguardanti parti del livello di networking del kernel.

- **File regolari.** Il supporto ai file regolari è stato aggiunto in CryoPID soltanto nelle ultime versioni.
  Attualmente, per ogni file regolare aperto dal processo obiettivo, vengono salvati il nome e la dimensione. Il contenuto vero e proprio viene memorizzato soltanto nel caso in cui il file risulti cancellato (in generale, da soggetti esterni); infatti, è probabile che il processo obiettivo, una volta ripristinato, cerchi nuovamente di interagire con lo stesso.
  È possibile ottenere quest'ultima, preziosa informazione, ispezionando il contenuto del link simbolico, presente nella directory `/proc/`*`pid`*`/fd`, che referenzia il file in questione. Tipicamente, quest'ultimo contiene il percorso assoluto al vero file aperto dal processo; nel caso in cui questo venga cancellato, allora al percorso assoluto viene automaticamente aggiunta in coda la stringa "(deleted)", permettendone così il rilevamento.

- **FIFO/pipe.** Questi sono metodi di IPC (Inter-Process Communication) molto utilizzati in ambienti UNIX-like per la loro facilità d'utilizzo.
  Allo stato attuale, il progetto supporta solo le pipe e soltanto quelle che collegano lo stesso processo (i.e., pipe "a se stessi") dato che CryoPID non riesce ad effettuare il *checkpoint/restart* di gerarchie di processi.
  L'approccio alla memorizzazione è abbastanza banale; viene creata una tabella hash indicizzata sul valore dell'i-node delle pipe nella quale vengono salvati i PID dei processi collegati e il valore dei due file descriptor utilizzati per implementarla. Ovviamente, in questo modo ogni pipe "a se stessi" è facilmente identificabile perché, agendo sullo stesso i-node, i due file descriptor che la implementano (lettura e

scrittura) ricadono nella stessa locazione della tabella hash.

4. **Segnali.**

   Come ogni altro CRS, anche CryoPID gestisce il *checkpoint/restart* dei segnali. In realtà, *cryopid* memorizza soltanto i gestori dei primi trenta segnali potenzialmente utilizzabili, senza salvare l'insieme dei segnali pendenti (i.e., i segnali che sono stati inviati al processo, ma bloccati).

   Per reperire il gestore di ogni segnale gestito dal processo obiettivo, viene utilizzata la system call `sigaction()`; lo si deduce anche dalle strutture dati utilizzate in CryoPID per tale funzionalità (figura 2.4). La struttu-

```
struct cp_sighand {
        int sig_num;
        struct k_sigaction *ksa;
};

struct k_sigaction {
        __sighandler_t sa_hand;
        unsigned long sa_flags;
        void (*sa_restorer)(void);
        arch_sigset_t sa_mask;
};
```

Figura 2.4: Strutture dati per la gestione dei segnali (cpimage.h)

   ra dati "k_sigaction" è una versione semplificata della "sigaction", quella originale[12].

   La dichiarazione di `sigaction()` si trova nel file `signal.h`, mentre il suo prototipo è il seguente:

   *int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);*

   Come riportato nel manuale, la `sigaction()` è una system call utilizzata per modificare il comportamento da tenere, da parte di un processo, alla ricezione di uno specifico segnale. È possibile specificare una particolare azione da compiere per tutti i segnali che non siano `SIGKILL` e `SIGSTOP`.

[12] Si consulti la sezione 2 del `man` riguardante la syscall `sigaction()`

Se il parametro attuale *act* non è `NULL`, la nuova azione per il segnale *signum* viene installata. D'altro canto, se il parametro attuale *oldact* non è `NULL`, questo viene utilizzato per il salvataggio dell'azione installata.

CryoPID sfrutta proprio quest'ultima opportunità offerta da `sigaction()`, riuscendo in questo modo a salvare, per ogni segnale supportato, il gestore installato al momento della fase di *checkpoint*, e non solo. Si riesce, infatti, anche a reperire le informazioni riguardanti la maschera dei segnali bloccati (durante l'esecuzione del gestore) e l'insieme dei flag utilizzati per modificare il comportamento del segnale in questione.

5. **Registri floating-point.**

   Il valore dei registri a virgola mobile è memorizzato mediante l'utilizzo della `PTRACE_GETFPREGS` nella struttura dati "user_fpregs_struct". Quest'ultima è definita nel file `user.h` e rappresenta l'interfaccia a livello utente alla gestione kernel dei registri floating-point. È stata pensata per essere utilizzata soprattutto in ambito di debug.

6. **Registri general purpose e non solo.**

   Le ultime informazioni che vengono memorizzate da *cryopid* riguardano la struttura dati "user", definita nel file `user.h`. Questa struttura dati rappresenta la parte utente delle informazioni kernel riguardanti i processi; tra le tante informazioni riportate è presente anche quella riguardante il valore dei registri general purpose. In particolare, la struttura dati "user_regs_struct" è quella addetta a tale scopo. Queste informazioni vengono memorizzate da *cryopid* tramite l'utilizzo di una serie di chiamate a `PTRACE_PEEKUSER`.

   In seguito, viene stabilito se il processo obiettivo è stato fermato nel corso di una system call; in caso affermativo, *cryopid* fa in modo di restituirgli, come valore di ritorno, l'errore `-EINTR`.

I frammenti di informazioni che *cryopid* ricava dal processo obiettivo vengono disposti in una lista di strutture dati, la cui definizione è osservabile in figura 2.5. Questa lista di informazioni è facilmente identificabile con l'immagine del processo "congelato", tanto da essere nominata "proc_image".

```
struct cp_chunk {
        int type;
        union {
                struct cp_misc misc; /* not in use right now */
                struct cp_regs regs;
                struct cp_fd fd;
                struct cp_vma vma;
                struct cp_sighand sighand;
                #ifdef __i386__
                        struct cp_i387_data i387_data;
                        struct cp_tls tls;
                #endif
        };
};
```

Figura 2.5: Struttura dati principale (cpimage.h)

Da un lato, la struttura dati "cp_chunk" riesce a memorizzare informazioni di tipo eterogeneo, sfruttando le caratteristiche delle `union` nel linguaggio C; dall'altro permette di realizzare una lista di oggetti esternamente identici, facile da utilizzare e modellare secondo le esigenze.

Per ogni informazione considerata da *cryopid*, le tipiche operazioni che vengono eseguite sono la ricerca, l'allocazione di una cp_chunk, la memorizzazione all'interno di questa ed infine l'aggiunta nella lista globale proc_image.

2.4.4 Alcuni trucchi in *cryopid*

Il lettore più attento e tecnicamente informato (riguardo la programmazione in ambiente UNIX-like) avrà sicuramente percepito la presenza di alcune inesattezze[13] durante la lettura della precedente sezione. In particolare, avrà particolari dubbi sul come alcune informazioni possano essere estrapolate dal processo obiettivo da parte dell'eseguibile *cryopid*.

Le potenzialità offerte dalla system call `ptrace()` e dal file system virtuale `/proc` sono inconfutabili; entrambi gli "strumenti" sono stati descritti in precedenza senza nascondere nulla al lettore. Inoltre, mentre da un lato `ptrace()` è una direttiva di sistema con specifiche funzionalità sotto il rigido controllo del

[13] Inserite, volutamente, per permettere una spiegazione degli argomenti più lineare e scorrevole.

kernel, dall'altro i file contenuti in `/proc/pid`, se utilizzati in lettura, sono accessibili a qualunque processo (non solo a quello il cui identificatore è `pid`). Si è omessa, invece, la descrizione sull'utilizzo di determinate system call e metodologie.

In generale, molte risorse (e.g., file descriptor, segnali, etc.) non possono essere manipolate da nessun altro processo all'infuori di quello proprietario. Ad esempio, non è permesso ad un processo di accedere (e/o manipolare) i flag di un file descriptor a meno che non risulti anche il proprietario di tale risorsa. Lo stesso discorso vale per i segnali. Tramite la `sigaction()` un processo può modificare soltanto il suo comportamento nei confronti di specifici segnali; non può farlo a nome di altri processi, nemmeno se ne è diventato il padre (virtuale) mediante `ptrace()`.

Le motivazioni attribuibili a tale comportamento sono diverse, molte delle quali da ricercare nel ruolo fondamentale del kernel di un sistema operativo; si pensi, ad esempio, a motivazioni inerenti la sicurezza.

Il progetto CryoPID scavalca questo ostacolo in modo particolare: forza il processo obiettivo ad eseguire delle system call che non aveva previsto, che neanche compaiono nel suo codice sorgente!

L'eseguibile *cryopid* ricerca, nello spazio degli indirizzi assegnato al processo tracciato, una istruzione utilizzata per causare un interrupt software, mezzo tipicamente usato per implementare system call. Una volta trovata, ne salva l'indirizzo di memoria, pronto ad utilizzarla secondo esigenza.

Tipicamente, istruzioni di questo genere si trovano all'interno di codice di libreria; esempio eloquente è rappresentato dalla *libc*, che per implementare i suoi servizi ne fa un uso massiccio.

Esempio di interrupt software, per architetture x86, è l'istruzione assembly `int0x80`. La funzione implementata in CryoPID per il riconoscimento del codice macchina di tale istruzione è quella in figura 2.6.

Per forzare il processo obiettivo ad eseguire le system call necessarie, *cryopid* implementa un meccanismo similare ad un normale context switch, sfruttando le funzionalità offerte da `ptrace()`.

Vengono memorizzati i valori dei registri del processo obiettivo per poi inserirvene di nuovi, necessari all'esecuzione della specifica syscall con i dovuti

```
is_a_syscall(unsigned long inst, int canonical)
{
        if ((inst&0xffff) == 0x80cd)
                return 1;
        return 0;
}
```

Figura 2.6: Codice di riconoscimento dell'istruzione *int0x80*

parametri (e.g., nelle architetture x86 il registro "eax" deve contenere il numero della syscall da eseguire, mentre "ebx", "ecx" ed "edx" vengono utilizzati per i parametri). È questo il momento nel quale viene impostato il valore del registro "eip" (i.e., extended instruction pointer) all'indirizzo dell'interrupt software precedentemente salvato[14].

Dopodiché, tramite la `PTRACE_SINGLESTEP`, viene fatto eseguire al processo obiettivo un passo di computazione, ottenendo l'esecuzione della system call voluta. Infine, non resta altro che memorizzare i risultati della computazione e ripristinare lo stato originale della CPU del processo tracciato.

Rimane da chiarire come sia possibile il passaggio di alcune informazioni dal processo obiettivo a *cryopid*, e viceversa, in particolari situazioni. Esistono, infatti, delle system call che necessitano, come parametro attuale, di un puntatore ad una zona di memoria dalla quale leggere dati utili alla computazione o nella quale restituire il risultato della stessa.

È il caso, ad esempio, della direttiva `ioctl()`, utilizzata per il *checkpoint* dello stato della console. Questa necessita, come terzo parametro, di un puntatore ad una zona di memoria dove scrivere il risultato della sua computazione.

Anche avendo a disposizione la possibilità di fare eseguire tale syscall al processo obiettivo, rimane il problema di specificare il valore del puntatore. Va immediatamente precisato che non risulterebbe corretto fornire, semplicemente, indirizzi a strutture dati appartenenti a *cryopid*; si ricorda che si sta lavorando in un ambiente a memoria protetta. Non è possibile per un processo (in questo ca-

[14] Il lettore avrà sicuramente intuito l'utilizzo, rispettivamente, di `PTRACE_GETREGS` e `PTRACE_SETREGS`.

so, quello obiettivo) scrivere, indisturbato, nello spazio degli indirizzi di un altro (*cryopid*).

Il problema viene affrontato in CryoPID in una maniera non troppo elegante, ma pratica e funzionante.

Durante il *checkpoint* dello spazio degli indirizzi del processo obiettivo, *cryopid* verifica la presenza di una "scribble_zone"; una pagina di memoria anonima, privata e con permessi di lettura e scrittura. Di questa, ne viene fatta una copia di riserva, in modo da poterne ripristinare lo stato subito prima di disporre, eventualmente, la continuazione dell'esecuzione al processo tracciato (cfr. 2.4.5).

Quando la system call da fare eseguire a quest'ultimo necessita di un puntatore a dati, questi vengono prima inseriti da *cryopid* (tramite `PTRACE_POKETEXT`) nella scribble_zone, e in seguito se ne fornisce il puntatore.

Un simile meccanismo viene utilizzato quando la system call necessita, invece, di un puntatore ad una zona di memoria dove riversare i dati della sua computazione. L'eseguibile *cryopid* indica un indirizzo all'interno della scribble_zone e, terminata l'esecuzione della syscall, ne recupera il contenuto tramite `PTRACE_PEEKTEXT`.

Nelle ultime versioni del progetto, quest'ultima implementazione è stata soppiantata, per motivi prestazionali, dall'utilizzo del file "mem" contenuto in `/proc`. Come riportato nel manuale, `/proc/`*`pid`*`/mem` può essere utilizzato per accedere alle pagine di memoria di un qualsiasi processo; chiaramente, in sola lettura. Tramite un `lseek()` ed una successiva `read()` si riesce ad eseguire lo stesso lavoro che, nelle prime versioni, veniva svolto mediante un ciclo di chiamate a `PTRACE_PEEKTEXT`; il risultato è un notevole guadagno computazionale[15].

2.4.5 Operazioni finali

Compiuto il *checkpoint* di tutte le risorse, *cryopid* esegue due azioni fondamentali, nei confronti del processo obiettivo, prima di mettere fine alla sua azione di tracciamento.

La prima consiste nel ripristino dello stato dei registri della CPU, salvato du-

[15]Si ricordi, infatti, che `ptrace()` è una system call e quindi una sua invocazione risulta relativamente più lenta rispetto ad una normale chiamata a funzione (implica dei *mode switch*).

rante le operazioni preliminari (cfr. 2.4.2), mentre la seconda prevede la ricostruzione del contenuto originale della scribble_zone. Senza queste operazioni, una potenziale ripresa dell'esecuzione del processo tracciato si concluderebbe, in breve tempo, in errore.

L'azione di tracciamento di *cryopid* può terminare in due modi, a seconda delle opzioni specificate al suo lancio da riga di comando.

Il comportamento standard prevede la chiamata a `PTRACE_DETACH`, la quale determina il proseguimento dell'esecuzione del processo obiettivo. Al contrario, se si è specificata l'opzione `-k` (oppure, `--kill`), il processo tracciato, ormai "congelato", viene fatto terminare mediante la chiamata a `PTRACE_KILL`.

2.5 Il layout del file immagine

Questa vuole essere una sezione di raccordo tra le descrizioni delle fasi di *checkpoint* e di *restart*. D'altronde, anche lo stesso file immagine lo è, dato che rappresenta l'output della prima e l'input necessario alla seconda.

La precedente descrizione della fase di "congelamento" del processo obiettivo è terminata senza menzionare il criterio di scrittura effettiva dell'immagine (cfr. 2.5.2). Per comprendere i dettagli di tale procedimento è necessaria una introduzione al layout, generico, della stessa.

Analogo discorso vale per la fase riguardante il ripristino del processo; la sua esamina non può essere affrontata senza prima avere compreso l'intero contenuto del file immagine.

Ogni file immagine, subito dopo essere stato caricato in memoria RAM, si presenta come in figura 2.7, dove sono state evidenziate le quattro aree, sempre presenti, particolarmente importanti nel funzionamento del progetto CryoPID.

In generale, l'immagine è un file ELF, acronimo per Executable and Linking Format. È il formato dei file eseguibili (binari), attualmente il più diffuso in ambiente Linux. In breve, definisce come il file è composto e organizzato; indica dove trovare il testo del programma, i dati inizializzati e non, quali librerie considerare, e molto altro. Questo permette al kernel e al binary loader di decifrare il file, caricarlo in memoria primaria ed eseguirlo.

Queste ultime annotazioni sul formato ELF danno la conferma di quanto si

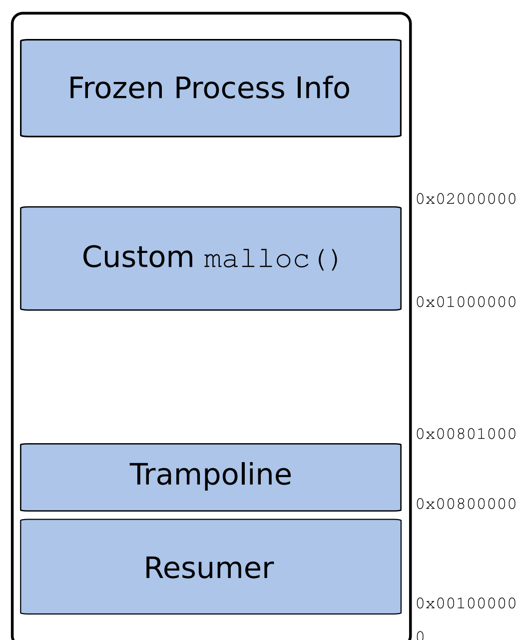

Figura 2.7: Layout del file immagine in memoria principale

affermava qualche sezione indietro; l'immagine è un file eseguibile contenente tutto il necessario per un corretto ripristino del processo originario. CryoPID non prevede nessun tipo di programma resumer autonomo.

2.5.1 Il file *stub*

Quanto asserito nelle ultime frasi è indiscutibile. Difatti, l'unico eseguibile prodotto dal processo di compilazione del progetto è il file *cryopid*. Il lettore ammetterà di essere sorpreso, e magari un po' dubbioso, di come sia allora possibile il *restart* dei processi "congelati"[16].

È giunto quindi il momento di svelare, nei dettagli, il meccanismo implementato nel progetto CryoPID.

Dopo essersi procurati il codice sorgente dal repository ufficiale[17] è possibile indagare sul numero dei file contenenti la funzione "`main`". Ebbene, sono tre:

[16] Anche il sottoscritto lo era agli inizi di questo lavoro di tesi.

[17] Mediante il comando da shell: `$ hg clone https://sharesource.org/hg/cryopid`

"`freeze.c`", "`stub_common.c`" e "`fork2_helper.c`" (crf. figura 2.8). L'ultimo non è interessante al momento.

```
~/cryopid$ grep "^int main" src/ -R
src/freeze.c:int main(int argc, char** argv)
src/stub_common.c:int main(int argc, char**argv, char **envp)
src/fork2_helper.c:int main(int argc, char **argv)
```

Figura 2.8: Funzioni `main()` in CryoPID

Tipicamente, la funzione "`main`", in un sorgente scritto in linguaggio C, è la prima ad essere eseguita all'avvio del programma; ne è l'entry point e quindi è sempre presente, perfino nel codice di un kernel. Proprio per quanto detto, ad un file contenente tale funzione (una volta compilato e linkato) corrisponde, solitamente, un eseguibile. Il progetto CryoPID fa eccezione, almeno in parte.

Dal primo file ("`freeze.c`") insieme con altri di tipo sorgente ed header, è generato l'eseguibile *cryopid*. Da "`stub_common.c`" ed altri, invece, viene prodotta una "stub", simile ad un file ELF ma non eseguibile direttamente. È un file oggetto che viene integrato in *cryopid* e da questi impiegato come base strutturale di tutte le immagini di processi "congelati".

In particolare, "`stub_common.c`" contiene il codice sorgente che implementa il resumer[18]; nella stub derivatavi vengono scritte, al termine della fase di *checkpoint*, le informazioni riguardanti il processo obiettivo[19].

In definitiva, non è presente un programma ad hoc il cui compito sia quello di ripristinare i processi congelati, bensì è come se ce ne fossero tanti quante sono le immagini dei processi "congelati" perché incluso in ognuno di esse.

Come precedentemente scritto nella sezione "Caratteristiche progettuali", CryoPID offre la possibilità di far migrare i processi tra macchine, potenzialmente, molto diverse; questo obiettivo pone seri problemi se, ad esempio, la fase di *restart* necessita di particolari librerie per compiere il proprio lavoro. Per ovviare a tali complicazioni, gli sviluppatori hanno optato per un processo di creazione statico della stub; ovvero, il codice sorgente delle funzioni di libreria utilizzate viene integrato (copiato) all'interno di quest'ultima.

[18] Si noti l'associazione con la prima regione (partendo dal basso) evidenziata in figura 2.7.

[19] Ci si sta riferendo all'ultima regione (partendo dal basso) rappresentata in figura 2.7.

Se da un lato questo metodo sopperisce ad eventuali mancanze di librerie installate, dall'altro causa l'aumento sistematico della dimensione dei file immagine. In particolare, la *glibc* è famosa per essere pesante e corposa, quindi poco compatibile con le caratteristiche richieste da CryoPID. Per mantenere i file immagine di dimensioni limitate, la stub è linkata utilizzando la "dietlibc"[20]; una libreria standard per il C conforme alle regole POSIX e che, come annuncia il nome, è stata pensata per essere leggera e veloce. Va precisato che *cryopid* è, invece, un eseguibile dinamicamente linkato che sfrutta le funzionalità offerte dalla GNU *glibc*.

2.5.2 La scrittura delle informazioni "congelate"

Con quanto scritto di seguito si chiude la descrizione riguardante la fase di *checkpoint* implementata nel progetto CryoPID.

Al termine del lavoro di tracciamento compiuto da *cryopid*, la lista proc_image contiene tutte le informazioni necessarie alla ricostruzione del processo "congelato"; non rimane altro che integrarle all'interno della stub.

Per cercare di ridurre al minimo la dimensione di tali informazioni, che gravano sulle totali del file immagine finale, si utilizza un meccanismo di compressione. Ve ne sono diversi supportati in CryoPID, quali "gzip" e "lzo". La scelta di quale tecnica utilizzare è lasciata all'utente e, attualmente, deve essere effettuata al tempo di compilazione del progetto (di default viene utilizzato gzip).

L'idea è quella di scansionare l'intera lista proc_image facendo attraversare, di volta in volta, la cp_chunk referenziata attraverso il processo di compressione; in seguito, l'informazione compressa viene scritta all'interno del file immagine. Più in dettaglio, per ogni cp_chunk da gestire, viene scritto un intero `(CP_CHUNK_MAGIC)`, il tipo di informazione contenuta nella struttura dati ed infine, in base a questo, i dati stessi. Tipicamente, `CP_CHUNK_MAGIC` viene inserito come "divisorio", fra una informazione e la seguente, per far comprendere alla fase di *restart* come gestire le diverse informazioni memorizzate.

I dati vengono scritti in coda al file stub, ad un offset preciso. In particolare, i dettagli ELF interni al file in esame vengono specificati mediante l'utilizzo di

[20] http://www.fefe.de/dietlibc/

un linker script personalizzato (i.e., `arch-i386/stub-linking.x`) passato al linker durante l'operazione di compilazione/linking del progetto. Tipicamente, il principale obiettivo di un linker script è quello di descrivere come e quali sezioni di un file ELF debbano essere gestite e caricate in memoria primaria per l'esecuzione.

Lo script distribuito nel progetto indica, tra le altre, una sezione chiamata "cryopid.image" (definita `NOLOAD`, da non caricare in memoria primaria perché non necessaria all'esecuzione) interna alla stub, oltre la quale dover scrivere le informazioni di stato del processo obiettivo. La sezione è posta alla fine del file, definendone indirettamente un offset. Da ciò si deduce il motivo della scrittura in coda dei dati.

Al termine dell'operazione finora esposta, viene scritta una ulteriore informazione (denominata "final chunk") necessaria ad evidenziare la fine delle informazioni riguardanti il processo "congelato" in modo tale da facilitare il lavoro del resumer.

2.5.3 Una `malloc()` alternativa

Come il lettore avrà notato, la figura 2.7 mostra una particolare sezione definita "Custom `malloc()`", necessaria alla fase di *restart* del processo "congelato".

Ovviamente, la dietlibc utilizzata (soltanto) dal resumer fornisce la funzione di libreria `malloc()`, una delle più utilizzate in qualunque progetto. Questa, insieme alle sue varianti, è utilizzata per allocare memoria al processo richiedente. Tipicamente, la memoria richiesta viene allocata sullo heap; questo soltanto se la dimensione necessitata risulta minore di `MMAP_THRESHOLD` byte (di default 128 KB e configurabile mediante `mallopt()`). Nei casi in cui la richiesta superi i limiti imposti viene mappata una zona di memoria anonima, solitamente fruibili attraverso la system call `mmap()`. Per comprendere a pieno il discorso finora esposto, si consiglia di osservare la figura 2.9.

Allo stato attuale, soltanto la versione per architetture x86 di CryoPID permette un corretto ripristino dello stato dello heap del processo "congelato"; ciò significa che potrebbero sorgere dei problemi utilizzando la funzione `malloc()` fornita direttamente dalla libreria. In particolare, potrebbe accadere che quest'ultima al-

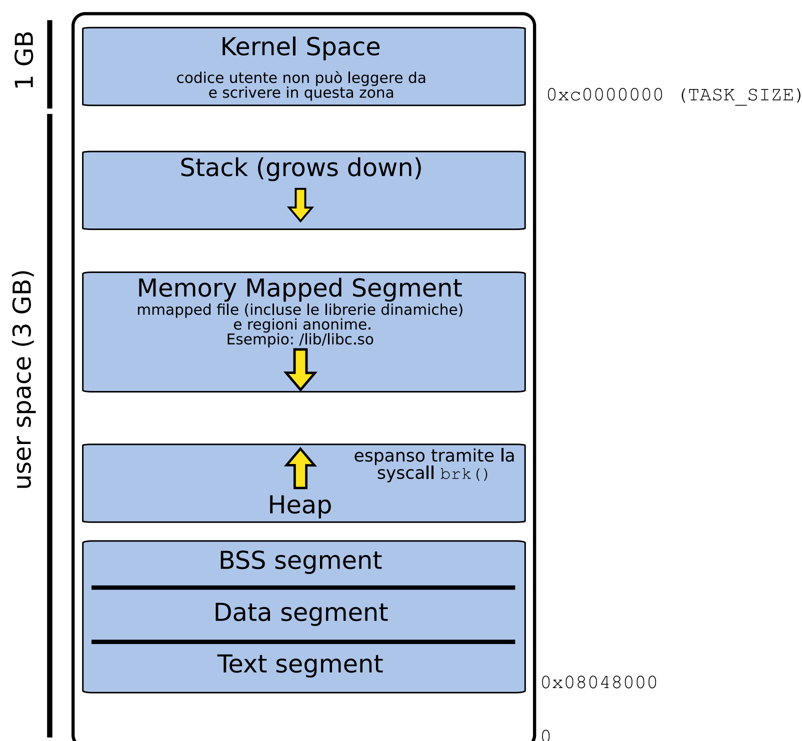

Figura 2.9: Layout di un processo in Linux (architettura x86)

lochi memoria nello spazio degli indirizzi destinato alle informazioni "congelate" causando, in definitiva, errori irreversibili.

Proprio per questo motivo, gli sviluppatori hanno deciso di implementare una funzione specifica per il progetto, `xmalloc()`, da impiegare al posto di quella fornita dalla libreria standard. CryoPID si serve della `xmalloc()` ogni qual volta necessiti di memoria. In realtà, per la fase di *checkpoint*, questa usufruisce a sua volta della `malloc()` fornita dalla *glibc*[21]; per la fase di *restart*, invece, provvede una propria implementazione (oggetto delle modifiche presenti nella prima "patch" sviluppata dal sottoscritto).

L'idea è di sfruttare il meccanismo offerto dalla system call `mmap()` per allocare pagine di memoria anonime ad indirizzi "bassi"[22], sicuramente non utilizzati dalle informazioni da ripristinare. La zona di memoria scelta per tale scopo è

[21] Il problema sopra riportato va preso in considerazione soltanto durante il processo di *restart*.

[22] Ci si riferisce a quelli inferiori a `0x08048000`, tipico indirizzo di inizio del segmento testo di un processo.

proprio quella compresa tra gli indirizzi `0x01000000` e `0x02000000`.

Ogni richiesta viene approssimata ad un multiplo di `PAGE_SIZE` (tipicamente, 4096 byte) e, di conseguenza, vengono mappate tante nuove VMA anonime quante necessarie. Se da un lato questo algoritmo (quello originale del progetto) risulta semplice, dall'altro "sperpera" eccessiva memoria. Il lettore esperto in materia avrà sicuramente capito la gravità dello spreco, ma volendo essere esaurienti vengono citate alcune stampe di debug che ne riportano un esempio lampante.

```
[old] using custom malloc. request in size: 5
[old] pointer value [dec]: 16777216, pointer value [x]: 1000000
[old] using custom malloc. request in size: 15
[old] pointer value [dec]: 16781312, pointer value [x]: 1001000
[old] using custom malloc. request in size: 4
[old] pointer value [dec]: 16785408, pointer value [x]: 1002000
```

Figura 2.10: Richieste (in byte) e rispettivi puntatori alle VMA allocate

Si noti come, anche per una richiesta di pochi byte, venga sempre allocata una intera pagina di memoria[23]. Altri problemi legati a tale implementazione sono il limite fisso alla dimensione, seppure ad oggi sufficiente, dell'area in esame e il suo indirizzo di allocazione.

Soprattutto a causa di quest'ultimo potrebbero verificarsi errori non prevedibili; infatti, non si può avere l'assoluta certezza che, ad esempio, nessun processo obiettivo (oppure libreria) utilizzi quella zona privatamente (anche se, di norma, gli indirizzi "bassi" non vengono utilizzati).

2.5.4 Il "trampolino"

L'ultima zona di memoria, denominata "Trampoline", che ricopre un ruolo fondamentale nel funzionamento complessivo di CryoPID (in particolare, della fase di *restart*), è posta sopra quella contenente il codice sorgente del resumer.

La finalità ultima di tale pagina di memoria è intuibile: vi vengono inserite le istruzione che, eseguite, determinano la ripresa dell'esecuzione del proces-

[23] La prima "patch" al progetto CryoPID risolve questo specifico problema risultando anche più efficiente dal punto di vista delle prestazioni (si legga la sezione 2.7 per i dettagli).

so da riattivare, proprio dal punto in cui era stato "congelato". Si spiega così il particolare nome "trampolino".

In particolare, durante il processo di *restart* viene richiesta la mappatura della pagina di memoria in questione tramite la syscall `mmap()`. Si cerca di allocare una pagina anonima accessibile in lettura, scrittura ed esecuzione; proprio all'indirizzo `0x00800000`.

In seguito, vi viene scritto il codice implementante le seguenti operazioni:

1. ripristino del contenuto del registro segmento *GS* (in generale, questo registro indirizza la Thread Local Storage del processo in esecuzione);
2. rimozione della memoria allocata mediante la `xmalloc()`;
3. rimozione della memoria contenente il codice sorgente del resumer;
4. ripristino del valore dei registri general purpose;
5. impostazione dei registri *CS* (i.e., Code Segment) ed eip (i.e., extended instruction pointer) al giusto indirizzo di memoria per la ripresa dell'esecuzione del processo "congelato";
6. "salto" (tramite proprio l'istruzione `jmp`, ma in codice macchina) all'esecuzione del codice del processo "congelato".

Il meccanismo, implementato in CryoPID, per ottenere quanto elencato è quello di scrivere tali operazioni direttamente in memoria, cioè in codice macchina (in esadecimale), mediante dereferenziazione di un puntatore all'indirizzo della pagina di memoria allocata in precedenza. Ciò significa che verranno, effettivamente, eseguite soltanto quando, attraverso un qualche meccanismo di modifica del flusso normale d'esecuzione, il registro eip del resumer verrà impostato all'indirizzo del trampolino (appunto, `0x00800000`).

Il motivo di tale implementazione va ricercato, perlopiù, nella natura delle operazioni da implementare. In particolare, molte istruzioni necessarie allo scopo sono accessibili soltanto nel linguaggio assembly (e.g., "saltare" ad un indirizzo di memoria specifico); inoltre, devono essere scritte in codice macchina per poterle inserire direttamente in memoria ed in seguito eseguirle.

A titolo di esempio, in figura 2.11 viene riportato il codice sorgente riguardante, rispettivamente, i precedenti punti 2, 5 e 6. Si noti come le operazioni non

```
/* munmap our custom malloc space */
*cp++=0xb8;*(long*)(cp) = __NR_munmap; cp+=4; /* mov foo, eax */
*cp++=0xbb;*(long*)(cp) = MALLOC_START; cp+=4; /* mov foo, ebx */
*cp++=0xb9;*(long*)(cp) = MALLOC_END-MALLOC_START; cp+=4; /* mov foo, ecx */
*cp++=0xcd;*cp++=0x80; /* int $0x80 */

[...]

/* jump back to where we were */
*cp++=0xea;
*(unsigned long*)(cp) = r->eip; cp+= 4;
/* ensure we use the right CS for the current kernel */
asm("mov %%cs,%w0":"=q"(r->xcs));
*(unsigned short*)(cp) = r->xcs; cp+= 2; /* jmp cs:foo */
```

Figura 2.11: Codice implementante le operazioni 2, 5 e 6

vengano eseguite, ma scritte in memoria dereferenziando la variabile puntatore *cp*. Verranno processate soltanto quando la computazione del resumer interesserà l'area "Trampoline".

Ovviamente, il codice che compone il trampolino è architettura dipendente; questo significa che CryoPID ne prevede uno per ogni architettura hardware supportata[24]. In generale, è importante implementare una interfaccia elegante che si frapponga tra il codice sorgente di base e quello dipendente dall'architettura in modo da facilitare il porting del progetto su hardware diverso.

2.6 La fase di *restart* in CryoPID

In ogni CRS, ad una fase di *checkpoint* corrisponde quella di *restart* che permette il ripristino dei processi a partire dalla loro immagine. Come descritto in precenza, in CryoPID non è presente un programma ad hoc che implementi il modulo software del resumer, piuttosto quest'ultimo è integrato all'interno di ogni file immagine (cfr. 2.5.1).

[24]Per ogni architettura supportata è presente il file `cp_r_regs.c` che implementa il rispettivo trampolino.

Il file “`stub_common.c`”, insieme ad altri file sorgente e non, implementa la fase in questione ed è il file da dove iniziare ad indagare per comprendere a pieno i meccanismi interni al resumer.

2.6.1 Operazioni preliminari

Una prima particolarità la si incontra nella funzione `main()` del file “`stub_common.c`”. Vengono salvate le tipiche informazioni passate all’inizio dell’esecuzione di ogni programma scritto in linguaggio C:

- il numero degli argomenti forniti da riga di comando (argc);
- la lista dei veri e propri parametri passati (argv);
- la lista delle variabili d’ambiente (envp).

In generale, quando un programma C è eseguito dal kernel, attraverso una delle funzioni `exec()`, viene chiamata una speciale procedura di inizializzazione prima che il `main()` sia eseguito. Questa prende alcune informazioni dallo spazio kernel, come ad esempio le liste dei parametri forniti da riga di comando e delle variabili d’ambiente, posizionandole sullo stack del processo creato. In seguito, innesca la funzione `main()` passandole il necessario.

Le precedenti informazioni debbono essere memorizzate perché altrimenti andrebbero perse con la rilocazione, compiuta dal resumer, dello stack; il suo indirizzo di base viene spostato al valore `0x00800000`[25] dove, per compiere la sua usuale funzione, viene mappata una pagina di memoria.

Il motivo di tale modifica è semplice. Il resumer, in generale, non può utilizzare lo spazio degli indirizzi che verrà assegnato alle informazioni da ripristinare (i.e., le VMA salvate), altrimenti sorgerebbero sicuramente malfunzionamenti. Si noti che il problema similare che si avrebbe con lo heap viene risolto mediante il meccanismo della `xmalloc()` (cfr. 2.5.3). Infine, gli stessi segmenti testo, dati e BSS (ma non solo) specifici del resumer vengono caricati in memoria principale ad indirizzi “bassi”, solitamente non utilizzati, proprio per non impegnare zone di memoria da assegnare al processo da ripristinare.

[25] Si ricordi che nelle architetture x86 lo stack cresce al contrario, da indirizzi “alti” verso i “bassi”. Non vi sarà, quindi, nessuna sovrapposizione con la memoria dedicata al trampolino.

A questo punto, vengono controllati i parametri richiesti dall'utente da linea di comando e, se risultano in regola, il vero processo di ripristino può avere inizio.

2.6.2 Ripristino delle informazioni "congelate"

La fase di *restart* necessita delle informazioni "congelate" scritte dall'eseguibile *cryopid* durante il processo di *checkpoint*. Con l'apertura in sola lettura del file `/proc/self/exe` il resumer ottiene un file descriptor all'immagine del processo da riesumare utilizzandolo per ricercare la sezione ELF col nome "cryopid.image", la quale segnale l'inizio delle informazioni da ripristinare (cfr. 2.5.2).

Ottenuto l'offset a queste ultime, non rimane altro che interpretarle e ricostituirle. Come è facilmente intuibile, viene utilizzato un procedimento inverso rispetto a quello praticato per la loro scrittura. Inizialmente viene letto l'intero `CP_CHUNK_MAGIC` che determina, ogni volta, l'inizio di una nuova informazione. Dopodiché, viene recuperato il tipo dell'informazione e in base a questo il giusto numero dei byte che la compongono. Si noti che ogni lettura è preceduta dalla decompressione dei dati attraverso lo stesso meccanismo utilizzato nella fase di checkpoint (e.g., gzip, lzo), ma questa volta impiegato per decomprimere.

Di seguito vengono descritte, nell'ordine tipico di lettura dal file immagine, le tipologie di risorse che vengono ripristinate e come avviene tale processo.

1. **TLS (Thread Local Storage).**

   Il rispristino della TLS è semplice. Dopo aver letto dal file immagine i dati costituenti una struttura dati "user_desc", mediante la system call `set_thread_area()` viene impostata la componente TLS specificata dal campo *entry_number* ivi contenuto .

2. **Spazio degli indirizzi (VMA).**

   Lo spazio degli indirizzi è una di quelle risorse il cui ripristino risulta critico. In generale, la fase di *restart* deve riportare lo stato delle VMA a quello immediatamente precedente il *checkpoint*. In CryoPID questo significa rimappare lo spazio degli indirizzi con i dati memorizzati nel file immagine,

sia per quanto riguarda le informazioni (i.e., codice, dati, etc.) relative al programma "congelato", che per le eventuali librerie dinamiche associate.

Dal file immagine vengono inizialmente letti tutti i dati relativi ad una struttura dati "cp_vma" (cfr. figura 2.2). In generale, il resumer utilizza la system call `mmap()` per ripristinare le diverse VMA, mappandole al giusto indirizzo, con i relativi flag e permessi di accesso.

Vi sono dei casi che risultano particolarmente interessanti. Nello specifico, se la VMA è appartenuta allo heap del processo obiettivo, allora tramite le syscall `brk()` e `mmap()` viene mappata su quello attuale ripristinandone i permessi di accesso tramite la `mprotect()`.

Un'altra possibilità è che la VMA non sia anonima, ovvero che abbia contenuto informazioni di una parte di un "mmapped file" che la fase di *checkpoint* non ha ritenuto necessario memorizzare (si tratta tipicamente delle zone di memoria dedicate al codice di librerie). In questo caso, si rintraccia il file originale grazie al suo nome salvato nell'immagine e se ne mappa in memoria le informazioni necessarie partendo da un determinato offset (anch'esso salvato dal processo *cryopid*).

Di ogni VMA che conteneva codice eseguibile, sia nel caso fosse stato memorizzato nel file immagine, sia nell'eventualità di doverlo recuperare dal file originale, se ne calcola il checksum e si verifica che sia identico a quello ricavato dal processo di *checkpoint*. La motivazione di tale controllo dovrebbe essere chiara: il codice deve rimanere il medesimo per ottenere una corretta ricostruzione del processo "congelato".

Quest'ultimo discorso è molto importante se si pensa alla possibilità di migrare i processi tra macchine diverse, in special modo per quanto riguarda le possibili librerie utilizzate. Anche una leggera discrepanza tra quelle impiegate nella macchina originale e quella destinataria potrebbe causare malfunzionamenti ed errori al momento del *restart*[26].

3. **File e file descriptor.**

[26] Proprio per questo motivo è possibile richiedere a *cryopid* di memorizzare anche quelle VMA dedicate al codice eseguibile (cfr. 2.4.3, punto 2).

La gestione dei file e file descriptor, come per la fase di *checkpoint*, comporta per il processo di *restart* diverse difficoltà e per questo è attualmente limitata.

In generale, il resumer deve ripristinare tutti i file che il processo aveva aperto; non solo, anche gli stessi file descriptor associati vanno ripristinati in modo tale che, una volta riesumato, il processo possa riutilizzarli nella stessa maniera e con gli stessi permessi.

Come per ogni altra risorsa da ripristinare, inizialmente vengono letti tutti i dati di base tipicamente contenuti in una struttura dati "cp_fd" (cfr. figura 2.3). In seguito, dal tipo di informazione se ne deduce quelli rimanenti da recuperare dal file immagine e, tramite diversi meccanismi, si cerca di ripristinarne la risorsa associata.

- **Device a caratteri (console).** Dal file immagine vengono estratte le informazioni costituenti una struttura dati termios. In seguito, il ruolo del file descriptor in esame viene ripristinato richiedendo il servizio di `TCSETS` (impostazione degli attributi forniti mediante struttura dati termios) alla system call `ioctl()`.
  Quest'ultima è una direttiva utilizzata per manipolare i parametri del device associato al file descriptor fornito. In questo caso, si sarebbe potuta utilizzare la funzione di libreria `tcsetattr()`, ottenendo gli stessi risultati.
- **Socket.** Gli unici supportati, almeno a livello di codice sorgente scritto, sono i socket TCP e UNIX. Come per la fase di *checkpoint*, i primi necessitano delle funzionalità offerte dal progetto "tcpcp" (TCP Connection Passing), ormai non più aggiornato. Il gruppo degli sviluppatori di CryoPID sta già pensando ad una soluzione alternativa.
  Quelli UNIX sono ripristinati mediante le tipiche funzionalità utilizzate nella programmazione di rete, anche se questa tipologia di socket è pensata per compiere IPC localmente. Dal file immagine vengono letti i dati destinati ad una struttura dati "cp_socket_unix", osservabile in figura 2.12. Dopodiché, viene creato un socket `AF_UNIX`, della categoria (`SOCK_STREAM` o `SOCK_DGRAM`) specificata nell'attributo

```
struct cp_socket_unix {
        int type, listening;
        struct sockaddr_un sockname;
        struct sockaddr_un peername;
};
```

Figura 2.12: Struttura dati “cp_socket_unix” (cpimage.h)

type della struttura dati ricavata, mediante la system call `socket()`. Se determinato nel campo *sockname*, viene associato all’indirizzo del dominio UNIX mediante la direttiva `bind()`. Per i socket UNIX, l’indirizzo da specificare è il percorso assoluto ad un file speciale da creare su disco.
In seguito, viene determinato il “ruolo” del socket; se da connettere ad un socket “remoto” oppure se da impostare in “ascolto” di richieste di connessione. Nel primo caso viene utilizzata la system call `connect()`, mentre nel secondo la direttiva `listen()`.
Infine, il file descriptor ritornato dalla syscall `socket()` viene duplicato, con `dup2()`, a favore di quello da ripristinare e, in seguito, chiuso.

- **File regolari.** Anche i file regolari vanno ripristinati, compresi quelli cancellati (tipicamente, da altri processi), a cui il processo obiettivo era legato da file descriptor.
  La struttura dati di riferimento per i file regolari è la “cp_file”, esaminabile in figura 2.13.

```
struct cp_file {
        char *filename;
        char *contents;
        int deleted;
        int size;
};
```

Figura 2.13: Struttura dati “cp_file” (cpimage.h)

Se la fase di *checkpoint* ha memorizzato il contenuto informativo del file nell’immagine, allora viene allocata tanta memoria quanta speci-

ficata nell'attributo *size*. Tali informazioni vengono poi lette dal file immagine e trasferite nella memoria appena ottenuta.
Questo meccanismo si rivela utile nel caso in cui il file fosse stato marcato come "deleted" dal processo di memorizzazione. Dapprima viene ricreato come file temporaneo, procendendo poi alla ricostruzione dei dati ivi salvati. Se, al contrario, del file è stato salvato soltanto il "filename", allora si cerca di riaprirlo con gli stessi permessi di accesso.
Infine, il file descriptor da ripristinare viene riassociato, impiegando `dup2()` e `close()`, al file in questione.

- **FIFO/pipe.** Attualmente, CryoPID riesce a ripristinare solo le pipe e soltanto quelle che collegano lo stesso processo (i.e., pipe "a se stessi"). Come al solito, oltre ai dati di base, vengono recuperate dal file immagine le informazioni necessarie ad una struttura dati "cp_fifo" (cfr. figura 2.14). Se quella da ripristinare è una pipe "a se stessi", allora ne viene creata una nuova tramite la system call `pipe()`. Non rimane altro che ripristinare i file descriptor, legandoli alla nuova pipe, mediante combinazioni di `dup2()` e `close()` di quelli appena creati.

4. **Segnali.**

   Il ripristino della gestione dei segnali è resa semplice dall'utilizzo della system call `sigaction()`.

   Dal file immagine sono estratte le informazioni costituenti una struttura dati "cp_sighand" (cfr. figura 2.4): il numero del segnale e una k_sigaction. In seguito, questi dati vengono forniti alla direttiva `sigaction()` in modo da ripristinare il gestore per lo specifico segnale, la maschera dei segnali da

```
struct cp_fifo {
        pid_t target_pid;
        int self_other_fd;
};
```

Figura 2.14: Struttura dati "cp_fifo" (cpimage.h)

bloccare durante l'esecuzione dello stesso e l'insieme dei flag determinanti il comportamento del segnale in esame.

Queste operazioni vengono, logicamente, ripetute per tutti i segnali gestiti in CryoPID e memorizzati nel file di *checkpoint*.

5. **Registri floating-point.**

   Attualmente i registri floating-point non sono ripristinati. Si sta pensando di sviluppare una soluzione simile a quella prevista per i general purpose. Per maggiori dettagli si legga il capitolo 4, "Sviluppi futuri".

6. **Registri general purpose e non solo.**

   Il ripristino di queste risorse è, generalmente, l'ultimo ad essere eseguito. Delle informazioni ricavate nella fase di *checkpoint*, l'intera struttura dati user, soltanto lo stato dei registri general purpose viene ricostituito; più precisamente, i dati memorizzati nella user_regs_struct.

   Il processo di ripristino è stato già descritto in dettaglio nella sezione 2.5.4, quella riguardante il trampolino.

2.6.3 Operazioni finali

Terminato il processo di ripristino delle informazioni memorizzate nel file immagine, il resumer esegue le operazioni che porteranno alla ripresa dell'esecuzione del processo, ormai, ricostruito.

Come accennato in precedenza, la fase di *restart* si conclude con l'esecuzione del codice implementante il trampolino attraverso la funzione `jump_to_trampoline()`.

Di seguito ne viene riportato il codice sorgente:

```
static inline void jump_to_trampoline()
{
        asm("jmp *%%eax\n" : : "a"(TRAMPOLINE_ADDR));
}
```

Figura 2.15: Funzione `jump_to_trampoline()` (stub.h)

La funzione non è altro che una istruzione assembly inline. In breve, viene caricato l'indirizzo della zona di memoria "Trampoline" (`TRAMPOLINE_ADDR`) nel registro eax per poi eseguire un `jmp` ("salto" in memoria) indiretto.

Così facendo, si ottiene la modifica del normale flusso di esecuzione del resumer, che riprende la sua computazione proprio dal codice del trampolino. Risultato finale è la ripresa della computazione del processo che era stato "congelato".

2.7 La prima "patch"

Quando si è provato per la prima volta il progetto CryoPID, questo non funzionava. In un primo momento aveva problemi in fase di compilazione. Iniziando ad analizzarlo, si è capito che il codice sorgente era datato, non aggiornato agli ultimi kernel header e *glibc*.

In particolare, CryoPID aveva problemi legati ad alcuni `#include` file che hanno avuto bisogno di un lavoro di modifiche e aggiornamenti. Il problema principale si riscontrava in fase di *restart*, dove la macro `PAGE_SIZE` forniva un valore errato, causando malfunzionamenti nella `xmalloc()`. Attualmente, questi problemi sono stati risolti grazie alle modifiche apportate nella prima "patch" presentata.

Quest'ultima contiene altre modifiche al progetto originale: viene proposta una diversa implementazione della `xmalloc()`, alternativa a quella già presente, per architetture x86.

La nuova versione risolve il problema principale della precedente `xmalloc()`, cioè l'eccessivo spreco di memoria allocata (cfr. 2.5.3). Inoltre, per come era stata implementata, le modifiche apportate comportano anche qualche miglioria sotto il profilo delle prestazioni.

L'idea è quella di strutturare maggiormente il meccanismo di mappatura della memoria a disposizione. La struttura dati pensata è riportata in figura 2.16.

Attualmente, si è pensato di utilizzare un array di "ele_area" a rappresentare l'intera zona di memoria. Ogni singolo elemento referenzia almeno un MB di memoria, scalabile a seconda della dimensione richiesta. La differenza principale con la precendente versione sta nel fatto che la memoria mappata viene utilizzata al massimo, poiché si cerca di lasciare inutilizzati meno byte possibili.

```
typedef struct {
        void *ptr_area;
        unsigned int size_left;
        unsigned int max;
} ele_area;
```

Figura 2.16: Struttura dati “ele_area” (common.c)

Ogni richiesta pervenuta alla `xmalloc()` viene prima allineata alla “parola” e poi evasa fornendo un puntatore a memoria interno ad una ele_area. Questo significa che, il più delle volte, non avviene una vera e propria allocazione; difatti, la chiamata a `mmap()` è eseguita soltanto quando ognuna delle zone già referenziate da elementi ele_area è esaurita oppure non delle dimensioni richieste. Eseguendo un numero minore di system call, la nuova `xmalloc()` risulta più performante rispetto alla precedente versione.

L’implementazione fornita è lungi dall’essere un allocatore di memoria più che sufficiente, ma è sicuramente un passo in avanti rispetto a quello presente originalmente. Le modifiche proposte non risolvono però gli altri problemi che affliggono la prima versione della `xmalloc()`; questo significa che rimangono i problema della dimensione massima della memoria allocabile e quello riguardante l’indirizzo fisso da cui iniziare ad allocare.

In realtà, il fatto stesso che la nuova versione sia basata su un array dimensionato a priori ne dimostra i limiti in scalabilità. Proprio per questo è già stato deciso di passare ad una gestione a lista dinamica di ele_area che permetterà, inoltre, di abbassare la dimensione minima di allocazione ad una pagina di memoria (tipicamente, 1 KB nelle architetture x86) con ulteriore diminuzione delle occasioni di spreco.

2.8 Attuali limitazioni

In questa ultima sezione viene affrontato il discorso concernente tutte le attuali limitazioni del progetto CryoPID. Vi sono delle scusanti.

In generale, il progetto è relativamente giovane se si pensa che il suo sviluppo è rimasto fermo per un lungo periodo, almeno quattro anni. In aggiunta, va detto

che la natura implementativa in "user space" del lavoro e la complessità oggettiva dei concetti concorrono alle limitazioni di CryoPID.

Di seguito vengono riportate quelle più rilevanti riguardanti il progetto, ognuna corredata da un commento sul rispettivo impatto nell'impiego concreto del CRS CryoPID.

Tipologia di applicazioni. Ad oggi, CryoPID è in grado di "congelare" e riattivare applicazioni formate da un solo processo costituito da un singolo thread di esecuzione. È sicuramente uno dei limiti più significativi del progetto se si pensa che, se non in casi banali, la maggioranza degli applicativi utilizzati è formata da diversi processi o, almeno, da più thread di esecuzione.

Tipologie di file. Le tipologie di file supportate da CryoPID sono: device a caratteri (limitato alla console utilizzata), file regolari, socket (nella pratica, limitato a quelli UNIX) e FIFO (limitato alle normali pipe). Il supporto è davvero limitato.

Soprattutto la mancanza di supporto alle named pipe e ai socket di rete ha, tipicamente, un grande impatto sulla diffusione di un software di *checkpoint/restart* come CryoPID.

File regolari. In generale, i file aperti ed i rispettivi offset sono gestiti dal *checkpoint/restart* in CryoPID. È presente anche un buon supporto ai file temporanei che non sono più accessibili attraverso il file system. Non vi è però, né la possibilità di memorizzare il contenuto dei file aperti all'interno del file immagine, né l'implementazione di un sistema alternativo a questo (e.g., file di copia nascosti). In definitiva, scenari di *restart* in cui non comparissero file necessari al processo non sono supportati.

È una mancanza non da poco, anche se risulta essere una delle problematiche più complesse da affrontare.

Socket. Attualmente, il codice sorgente di CryoPID prevede soltanto i socket TCP e UNIX. I primi erano gestiti utilizzando il progetto "tcpcp" ormai fermo e non aggiornato. Sistemi software come CryoPID dovrebbero avere una buona

gestione dei socket perché fondamentali per numerose applicazioni, anche se non è affatto semplice implementare un supporto completo, soprattutto a livello utente.

Applicazioni grafiche. È presente una minima implementazione di supporto alle applicazioni grafiche che utilizzano le librerie GTK+. Il codice scritto a tale scopo, però, è abbastanza superato e trascurato perché scritto agli inizi del progetto e oggi non più aggiornato.

Anche il lettore meno esperto compreder à che la gestione delle applicazioni grafiche è molto importante, soprattutto se l'obiettivo del software di *checkpoint/restart* è generico e comprendente anche le normali applicazioni eseguite dagli utenti in ambiente desktop. Risulta chiaro che un buon supporto a tali applicativi determinerebbe una maggiore diffusione del progetto.

procfs. Il file system virtuale procfs è un ottimo strumento, utile per l'analisi di tutte le più importanti risorse utilizzate da un processo. Lo stesso eseguibile *cryopid* ne fa un largo uso.

È abbastanza comprensibile che, in fase di *restart*, procfs venga popolato di tutte quelle informazioni riguardanti il processo riesumato. Attualmente, questo non avviene in modo completamente corretto. Ad esempio, le informazioni riguardanti lo spazio degli indirizzi non sono accurate come quelle presenti in fase di *checkpoint*.

Un altro problema, abbastanza "fastidioso" perché non sempre verificato, si riscontra nel mancato ripristino del nome del processo in `/proc/`*`pid`*`/cmdline`, tanto che il programma "ps" spesso non riporta il processo riesumato tra quelli attivi.

Locazione del resumer. Nell'implementazione attuale, il codice del resumer è posto all'indirizzo di memoria `0x00100000` (cfr. figura 2.7).

La fase di *checkpoint* evita di processare e memorizzare lo spazio degli indirizzi dedicato a tale scopo perché inutile ai fini del *restart*. Questo significa che nel caso il processo obiettivo utilizzi lo spazio di memoria in esame, la sua memorizzazione fallirebbe perché scambiato per il resumer.

Dovrebbe essere abbastanza facile comprendere la gravità di tale limitazione. Un piccolo appiglio di salvezza è dovuto al fatto che lo spazio degli indirizzi al di sotto di `0x08048000` non è, in generale, utilizzato dai normali processi.

vDSO. L'acronimo vDSO sta per "virtual Dynamic Shared Object", noto anche con il nome "linux-gate.so.1". Come suggerisce il nome, il vDSO è uno shared object virtuale (non esiste un vero e proprio file) esposto dal kernel e mappato ad un indirizzo specifico, tipicamente `0xffffe000` in ambito x86, nello spazio degli indirizzi di ogni processo. La sua funzione principale è quella di "cancello" tra lo spazio "user" e quello "kernel", cioè quella di eseguire le system call.

L'idea del vDSO è nata con l'arrivo, nelle architetture più recenti, delle nuove istruzioni `sysenter` e `sysexit` per l'interfacciamento alle system call. Questo significa che, oltre alla famosa `int0x80`, vi è un secondo modo di invocarle, decisamente più veloce ed efficiente.

Il kernel decide, durante la fase di boot, quale sistema adottare impostando, infine, il meccanismo scelto all'interno del vDSO. Intuitivamente questo vuol dire che, ogni qual volta sia necessario eseguire una syscall, non si utilizza direttamente l'istruzione dovuta, bensì viene compiuta una `call` al vDSO che poi si incarica di invocare, tramite il meccanismo selezionato, la system call richiesta.

Il vDSO è all'interno del kernel Linux da diverso tempo. Una completa attivazione causa la randomizzazione della sua locazione all'interno dello spazio degli indirizzi dei processi. In altre parole, processi diversi (oppure lo stesso eseguito in momenti diversi) potrebbero presentare il vDSO mappato ad indirizzi diversi. Il motivo principale va ricercato nel campo della sicurezza: la randomizzazione della posizione del vDSO diminuisce la probabilità di successo di alcuni tipi di attacco.

Attualmente, il progetto CryoPID supporta soltanto la modalità "compact vDSO", cioè la versione mappata ad un indirizzo di memoria fisso in ogni processo. La motivazione dovrebbe, ormai, risultare chiara.

Il processo ripristinato deve continuare a lavorare senza la minima problematica legata al suo *restart*; in particolare, lo stato della sua memoria deve essere identico a quello antecedente il *checkpoint*. Evidentemente, questo non accadrebbe in presenza di randomizzazione del vDSO, causando errori irreversibili.

Il problema non è banale, anche perché le maggiori distribuzioni Linux adottano, in modo predefinito, il vDSO completamente attivo. In realtà, la sua modalità di utilizzo è modificabile a piacimento attraverso procfs. Le conseguenze legate alla sicurezza rimangono comunque un problema aperto.

Randomizzazione della memoria. Un discorso simile al precedente va fatto per la tecnica nota come ASLR, ovvero "Address Space Layout Randomization". Questa è stata ideata per ridurre al minimo l'efficacia degli attacchi di tipo "return-to-libc", i quali sfruttano la posizione fissa di alcuni elementi nel layout di memoria dei processi (e.g., stack, heap, etc.). Tipicamente, l'attacco inizia con un buffer overflow attraverso il quale un indirizzo di ritorno, posto in cima allo stack, viene rimpiazzato con l'indirizzo di una diversa funzione. In seguito, un'altra parte dello stack è sovrascritta per fornire gli argomenti a quest'ultima. Questo permette all'attaccante di eseguire codice maligno senza doverlo "iniettare" all'interno del processo obiettivo. Il nome dell'attacco è dovuto alla funzione che viene fatta eseguire.

Infatti, sebbene l'attaccante potrebbe impostare liberamente il codice di ritorno, la *libc* è l'obiettivo più utilizzato perché presente in qualsiasi processo e fornisce numerose funzioni "appetibili" (e.g., la funzione `system()` permette di eseguire qualsiasi programma).

Questi attacchi sfruttano il fatto che gli indirizzi di importanti aree di memoria, quali lo stack e lo heap, hanno valori fissi e noti a priori. La loro randomizzazione diminuisce la probabilità di successo degli attacchi perché richiede che il codice maligno "indovini" la posizione di queste aree.

Ora, la randomizzazione della locazione di queste ultime crea numerosi e gravi problemi alla fase di *restart* per gli stessi motivi descritti in precedenza, tanto che i sistemi che adottano tecniche di ASLR[27] non sono supportati dal progetto.

Quanto detto per il vDSO vale anche in questo caso. In generale, un qualsiasi progetto non è ben accetto se causa insicurezza o "falle" nel sistema in cui lo si vorrebbe impiegare.

[27] Si pensi, ad esempio, ai progetti PaX (http://pax.grsecurity.net/) ed Exec Shield.

Capitolo 3

Scherzare la *glibc* con un hack!

Person who say it cannot be done
should not interrupt person doing it.
– Chinese proverb

3.1 Un semplice test...

Dopo avere sviluppato la prima “patch” al progetto CryoPID, si è continuata una intensa attività di studio e testing del suo codice sorgente.

CryoPID fornisce anche una serie di semplici programmi di test, che mostrano all’utente le potenzialità del progetto. Ora, chiunque si aspetterebbe che questi funzionino senza problemi, proprio perché rilasciati insieme al progetto stesso.

In realtà, anche CryoPID si è rivelato un campo fertile per la famosa *legge di Murphy*[1]. Quasi per caso, mentre si stava presentando il progetto ad un compagno di studi, ci si è accorti che il test “sigtest”, una volta eseguito il suo *restart*, non funzionava come dovuto; sembrava bloccato.

Il codice sorgente del test è veramente semplice da comprendere. In generale, viene provata la gestione dei segnali; ovvero, se sono memorizzati e ripristinati in modo corretto.

Come visibile in figura 3.1, in cui sono state rimosse le direttive `#include`, vengono installati gestori personalizzati per ognuno dei segnali `SIGUSR1` e `SIGUSR2`

[1] Legge di Murphy: “*Se qualcosa può andar male, andrà male.*”

e, in seguito, viene eseguito un ciclo infinito di autonotifiche mediante chiamate `raise()`.

```
void sig_handler(int sig) {
        printf("Got signal %d!\n", sig);
}

int main() {
        signal(SIGUSR1, sig_handler);
        signal(SIGUSR2, sig_handler);
        for(;;) {
                raise(SIGUSR1);
                sleep(2);
                raise(SIGUSR2);
                sleep(2);
        }
}
```

Figura 3.1: File di test "sigtest.c"

La dichiarazione della `raise()` si trova nel file `signal.h`, mentre il suo prototipo è il seguente:

int raise(int sig);

Questa funzione di libreria spedisce il segnale specificato nel parametro *sig* al processo chiamante. Considerando il codice precedente e una corretta fase di *restart* del processo, l'output dovrebbe essere costituito da una serie infinita di stampe a video alternate, come le seguenti:

```
Got signal 10!
Got signal 12!
Got signal 10!
Got signal 12!
[...]
```

In precedenza si è accennato al comportamento errato del processo riesumato che sembrava bloccato, dato che niente veniva stampato in output sulla console.

Si è poi compreso, invece, che il ripristino dei segnali avveniva correttamente in quanto, spedendoli "manualmente" al processo[2], questi venivano gestiti sen-

[2] Tipicamente, mediante il comando shell `kill`.

za alcun problema. In particolare, spedendo i segnali `SIGUSR1` e `SIGUSR2` si otteneva l'output atteso.

3.2 `getpid()` caching

Assodata la corretta gestione dei segnali da parte del processo di *restart*, ci si è concentrati sulla funzione `raise()` cercando di comprenderne a fondo il funzionamento. La sua implementazione è banale e, come riportato nella rispettiva pagina del manuale, risulta equivalente alla seguente istruzione:

```
kill(getpid(), sig);
```

La system call `kill()` è alla base del sistema dei segnali in ambienti UNIX-like. Permette di spedire il segnale *sig* al processo il cui *pid* figuri come primo parametro.

Verificato che tale syscall non avesse bug, il cerchio si è stretto attorno alla funzione `getpid()`, una delle più utilizzate in ogni programma. Il valore ritornato da ogni suo utilizzo è l'identificatore del processo invocante. Fin qui nulla di strano, tutto estremamente semplice e chiaro.

Leggendo, però, il manuale della `getpid()` ci si accorge della sezione "`NOTES`" in cui sono riportate alcune peculiarità riguardanti l'implementazione della funzione in base alla versione della GNU *glibc*. In particolare, si legge che dalla versione 2.3.4 della libreria la funzione wrapper[3] alla system call `getpid()` esegue il caching del PID in modo da ridurre i costi computazionali derivanti da possibili chiamate successive alla stessa. Questo significa che, durante l'esecuzione di un processo che fa uso della `getpid()`, la syscall vera e propria viene eseguita soltanto la prima volta e il PID inserito in una qualche zona di memoria destinata alla libreria. In questo modo, ad ogni nuova invocazione viene ritornato il PID salvato, senza ricorrere ad una system call.

Va detto che è pratica comune l'utilizzo dei wrapper in una libreria come la *glibc* perché permettono al programmatore l'impiego di una interfaccia, ai servizi del sistema operativo, comune e più semplice da utilizzare.

[3] In generale, un *wrapper* è una funzione che ne chiama una seconda con piccole computazioni addizionali.

È anche vero, però, che la `getpid()` non è una syscall "lenta" e quindi non si comprende questa scelta implementativa. Probabilmente ciò si deve a motivi di performance riconducibili al suo massiccio utilizzo all'interno delle libreria stessa. Quello che è certo è che tale modifica non è stata accettata[4] *cordialmente* dalla comunità di sviluppo del kernel Linux, fin dalla sua introduzione.

3.3 Il problema

Per il progetto CryoPID l'idea del "PID caching" si è dimostrata un disastro perché alla base del malfunzionamento della funzione `raise()` e di tutti quei possibili scenari in cui si farebbe affidamento sul corretto risultato ritornato dalla system call `getpid()`.

Il perché di tale affermazione sarà chiaro dopo la seguente spiegazione delle cause che portano al malfunzionamento del processo "sigtest" una volta ripristinato dalla fase di *restart*.

Nel capitolo precedente si è descritto nei dettagli quali risorse vengono considerate durante il processo di *checkpoint*. Tra queste, la TLS (Thread Local Storage) ricopre un ruolo fondamentale per il problema che si vuole esporre. In particolare, è proprio la zona dedicata alla TLS di ogni processo a mantenerne il PID dopo la prima invocazione della `getpid()`.

Memorizzando l'intera TLS viene salvato nel file immagine anche il PID del processo obiettivo. Questo significa che durante la fase di *restart*, insieme alla TLS, anche questo è ripristinato dal resumer.

Avendo presente l'implementazione della funzione `raise()`, riportata nella sezione precedente, risulta chiaro il perché del suo comportamento errato. La `getpid()` invocata internamente, analizzando la TLS ripristinata, ritorna il PID ivi memorizzato: l'identificatore del processo obiettivo al tempo del suo *checkpoint*.

Questo è, ovviamente, un errore colossale perché quello ritornato non è, di certo, il vero PID del processo riesumato. Infatti, quello ripristinato è pur sempre un nuovo processo, anche se ricostruito in base alle informazioni memorizzate nel

[4] Si rimanda alla *colorita* discussione apparsa su LKML: http://lkml.org/lkml/2004/6/5/61

file immagine. Ciò significa che il kernel gli assegna un nuovo identificatore, un PID diverso da quello memorizzato nella TLS ripristinata.

In definitiva, il funzionamento della `getpid()` induce all'errore la funzione `raise()` che spedisce il segnale al vecchio processo, che potrebbe anche essere "defunto"[5]. Particolarmente interessante ed esplicativo è il codice sorgente della `getpid()`, del quale viene riportato di seguito un frammento.

```
pid_t
__getpid (void)
{
#ifdef NOT_IN_libc
        INTERNAL_SYSCALL_DECL (err);
        pid_t result = INTERNAL_SYSCALL(getpid, err, 0);
#else
        pid_t result = THREAD_GETMEM(THREAD_SELF, pid);
        if (__builtin_expect (result <= 0, 0))
                result = really_getpid (result);
#endif
        return result;
}
```

Figura 3.2: Frammento di implementazione della `getpid()`

La parte incriminata è quella scritta nel ramo `#else` della condizione di compilazione. In breve, viene recuparato il PID del processo chiamante dalla TLS e verificato il suo valore. Soltanto nel caso in cui risultasse essere minore o uguale a zero, allora verrebbe chiamata la "really_getpid". È proprio quest'ultima ad eseguire, effettivamente, la system call per poi memorizzare il valore restituito nella TLS.

Altrimenti, se il valore risultasse già memorizzato in precedenza, viene ritornato il PID senza eseguire nessuna syscall.

In conclusione, a conferma della natura del problema, si è cercato di "congelare" il test prima che questo eseguisse la `getpid()`, con conseguente caching del PID, ottenendo il comportamento corretto una volta eseguito il suo *restart*.

[5]Il lettore avrà quindi intuito che, in realtà, il "sigtest" ripristinato non era bloccato, bensì segnalava il processo sbagliato (quello originale).

3.4 Un *hack* come soluzione

Va sottolineato che il problema descritto finora è dovuto al wrapper implementato dalla *glibc*. Se, infatti, il codice del test invocasse la relativa system call direttamente, senza il supporto della libreria, non si incontrerebbe alcun malfunzionamento. Ad esempio, basterebbe utilizzare la direttiva `syscall()` fornendole il numero della *getpid* (`SYS_getpid`).

Dovrebbe risultare evidente che l'utilizzo di questo espediente per rimpiazzare l'impiego della `getpid()` non può essere considerato una buona soluzione al problema. Non è pensabile obbligare il programmatore a questa metodologia nel caso voglia dotarsi delle funzionalità offerte da CryoPID. Inoltre, il progetto stesso nasce con la precisa volontà di non imporre la modifica del codice sorgente degli applicativi da "congelare".

L'unica via praticabile è l'aggiornamento della cache, invalidandola oppure forzando la `getpid()` alla memorizzazione del vero PID. Nel primo caso, basterebbe invalidare la cache del processo obiettivo durante la fase di *checkpoint*, ottenendo la memorizzazione del vero PID alla prima chiamata `getpid()` effettuata dopo il processo di ripristino. L'altro approccio è maggiormente diretto e, in generale, utilizzabile soltanto in fase di *restart*.

Sicuramente, la prima idea è più "pulita" ed è quella che andrebbe presa in considerazione, se non fosse che risulti non utilizzabile in pratica. Attualmente, non vi è alcuna funzione messa a disposizione del programmatore da parte della libreria perché si riesca nell'intento di azzerare la cache in questione. In generale, servirebbe una funzione che eseguisse, almeno, le seguenti operazioni:

```
void
getpid_cache_clear (void)
{
#ifndef NOT_IN_libc
        THREAD_SETMEM (THREAD_SELF, tid, 0);
        THREAD_SETMEM (THREAD_SELF, pid, 0);
#endif
}
```

Figura 3.3: Funzione per l'azzeramento della cache

Quella precedente è l'implementazione proposta dal professore Renzo Davoli agli sviluppatori della libreria *eglibc*[6], i quali si sono dimostrati non da meno dei "cugini" della *glibc*. Almeno per ora, una tale funzionalità non verrà supportata.

La seconda idea è quella concretamente implementata e fornita come seconda "patch", attualmente disponibile solo per architetture x86, al progetto CryoPID. La finalità di tale soluzione è di costringere il wrapper di `getpid()` all'aggiornamento della cache, ovvero del valore salvato presso la TLS del processo da ripristinare.

Come descritto in precedenza, l'implementazione fornita dalla libreria prevede un frammento di codice sorgente che effettua la vera e propria system call per poi salvare in cache il valore ritornato. L'intenzione è di sfruttare tale possibilità, cercando di bypassare il controllo eseguito a monte sul contenuto della TLS. In questo modo, il valore del PID verrà, sicuramente, aggiornato a quello reale.

In primo luogo, si è analizzato il codice assembly della funzione `getpid()` in modo da comprenderne a pieno l'implementazione. Quello seguente è stato ricavato dalla libreria dinamica `/lib/libc-2.9.so` presente in un sistema Debian GNU/Linux "unstable":

```
00098520 <__getpid>:
   98520: 65 8b 15 4c 00 00 00    mov %gs:0x4c,%edx
   98527: 83 fa 00                cmp $0x0,%edx
   9852a: 89 d0                   mov %edx,%eax
   9852c: 7e 02                   jle 98530 <__getpid+0x10>
   9852e: f3 c3                   repz ret
   98530: 75 0a                   jne 9853c <__getpid+0x1c>
   98532: 65 a1 48 00 00 00       mov %gs:0x48,%eax
   98538: 85 c0                   test %eax,%eax
   9853a: 75 f2                   jne 9852e <__getpid+0xe>
   9853c: b8 14 00 00 00          mov $0x14,%eax
   98541: cd 80                   int $0x80
   98543: 85 d2                   test %edx,%edx
   98545: 89 c1                   mov %eax,%ecx
   98547: 75 e5                   jne 9852e <__getpid+0xe>
   98549: 65 89 0d 48 00 00 00    mov %ecx,%gs:0x48
   98550: c3                      ret
```

Figura 3.4: Funzione `getpid()` in codice macchina e assembly

[6]Una distribuzione della *glibc*, compatibile anche a livello binario, alla quale ultimamente sono passati progetti come Debian GNU/Linux (http://www.eglibc.org/home).

In generale, si divide in due aree funzionali[7]. Nella prima avviene il controllo del valore mantenuto nella TLS, mentre nella seconda è implementato il meccanismo che si vuole sfruttare. Più precisamente, quest'ultima sezione inizia all'indirizzo `0x0009853c`.

Viene effettuata la system call `getpid()` mediante il caricamento del rispettivo numero (`0x14`) nel registro eax e l'esecuzione dell'interrupt software `int0x80`. In seguito, se il tid (i.e., identificatore del thread) in edx è uguale a zero, il valore ritornato dalla syscall appena eseguita viene memorizzato nella TLS, attraverso l'istruzione `mov %ecx,%gs:0x48`.

Risulta evidente che tale implementazione della funzione `getpid()` non sia, in generale, la stessa per ogni distribuzione Linux e in ogni versione della libreria *glibc*. In realtà, questo discorso è corretto solo in parte perché le maggiori distribuzioni forniscono, tipicamente, le ultime versioni della *glibc*, la quale contiene da tempo il meccanismo di caching in esame.

3.5 `getpid()` hack in CryoPID

Checkpoint. Dallo studio del codice assembly si evince che attraverso `%gs:0x48` è possibile accedere alla locazione, interna alla TLS, in cui viene memorizzato il PID. In generale, questa preziosa informazione basterebbe a risolvere il problema perché permetterebbe addirittura, tramite poche istruzioni, di invalidare direttamente la cache. Nella pratica, però, non è stata presa in considerazione una implementazione così sbrigativa, se non per alcuni test di fattibilità svolti agli inizi di questo lavoro di tesi.

La ragione principale è la portabilità di tale soluzione. È vero; quello che si vuole implementare è un *hack*, ma questo non significa che non lo si voglia il più "pulito" possibile e in grado di funzionare su un numero elevato di distribuzioni Linux e con differenti versioni della *glibc*. Infatti, non è possibile affermare con certezza la validità generale di `%gs:0x48`, anche se vi sono buone probabilità che lo sia.

Si è cercato, quindi, di implementare una soluzione maggiormente generica.

[7] È abbastanza ovvio, in quanto il codice assembly non è nient'altro che una differente rappresentazione dello stesso sorgente scritto, in questo caso, nel linguaggio C.

L'idea di base è abbastanza semplice: appoggiarsi alla *glibc* utilizzata dal processo obiettivo per recuperare il frammento di codice appartenente alla `getpid()` necessario all'implementazione dell'*hack*. Una volta recuperato quanto opportuno, deve essere fatto eseguire al resumer prima che il controllo venga restituito al processo ricostruito.

In primo luogo, durante la fase di *checkpoint* dello spazio degli indirizzi viene individuata la memoria mappata alla *libc*, per carpirne il nome del file di origine. Questo viene poi aperto in lettura e analizzato nei dettagli.

I file contenenti codice macchina e dati appartenenti a librerie sono "shared object"; in generale, vengono utilizzati dal "link editor" nella creazioni di file oggetto oppure combinati dal "dynamic linker" con un file eseguibile per la costruzione di un processo. Essenzialmente, sono file ELF del tutto similari, almeno nella struttura, a quelli eseguibili. La differenza sostanziale sta nel fatto che non sono direttamente processabili, non hanno un ruolo fine a se stessi.

Nel file della *libc* viene cercato il simbolo di funzione `__getpid`, che identifica la sezione contenente il codice riportato, in precedenza, in figura 3.4. A tale scopo, vengono prima individuate le sezioni `.dynsym` e `.dynstr` che mantengono, rispettivamente, la tabella dei simboli utile al linking dinamico e un insieme di stringhe rappresentanti i nomi associati agli elementi della stessa.

Ognuno di essi è rappresentato dalla seguente struttura dati, definita nel file `elf.h`:

```
typedef struct {
        Elf32_Word      st_name; /* Symbol name */
        Elf32_Addr      st_value; /* Symbol value */
        Elf32_Word      st_size; /* Symbol size */
        unsigned char   st_info; /* Symbol type and binding */
        unsigned char   st_other; /* Symbol visibility */
        Elf32_Section   st_shndx; /* Section index */
} Elf32_Sym;
```

Figura 3.5: Elemento della tabella dei simboli

Dopo aver trovato tale struttura dati per il simbolo `__getpid`, mediante l'attributo *st_value* si ottiene la posizione, all'interno del file ELF, del codice macchina implementante tale funzione. Non rimane altro che estrarre il frammento

di codice utile al completamento dell'*hack*. Questo significa dover riconoscere quello implementante la parte interessante dell'intera funzione `getpid()`, ci si riferisce alla seconda sezione funzionale (cfr. 3.4).

Un semplice metodo è quello della ricerca di una "signature", cioè di una sequenza di byte che identifichi univocamente il codice macchina da considerare. Problema fondamentale di tale approccio è la scelta, non banale, di quanti e quali byte ritenere necessari allo scopo. È abbastanza intuibile che prendendo in esame una successione troppo corta si potrebbe, erroneamente, prendere in considerazione dei falsi positivi. Al contrario, una serie troppo lunga potrebbe indurre falsi negativi.

Per quanto riguarda il problema della `getpid()`, è stato più semplice del previsto trovare una sequenza di byte presente, sicuramente, in tutte le attuali *glibc* e univoca nell'arco dell'intera implementazione della funzione. Infatti, come il lettore avrà già notato, l'istruzione assembly `mov $0x14,%eax` è presente soltanto una volta[8] e determina proprio l'inizio della parte di funzione da sfruttare. Il codice sorgente scritto allo scopo di identificare tale istruzione è basato su un algoritmo naïve di "string matching", davvero elementare.

L'algoritmo non fa altro che leggere cinque byte alla volta appartenenti alla funzione `getpid()` e confrontarli con quelli costituenti la signature da ricercare. Per assicurare una terminazione certa del ciclo di ricerca, ogni blocco letto viene confrontato anche con un "modello di uscita" costituito dai byte delle ultime cinque istruzioni di codice macchina della funzione.

Una volta riconosciuta la signature cercata, questa viene copiata per essere poi salvata all'interno del file immagine. In figura 3.6 vengono riportati, rispettivamente, i due frammenti del codice sorgente implementante quanto discusso finora.

Come ogni altra informazione recuperata durante la fase di *checkpoint*, anche la signature appena estratta dal codice macchina della funzione `getpid()` deve essere memorizzata nel file immagine.

Proprio per questo è stato necessario aggiungere una struttura dati a quelle già presenti; a tal proposito si veda la figura 3.7.

---

[8] Si noti che, come ci si aspetta, anche a livello di codice macchina i byte dell'istruzione sono univoci nell'arco dell'intera funzione.

```
/* first bytes of the signature searched: mov $0x14, %eax */
const unsigned char const_signature[] = {0xb8, 0x14, 0x00, 0x00, 0x00};
/* nop signature, end of __getpid function */
const unsigned char const_null_signature[] = {0xc3, 0x90, 0x90, 0x90, 0x90};

[...]

/* looking for a good signature.
we are looking for machine code of mov $0x14,%eax.
it is SYS_GETPID first sign */
while(1) {
    if (read(libc_fd,&code[0],CMP_SIGNATURE_LENGTH) < CMP_SIGNATURE_LENGTH) {
        info("[E] failed to read from __getpid");
        return EXIT_FAILURE;
    }
    /* compare the code fetched against the const signature */
    if (memcmp(code, const_signature, CMP_SIGNATURE_LENGTH) == 0)
        break;
    /* end of __getpid, no good */
    if (memcmp(code, const_null_signature, CMP_SIGNATURE_LENGTH) == 0)
        return EXIT_FAILURE;
    /* seek back to get ready for fetching other CMP_SIGNATURE_LENGTH bytes
    because the read() advances the file offset */
    if (lseek(libc_fd, sizeof(unsigned char)-CMP_SIGNATURE_LENGTH, SEEK_CUR) == -1) {
        info("[E] failed to seek during fetch __getpid signature loop");
        return EXIT_FAILURE;
    }
}

[...]

*buffer = (char*) xmalloc(SIGNATURE_LENGTH);
/* fetch the signature from libc */
memset(*buffer, 0, SIGNATURE_LENGTH);
if (read(libc_fd, *buffer, SIGNATURE_LENGTH) == -1) {
    info("[E] unable to fetch the signature");
    return EXIT_FAILURE;
}
```

Figura 3.6: Verifica ed estrazione della *signature* (`getpid_hack_write.c`)

```
struct cp_getpid {
        char* code;
};
```

Figura 3.7: Struttura dati “cp_getpid” (cpimage.h)

Ovviamente, questa viene inserita nella lista delle informazioni da memorizzare; la proc_image. Subisce il processo di compressione per poi essere scritta, come ultima delle informazioni riguardanti il processo obiettivo, nel file di *checkpoint*.

Restart. Le informazioni in esame vengono lette dal file immagine per ultime e quindi sono ripristinate quando ormai il processo di *restart* è prossimo a terminare la proprio funzione. Vengono prima decompresse e, in seguito, i byte costituenti il codice macchina estratto dalla libreria *glibc* sono posti in un buffer allocato dinamicamente mediante `xmalloc()`.

È il momento chiave dell'intero *hack*. Come anticipato, il codice macchina estrapolato dalla funzione `getpid()` deve essere fatto eseguire dal resumer prima che il controllo sia restituito al processo ricostruito. L'idea è di sfruttare un meccanismo del tutto similare a quello alla base del trampolino (cfr. 2.5.4).

In particolare, viene mappata una pagina di memoria anonima accessibile in lettura, scrittura ed esecuzione tramite la system call `mmap()`. La zona richiesta è immediatamente sopra quella allocata per contenere il codice implementante il trampolino. Per comprendere meglio il discorso, si guardi la seguente figura:

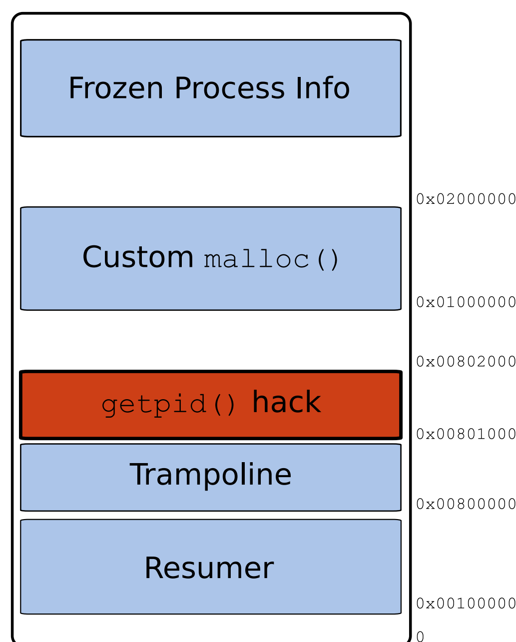

Figura 3.8: Layout del file immagine in memoria principale, con *hack*

Nella zona appena allocata sono inseriti i byte del codice macchina necessari a forzare l'aggiornamento della cache appartenente alla TLS. Perché l'*hack* funzioni a dovere almeno questa deve essere già stata ripristinata, per ovvie motivazioni. Ecco spiegato il motivo per cui l'intervento di *restart* in esame viene eseguito per

ultimo. Non solo, il registro di segmento GS deve essere settato in modo da indirizzare nuovamente la TLS appena ripristinata. Anche questa operazione può essere effettuata soltanto pochi istanti prima di restituire il controllo al processo riesumato perché, altrimenti, potrebbero innescarsi dei malfunzionamenti nella fase di *restart*.

Il codice sorgente implementante quanto detto è riportato di seguito:

```
char *dest = (char*)GETPID_HACK_ADDR;

[...]

/* Create region for __getpid refresh hack */
mmap((void*)GETPID_HACK_ADDR,_getpagesize,PROT_READ|PROT_WRITE|
        PROT_EXEC,MAP_FIXED|MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);

/* set up gs */
if (!emulate_tls && r->xgs != 0) {
    /* mov foo, %eax  */
    *dest++=0x66;*dest++=0xb8; *(short*)(dest) = r->xgs; dest+=2;
    /* mov %eax, %gs */
    *dest++=0x8e;*dest++=0xe8;
    /* mov $0x00, %edx */
    *dest++=0xba;*dest++=0x00;*dest++=0x00;*dest++=0x00;*dest++=0x00;
    /* mov $0x00, %ecx */
    *dest++=0xb9;*dest++=0x00;*dest++=0x00;*dest++=0x00;*dest++=0x00;

    /* insert the machine code of __getpid */
    memcpy(dest, data, SIGNATURE_LENGTH);
}

/* restoring VMA previous settings */
mprotect((void*)GETPID_HACK_ADDR,_getpagesize,PROT_READ|PROT_EXEC);
```

Figura 3.9: Principale passaggio nell'implementazione dell'*hack*

Si noti come le operazioni non vengano eseguite, ma scritte in memoria dereferenziando la variabile puntatore *dest*, la quale indirizza proprio la zona di memoria allocata allo scopo. Anche la stessa sequenza estrapolata dalla `getpid()` viene inserita direttamente in memoria attraverso `memcpy()`[9].

Come per il trampolino, anche il codice macchina appena inserito deve essere

[9] Il lettore avrà, adesso, compreso perché è stato necessario recuperare il *codice macchina* della funzione `getpid()`.

processato dal resumer per essere effettivamente eseguito. A tale proposito è stata scritta la seguente funzione `jump_to_getpid_hack()`:

```
static inline void jump_to_getpid_hack()
{
        asm("call *%%eax\n" : : "a"(GETPID_HACK_ADDR));
}
```

Figura 3.10: Funzione `jump_to_getpid_hack()` (stub.h)

La funzione non è altro che una istruzione assembly inline, attraverso la quale viene prima caricato nel registro eax l'indirizzo della zona di memoria allocata per l'*hack* (`GETPID_HACK_ADDR`) e, in seguito, eseguita una `call` ("chiamata a funzione") indiretta.

Come per una istruzione `jmp`, anche la `call` permette di modificare il normale flusso di esecuzione, in questo caso, del resumer. Si è scelta quest'ultima perché controparte naturale dell'istruzione `ret` ("ritorno da funzione"), posta al termine del codice estrapolato dalla `getpid()`.

La soluzione implementata non è esente da problematiche più o meno gravi.

Innanzitutto, il fatto di essere un *hack* comporta limitazioni intrinseche. La più imporante è la dipendenza alla specifica implementazione della `getpid()`. Anche una minima modifica, in realtà assai rara, a quest'ultima causerebbe la "rottura" del meccanismo sviluppato.

Inoltre, la soluzione non è completamente libera da dipendenze architetturali. Di certo, è molto più indipendente di quella discussa per prima (i.e., la proposta iniziale di *hack*), anche solo per il fatto di "appoggiarsi" all'implementazione della `getpid()` fornita dalla *glibc* utilizzata dal processo obiettivo. È altrettanto sicuro che una funzione "ufficiale" pensata per invalidare la cache sarebbe la soluzione ottimale.

Un più serio problema è dovuto alla presenza di programmi compilati staticamente (scarsamente diffusi ma comunque presenti, specialmente nel caso di software chiuso o proprietario). Attualmente, una tale soluzione non funzionerebbe come dovuto.

Va ricordato, infine, che la soluzione proposta è per sua natura un *hack*, cioè un espediente non particolarmente elegante, ma che permette di ottenere il comportamento desiderato altrimenti non previsto (in questo caso, dalla *glibc*).

3.6 `LD_PRELOAD`, una soluzione alternativa

Oltre alla soluzione finora proposta, se ne è cercata una più generica ed elegante. Si è arrivati a pensare alla "virtualità" come concetto da sfruttare per risolvere il problema legato al caching implementato nella *glibc*.

In un primo momento si è provato ad implementare una soluzione basata sul meccanismo di tracciamento fornito da `ptrace()`. L'idea era di sfruttare tale system call per catturare le istruzioni eseguite dal processo riesumato in modo da sostituire ogni chiamata alla funzione `getpid()` con la reale syscall.

Si è presto compreso che una soluzione del genere, anche se implementabile, è inattuabile nella pratica. In particolare, per come è concepita la system call `ptrace()`, un tracciamento step-by-step sarebbe troppo pesante dal punto di vista computazionale.

In seguito, si è ripiegato su un meccanismo di ridefinizione delle chiamate a librerie dinamiche. Come accennato in precedenza, questa implementazione sfrutta il processo di preloading di librerie dinamiche tramite l'impostazione della variabile d'ambiente `LD_PRELOAD`, permettendo di ridefinire un insieme arbitrario di chiamate di libreria (in questo caso, della funzione `getpid()`).

In realtà, non si è implementato nessun nuovo componente, bensì si è utilizzato il lavoro finora effettuato dal team di sviluppo del progetto "Virtual Square"[10] del professore Renzo Davoli. Tra i tanti sviluppati, si è provata la "PureLibc"[11].

Questa permette la virtualizzazione delle chiamate alla *glibc* da parte di un processo, permettendo di cambiare il loro comportamento. Il meccanismo di virtualizzazione viene inizializzato, all'interno di una shell, mediante il processo di preloading della PureLibc:

```
export LD_PRELOAD=/usr/lib/libpurelibc.so
```

[10]http://wiki.virtualsquare.org/index.php/Main_Page
[11]http://wiki.virtualsquare.org/index.php/PureLibc

Eseguito tale comando, tutte le chiamate alla libreria *glibc* fatte dai processi eseguiti nella shell vengono reindirizzate alle funzioni sviluppate nella PureLibc. È certamente un meccanismo potente.

L'attuale implementazione fornisce una funzione `getpid()` senza il sistema di caching, eliminando di fatto il problema alla radice.

Va sottolineato che il preloading della PureLibc è necessario in fase di *checkpoint*, mentre risulta inutile in quella di *restart*. La motivazione è abbastanza ovvia: la PureLibc risulta utile in fase di *checkpoint* perché evita il caching del PID del processo obiettivo. Una volta riesumato, non sarà più necessario evitare tale problema perché alla prima chiamata `getpid()` verrà eseguita la vera syscall e, in seguito, popolata la cache con il giusto PID.

A differenza di quanto il lettore si aspetterebbe, una soluzione basata sul preloading di librerie dinamiche non è del tutto ottimale. Dal punto di vista tecnico, una soluzione di questo tipo non risolve il problema del caching, lo aggira o lo evita. Infatti, non agisce direttamente sulla cache ma fa in modo che, durante la fase di *checkpoint*, questa non sia mai modificata.

In particolare, vengono reindirizzate soltanto quelle chiamate a funzione per cui, nel caso specifico, la PureLibc presenta una implementazione alternativa. Le altre continuano ad essere fornite dalla normale *glibc*. Risulta chiaro che la libreria precaricata debba provvedere a tutte quelle funzioni che necessitano, internamente, di una chiamata a `getpid()` perché, altrimenti, non eseguirebbero correttamente[12]. Ne è un esempio la funzione `raise()`.

Inoltre, questa soluzione presenta la stessa limitazione di quella basata sull'*hack*, dato che il preloading non ha effetti sui programmi compilati staticamente.

Oltretutto, si ricorda che il progetto CryoPID non prevede l'utilizzo della variabile d'ambiente `LD_PRELOAD` per adempiere al proprio compito. Questo è il motivo, ancora più importante del precedente, a causa del quale una soluzione di questo genere non è stata presa, effettivamente, in considerazione.

[12] Questo ulteriore problema non si presenta con l'*hack* , presente nella seconda "patch", perché agisce direttamente sulla cache.

3.7 Proof-of-concept: “byword”

Al termine del lavoro svolto, si è scritto un piccolo programma che dimostrasse i vantaggi della versione di CryoPID fornita di *hack* rispetto a quella precedente.

Si è implementato uno scenario client-server, in cui un server spedisce detti “geek” al client richiedente (cfr. figura 3.11). Quello scritto è stato inserito tra i programmi di test del progetto CryoPID.

Di seguito viene descritto il funzionamento, rispettivamente, del server e quello di un tipico client.

Server. Il comportamento tipico del server, scritto in “`sbyword.c`”, è quello di analizzare, innanzitutto, le frasi contenute nel file di detti passatogli dall’utente da riga di comando. In seguito, il server crea/apre la “server fifo”, `/tmp/wkfifo`, il cui compito principale è quello di mantenere le richieste provenienti dai diversi client.

A questo punto, il server esegue un ciclo infinito di esecuzione nel quale legge le richieste dei client e cerca di soddisfarle.

Tipicamente, ogni client spedisce sulla “server fifo” il proprio PID. Alla lettura di quest’ultimo, il server crea/apre una “custom fifo”, `/tmp/cfifo<client_pid>`, da condividere solo con lo specifico client e, in seguito, spedisce a questi un segnale `SIGUSR1` (più precisamente, il segnale viene spedito al processo il cui PID è stato letto dalla “server fifo”) per informarlo dell’imminente trasmissione. Infine, il detto viene scritto nella fifo appena creata/aperta, dopo essere stato scelto in modo casuale fra quelli disponibili.

Client. Il comportamento tipico di un client, scritto in “`cbyword.c`”, è quello di richiedere il servizio eseguito dal server mediante scrittura del proprio PID sulla fifo `/tmp/wkfifo`. Dopodiché, si pone in attesa della segnalazione `SIGUSR1` (il gestore è stato installato in precedenza) da parte del server.

Quando questa arriva, il client apre la “custom fifo” appena creata/aperta dal server, legge il detto speditogli ed, infine, lo stampa a video.

Il lettore avrà sicuramente compreso la finalità di tale programma di test. In

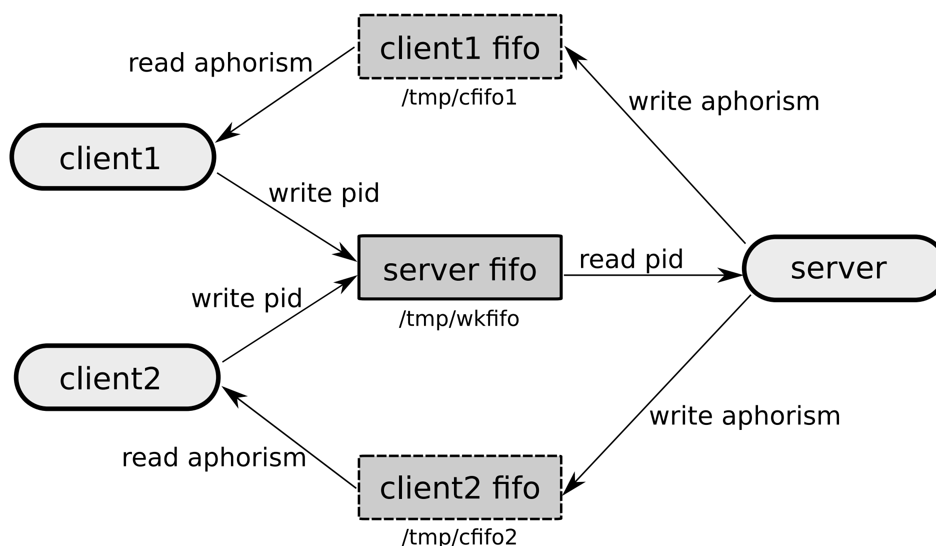

Figura 3.11: “byword”, schema di funzionamento

generale, sono i client che debbono essere “congelati” e, in seguito, ripristinati per veficare il corretto funzionameto dell’*hack* implementato.

Nello specifico, senza l’aggiornamento della cache, il processo riesumato scriverebbe nella “server fifo” il PID errato e, conseguentemente, non riceverebbe mai la segnalazione `SIGUSR1` da parte del server. Infatti, il segnale andrebbe perso perché spedito ad un processo non più in esecuzione.

È facile comprendere come, con l’aggiornamento della cache, tutto questo non accada. Scrivendo il giusto PID nella “server fifo”, il normale ciclo di funzionamento riprende senza alcun problema, come se nulla fosse accaduto.

Capitolo 4

Sviluppi futuri

In questo capitolo viene presentata la “roadmap” riguardante il progetto CryoPID. Le novità interessano sia nuove caratteristiche che migliorie al codice sorgente, anche alla luce delle limitazioni presentate in 2.8.

Aggiornamento dietlibc. Attualmente, il progetto CryoPID utilizza la penultima release della dietlibc. Numerose sono le migliorie apportate nell’ultima versione della libreria, tanto che la sua adozione è la prima questione che verrà affrontata, ma soltanto dopo aver consegnato questo lavoro di tesi.

Aggiornamento del supporto a x86-64. Tutto il lavoro finora presentato si riferisce alla versione x86 di CryoPID. Come anticipato in precedenza, diverse sono le architetture supportate (alcune in maniera minima). Sicuramente la versione x86_64 del progetto ha una notevole importanza per la diffusione di tale architettura sul mercato.

Ad oggi, però, il suo sviluppo è fermo e quello rilasciato è un codice non aggiornato, spesso non funzionante.

Implementazione del “dotfile”. Una delle prime idee scaturite durante lo studio di CryoPID è quella di realizzare un “dotfile” per il progetto, ovvero un file di configurazione delle opzioni utilizzabili. In generale, quest’ultimo aiuterebbe non poco l’utente nell’impiego e nella gestione della fase di *checkpoint*.

Autoconf. Il progetto è dotato di un normale Makefile come aiuto al processo di compilazione. Finora è risultato sufficiente, ma con il passare delle versioni del kernel e della *glibc* si necessita di uno strumento più potente e flessibile, quale autoconf[1]. In generale, questo permette di generare degli script che configurano automaticamente il software per il sistema obiettivo.

Sarà, sicuramente, una delle prime novità che verranno introdotte nella prossima versione di CryoPID.

Documentazione. Ad oggi, la documentazione distribuita insieme al codice sorgente del processo è insufficiente, per non dire assente. In generale, fornire una buona documentazione è, chiaramente, uno degli aspetti più importanti nella crescita globale di un progetto perché permette a nuovi sviluppatori di parteciparvi attivamente, integrandosi in tempi brevi.

Si è riscontrato, infatti, che molti ammiratori del progetto CryoPID non partecipano con proprie modifiche ed idee perché non in grado di comprendere, dal solo codice sorgente, il suo funzionamento basilare. Il problema legato alla documentazione dovrà essere risolto il prima possibile. Questo lavoro di tesi può essere considerato un buon inizio.

Di seguito vengono riportate le attuali problematiche o limitazioni sulle quali il team di sviluppo concentrerà i maggiori sforzi nel lungo termine:

- Supporto ai socket di rete.
- Supporto alle applicazioni costituite da due o più processi.
- Ripristino dei registri floating-point.
- Soluzione alla randomizzazione della memoria.
- Corretto repristino del nome del processo in `/proc/`*`pid`*`/cmdline`.
- Possibilità di memorizzare il contenuto dei file regolari all'interno dell'immagine.

[1] http://www.gnu.org/software/autoconf/

Capitolo 5

Conclusioni

Questo lavoro di tesi ha analizzato la possibilità di eseguire il *checkpoint/restart* di processi in ambiente Linux, cioè la capacità di memorizzare in un file l'intero stato di un processo per poi poterlo riattivare, dal punto in cui lo si era sospeso, in un secondo momento.

Inizialmente si è cercato di studiarne lo stato dell'arte, cercando di comprendere le problematiche e gli scenari di utilizzo. In seguito, sono stati analizzati i maggiori progetti CRS ("Checkpoint/Restart Systems") riassumendone le diverse soluzioni adottate per fronteggiare i problemi dovuti alla memorizzazione e al ripristino dello stato delle numerose risorse possedute da un processo. Questo ha permesso di cogliere, in linea generale, l'approccio al problema.

Infine, si è preso in considerazione il progetto CryoPID. Si è, prima di tutto, cercato di capirne il funzionamento globale per poi analizzarne i dettagli implementativi attraverso lo studio del codice sorgente scritto nel linguaggio C.

La fase di studio e di testing del progetto ha portato alla luce alcuni problemi di CryoPID derivanti dall'utilizzo del PID caching da parte della libreria *glibc*. La soluzione proposta in questa dissertazione è un *hack*, una sorta di "trucco" che permette a CryoPID di scavalcarne i limiti ed eseguire correttamente il proprio lavoro.

Il lavoro svolto sul progetto ha permesso di entrare in contatto con una comunità di sviluppatori brillanti e ricchi di idee. Attualmente, il sottoscritto ne fa parte attivamente, tanto che le due "patch", presentate in questo lavoro di tesi, sono state

accolte immediatamente.

Si può affermare, con relativa sicurezza, che il progetto CryoPID è ancora troppo “acerbo” per poter essere considerato in uno scenario produttivo; troppo importanti sono ancora le limitazioni presenti al suo interno.

Piace, però, concludere con una citazione che ha spronato e ispirato continuamente a credere in questo lavoro di tesi:

> *Non sarà ancora quella bella soluzione lineare da utilizzare in ambienti produttivi e critici, però, se non si studiano le soluzioni acerbe (che tentano di risolvere problemi "maturi"), non si innova...*
>
> – Un hacker “vecchio stampo”

Elenco delle figure

2.1 Output del comando “cat /proc/self/maps” 26
2.2 Struttura dati “cp_vma” (cpimage.h) 28
2.3 Struttura dati “cp_fd” (cpimage.h) 30
2.4 Strutture dati per la gestione dei segnali (cpimage.h) 32
2.5 Struttura dati principale (cpimage.h) 34
2.6 Codice di riconoscimento dell’istruzione *int0x80* 36
2.7 Layout del file immagine in memoria principale 39
2.8 Funzioni `main()` in CryoPID 40
2.9 Layout di un processo in Linux (architettura x86) 43
2.10 Richieste (in byte) e rispettivi puntatori alle VMA allocate 44
2.11 Codice implementante le operazioni 2, 5 e 6 46
2.12 Struttura dati “cp_socket_unix” (cpimage.h) 51
2.13 Struttura dati “cp_file” (cpimage.h) 51
2.14 Struttura dati “cp_fifo” (cpimage.h) 52
2.15 Funzione `jump_to_trampoline()` (stub.h) 53
2.16 Struttura dati “ele_area” (common.c) 55

3.1 File di test “sigtest.c” 61
3.2 Frammento di implementazione della `getpid()` 64
3.3 Funzione per l’azzeramento della cache 65
3.4 Funzione `getpid()` in codice macchina e assembly 66
3.5 Elemento della tabella dei simboli 68
3.6 Verifica ed estrazione della *signature* (`getpid_hack_write.c`) . . 70
3.7 Struttura dati “cp_getpid” (cpimage.h) 70
3.8 Layout del file immagine in memoria principale, con *hack* 71

3.9 Principale passaggio nell'implementazione dell'*hack* 72
3.10 Funzione `jump_to_getpid_hack()` (stub.h) 73
3.11 "byword", schema di funzionamento 77

Bibliografia

[1] Oren Laadan and Jason Nieh. *"Transparent Checkpoint/Restart of Multiple Processes on Commodity Operating Systems."*
In *Proceedings of the 2007 USENIX Annual Technical Conference (USENIX 2007).* Santa Clara, CA, June 2007.

[2] Michael Rieker, Jason Ansel, and Gene Cooperman. *"Transparent User-Level Checkpointing for the Native POSIX Thread Library for Linux."*
In *The 2006 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'06).* Las Vegas, NV. Jun, 2006.

[3] Jason Ansel, Kapil Arya, and Gene Cooperman. *"DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop."*
In *23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'09).* Rome, Italy. May, 2009.

[4] Victor C. Zandy. *ckpt - Process Checkpoint Library.*
http://pages.cs.wisc.edu/~zandy/ckpt

[5] AA. VV. *Condor - High Throughput Computing.*
http://www.cs.wisc.edu/condor/

[6] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. *"Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System."*
http://www.cs.wisc.edu/condor/doc/ckpt97.pdf, 1997.

[7] Jason Duell, Paul Hargrove, Eric Roman. *"Requirements for Linux Checkpoint/Restart."*
In *Berkeley Lab Technical Report (publication LBNL-49659)*, May 2002.

[8] Eric Roman. *"A Survey of Checkpoint/Restart Implementations."*
In *Berkeley Lab Technical Report (publication LBNL-54942)*, July 2002.

[9] Jason Duell, Paul Hargrove, Eric Roman. *"The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart."*
In *Berkeley Lab Technical Report (publication LBNL-54941)*, December 2002.

[10] Paul Hargrove and Jason Duell. *"Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters."*
In *Proceedings of SciDAC 2006.* June 2006.

[11] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. *"The Design and Implementation of Zap: A System for Migrating Computing Environments."*
In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002).* Boston, MA, December 9-11, 2002, pp. 361-376.

[12] Hua Zhong and Jason Nieh. *"CRAK: Linux Checkpoint / Restart As a Kernel Module"*.
In *Technical Report CUCS-014-01, Department of Computer Science, Columbia University.* November 2001.

[13] Sukadev Bhattiprolu, Eric W. Biederman, Serge Hallyn, Daniel Lezcano. *"Virtual servers and checkpoint/restart in mainstream Linux."*
http://www.mnis.fr/fr/services/virtualisation/pdf/cr.pdf

[14] Renzo Davoli et al. *Virtual Square*. http://www.virtualsquare.org

[15] Renzo Davoli et al. *PureLibc*. http://wiki.virtualsquare.org/index.php/PureLibc

[16] AA.VV. *Wikipedia*. http://en.wikipedia.org

Ringraziamenti

Il primo pensiero va alla mia famiglia che mi ha sempre sostenuto, anche se spesso rimproverato per essere uno “sfigato” (grazie *Picchio*). Lo prendo come un complimento.

Un grazie speciale lo dedico ad Alessia, con la quale condivido da anni i momenti più felici ed intensi all’infuori dello studio. Si è ormai “arresa” al mio amore per questa materia e lo studio in generale; spero che abbia compreso quanto quello che provo per lei sia più importante e profondo.

È sempre stata presente durante i momenti più duri di questo percorso universitario ed è stupendo pensare di avere accanto una persona che ti sappia comprendere al volo e crede ciecamente in te. La gran parte della mia felicità la devo a lei.

Ringrazio i miei “vecchi” amici, quelli delle colline marchigiane, con i quali ho condiviso tutto fino a pochi anni fa e con i quali ormai, a causa del pendolarismo settimanale, condivido soltanto pochi attimi del mio tempo libero. Anche loro (Tommaso, Valentino, Luca e tutti gli altri) hanno rinunciato nel volermi fare uscire spesso il sabato sera.

Lo studio a Bologna ha portato nuovi compagni di avventure, amici con i quali condividere le ore sul computer e il suo affascinante mondo. Un grazie particolare va a Tiziano, detto *Tappo*, per i momenti di studio e disperazione passati su VDE, la condivisione di un intero percorso di studi e le accese discussioni politiche fatte in chat. Proprio a lui devo il motivo di questo lavoro di tesi.

Un grazie al “compagno” Rudi, con il quale condivido le idee politiche e non solo. Devo a lui i consigli che mi hanno permesso l’ingresso al Collegio Superiore.

Ringrazio il gruppo de “i pazzi di Reggio” (*Uovo*, *Bigo*, *Bedo, Mone* e gli altri) che mi hanno sempre aiutato, sin dagli inizi, nel configurare Debian sulla

mia macchina e con i quali condivido una smodata passione per le follie che si possono creare con un computer.

Come non ricordare anche tutti quelli che frequentano #osd, in particolare i personaggi dal sapere irraggiungibile quali il Professore Renzo Davoli (*reenzo*), *shammash*, *godog* e il “mitico” *garden*, che sono per me un costante stimolo allo studio e alla ricerca.

Un pensiero lo rivolgo alla mia “nuova” famiglia a Bologna, con la quale ho trascorso momenti davvero felici. Un grazie particolare va a *Gigio*, ormai un fratello aggiunto, con cui condivido da tempo la passione per l’informatica e grazie al quale ho potuto affrontare serenamente un primo anno di “collegio” a dir poco singolare.

Infine, ringrazio zio Adriano, eterno burlone, per il grande aiuto offerto nella correzione di questa tesi e tutti quelli che non ho citato, ma che probabilmente ringrazierò a voce facendo finta di ricordarmi chi sono.