TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

DIPARTIMENTO DI INGEGNERIA CIVILE E INGEGNERIA INFORMATICA
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Meccanismo d'autorizzazione per accedere ad oggetti critici del kernel basato sul percorso d'esecuzione

Candidato:
Simone Tiberi
(Mat. 0299908)

Relatori:
Prof. **Francesco Quaglia**
Prof. **Alessandro Pellegrini**

ANNO ACCADEMICO 2021/22

Alla dolcezza e saggezza dei miei nonni.

A nonno Enzo, ovunque tu sia, ora sono un po' di più "il tuo scienziato".

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei listati

Capitolo 1

Introduzione

Al giorno d'oggi il tema della sicurezza è sempre più centrale nella concezione e nello sviluppo di soluzioni IT. Questo aspetto è senza dubbio collegato a due fenomeni: l'aumento esponenziale del numero di calcolatori in esercizio e l'influenza del mondo informatico nell'ambito economico-politico. Quest'ultima in particolare, induce quotidianamente gruppi di persone (i.e. i cracker) a specializzarsi nello sviluppo di malware, ovvero prodotti software che agiscono contro l'interesse dell'utente [1], spesso a scopo di lucro. Per questo motivo, nel corso degli anni, sono state sviluppate contromisure sempre più raffinate per far fronte a malware sempre più pericolosi ed elaborati.

Una tipologia *storica* d'attacco è il buffer overflow, documentato già nel 1972 dal Computer Security Technology Planning Study [2] e realizzato per la prima volta nel 1988 dall'hacker statunitense Robert Morris [3]. Negli anni questa tipologia d'offensiva è stata raffinata e declinata in diverse varianti ed allo stesso tempo sono state studiate tecniche di difesa sempre più raffinate, quali ad esempio ASLR (Address Space Layout Randomization) e *stack canaries*.

Trentacinque anni dopo la pubblicazione del primo articolo relativo agli attacchi di tipo buffer overflow citato poc'anzi, Hovav Shacham *et al.* [4] hanno introdotto alla comunità scientifica ROP (Return Oriented Programming), una nuova variante di estrema efficacia. Questa si basa sul riuso di codice già presente all'interno dell'address space del binario target, a differenza delle altre che richiedono il caricamento esplicito sullo stack dei byte necessari all'exploit. Questo particolare rende la tecnica estremamente efficace e sotto opportune ipotesi Turing completa [5].

Tale efficacia è ancor più preoccupante se il target di questa tipologia d'attacchi diventa il kernel di un sistema operativo. Questo perché la probabilità di individuare all'interno della sezione testo i byte necessari per costruire l'exploit è direttamente proporzionale alla dimensione del binario ed inoltre l'efficacia della contromisura più efficace, ovvero ASLR, si riduce notevolmente [6].

Questo lavoro di tesi si cala nel contesto delle contromisure ad attacchi basati su ROP a livello kernel. In particolare si pone l'obiettivo di fornire un meccanismo di autorizzazione all'esecuzione di specifiche funzioni critiche sulla base della validazione del percorso d'esecuzione pregresso.

Nel capitolo 2, si propone una disamina del funzionamento di ROP e di alcuni esempi di contromisure proposte in letteratura scientifica. Nel capitolo 3, viene fornita un'analisi del processo di caricamento dei moduli kernel in Linux, in quanto necessario per comprendere alcune scelte implementative per la realizzazione dell'architettura di protezione. Nel capitolo 4 è riportata la descrizione completa del lavoro svolto e nel capitolo 5 una valutazione funzionale e prestazionale. Infine, nel capitolo 6, vengono mostrati alcuni spunti per futuri sviluppi e/o migliorie.

Capitolo 2

Return Oriented Programming

ROP è una tecnica di exploit introdotta nel 2007 da Hovav Shacham *et al.* [4] nell'ambito degli attacchi di tipo buffer overflow. L'idea alla base del suo funzionamento è quella di sfruttare i cosiddetti gadget, ovvero sequenze di byte già presenti all'interno della sezione testo del binario, per far eseguire istruzioni macchina arbitrarie al processore.

Un gadget tipicamente si presenta in una forma simile a quanto riportato nel listato 2.1, ovvero come una qualsivoglia istruzione macchina seguita da una `ret`. Quest'ultima, assieme alla manipolazione dello stack frame da parte dell'attaccante, permette la realizzazione di execution flow arbitrari motivo per cui, in [5], la tecnica è stata mostrata essere Turing completa, assumendo di avere a disposizione un set di gadget adeguato all'interno del binario.

</> Listato 2.1: Esempio di gadget

```
pop %rbx    ; byte 0x5b
ret         ; byte 0xc3
```

È opportuno osservare come il kernel sia vulnerabile *by design* ad attacchi basati su ROP in quanto:

- ha una notevole estensione della sezione testo, il che favorisce la ricerca di un set adeguato di gadget;
- rimane in esecuzione per intervalli temporali relativamente lunghi, se paragonati a quelli delle applicazioni che vi eseguono al di sopra. Quest'ultimo aspetto è

vantaggioso in quanto permette all'attaccante di bypassare la contromisura KASLR semplicemente procedendo per *trials & errors*.

Inoltre la natura CISC (Complex Instruction Set Computer) dell'architettura di riferimento (i.e. Intel x86_64 [7]) rende ancor più facile reperire gadget utilizzabili, in quanto non vengono posti limiti sull'allineamento in memoria delle istruzioni da eseguire[1]. Ad esempio, nel lavoro di Shacham *et al.* [4], viene mostrato come la sequenza di byte:

```
f7 c7 07 00 00 00 0f 95 45 c3
```

nel caso in cui venisse eseguita interamente a partire dal primo byte, avrebbe una semantica pari a quanto riportato a sinistra nel box 2.2. Viceversa se eseguita a partire dal secondo byte (i.e. `0xc7`) avrebbe quella riportata a destra.

</> Listato 2.2: Interpretazione della sequenza di byte `<f7 c7 […] 45 c3>`

A partire dal primo byte

```
test $0x00000007, %edi
setnzb -61(%ebp)
```

A partire dal terzo byte

```
movl $0x0f000000, (%edi)
xchg %ebp, %eax
inc %ebp
ret
```

2.1 | Building block per ROP

Nella sezione precedente è stato sottolineato come in [4] la tecnica ROP sia stata mostrata essere Turing completa. Ciò implica che un qualunque program flow può essere ricodificato attraverso un opportuno set di gadget. L'obiettivo di questa sezione è quello di analizzare e descrivere nel dettaglio la *codifica* di elementi classici della logica di programmazione (e.g. assegnazione di valori, salti condizionati o meno) in ambito ROP.

[1] Cosa ad esempio presente in architetture RISC come ARM.

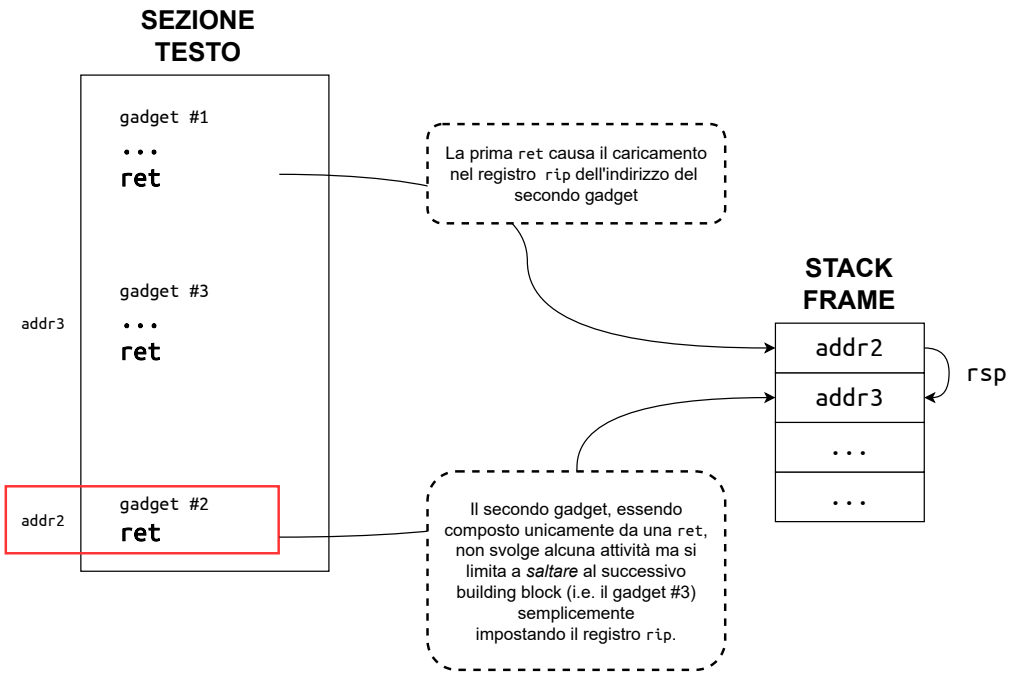

Figura 2.1: *NOP in ROP*

2.1.1 | NOP in ambito ROP

L'istruzione `nop`, il cui unico obiettivo è quello di far avanzare il program counter, può essere realizzata in ambito ROP, semplicemente caricando sullo stack l'indirizzo di una istruzione `ret`. In questo modo è possibile realizzare il flusso mostrato in figura 2.1.

È opportuno sottolineare come questo building block non sia del tutto equivalente ad un'istruzione `NOP`, in quanto oltre all'aggiornamento del program counter causa una modifica dello stack pointer, la quale potrebbe essere *visibile* ad esempio un gestore di interruzione innestato lungo l'esercizio del thread che sta eseguendo secondo una regola *return-oriented*.

2.1.2 | Caricamento di una costante in ambito ROP

Un altro elemento fondamentale per la costruzione di program flow è il caricamento di costanti all'interno di un registro. Si consideri ad esempio l'istruzione:

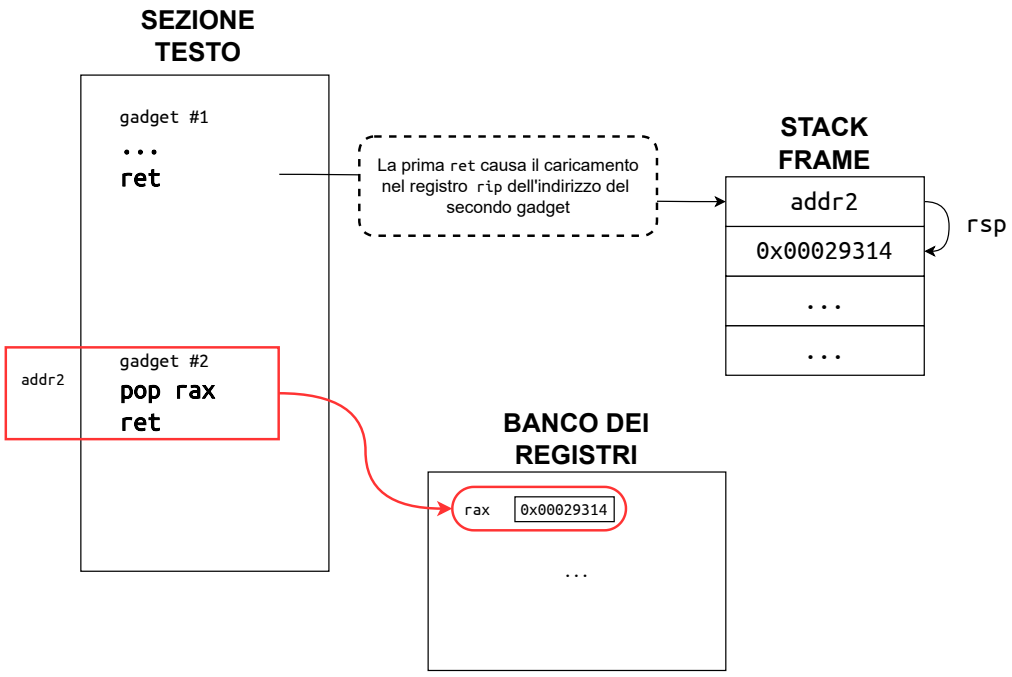

Figura 2.2: *Caricamento di una costante all'interno di un registro in ROP*

```
mov $0x00029314, %rax
```

è inverosimile sperare di trovare gadget che la includano precisamente all'interno del binario, in quanto esistono 2^{64} costanti differenti per sistemi a 64 bit. Per questo motivo, in ambito ROP è possibile emulare l'istruzione in questione caricando il valore necessario sulla cima dello stack e sfruttando un gadget come quello riportato nel listato 2.3. In questo modo è possibile realizzare il flusso mostrato in figura 2.2.

</> Listato 2.3: Gadget per il caricamento di costanti

```
pop %rax    ; byte 0x58
ret         ; byte 0xc3
```

Per quanto riguarda le operazioni in memoria, siano esse in lettura o scrittura, è possibile sfruttare il meccanismo descritto poc'anzi per il caricamento di costanti all'interno di registri al fine di installare il corretto indirizzo lineare da esprimere per poi sfruttare gadget contenenti le operazioni desiderate (e.g. `mov 8(%rbx), %rcx`). In questi casi occorre tener conto dello spiazzamento specificato nel gadget per caricare l'indirizzo corretto: se

ad esempio si volesse leggere una `qword` dall'indirizzo `0xdeadbeef` e caricarla nel registro `rcx`, sfruttando il gadget citato come esempio bisognerebbe caricare in `rbx` l'indirizzo:

```
0xdeadbeef - 8 = 0xdeadbee7
```

2.1.3 | Salti in ambito ROP

Nei programmi *tradizionali*, un salto non condizionato consiste semplicemente nella modifica del contenuto del registro `rip` in funzione della posizione in memoria dell'istruzione successiva desiderata. Un'implementazione equivalente di questa logica in ambito ROP passa per la modifica dello stack pointer. In particolare, si consideri uno scenario in cui sullo stack vengono caricati gli indirizzi di n gadget al fine di realizzare uno specifico program flow. Nel caso in cui ad un certo punto del flusso si volesse ripartire ad eseguire dall'inizio si potrebbe replicare due volte il medesimo setup dello stack frame oppure adottare la strategia mostrata in figura 2.3 in cui:

- si carica sullo stack frame il puntatore alla entry dello stesso in cui è presente l'indirizzo del gadget da cui si vuole ripartire ad eseguire
- si sfrutta un gadget per caricare tale indirizzo nel registro `rsp` ed emulare il salto attraverso una `ret`.

Il discorso diventa più complicato nell'ambito dei salti condizionati, in quanto le varie istruzioni esistenti (e.g. `je`, `jz`, …) causano una modifica del registro `%rip`, mentre in ambito ROP è necessario manipolare lo stack pointer. Nel listato 2.4 è riportato un possibile modo di costruire un set di gadget a tal proposito ispirato a quello presente in [8] il quale è descritto nell'elenco numerato seguente:

1. l'istruzione a riga 1 è funzione di quale tipologia di salto condizionato si vuole realizzare. Nel caso specifico di questo esempio il saltare o meno si basa sul confronto tra il contenuto del registro `%eax` con il valore `1`;

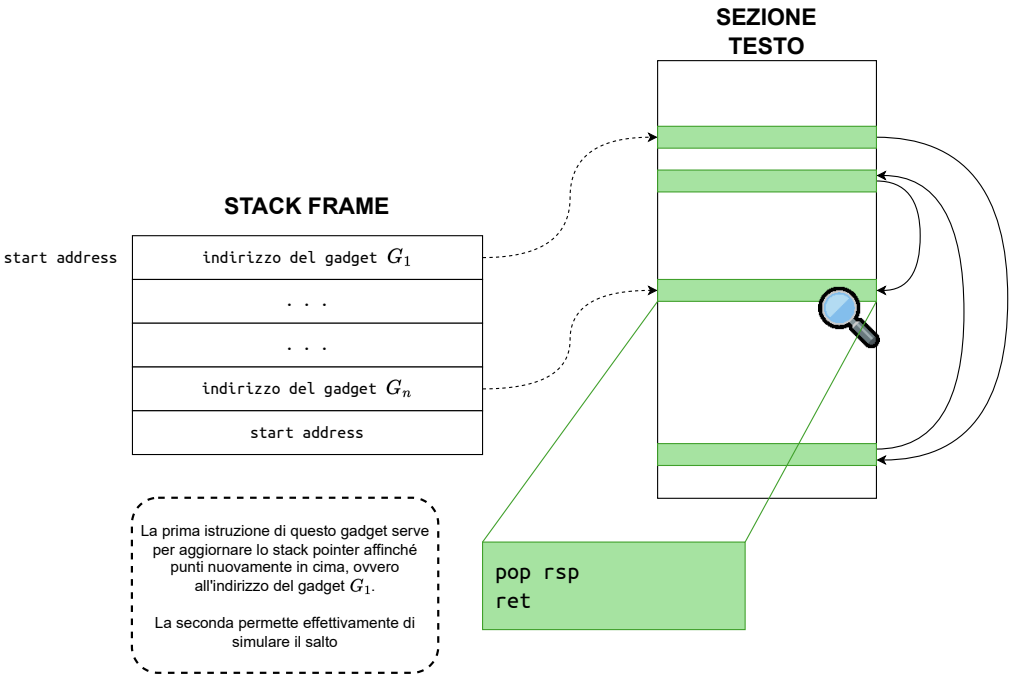

Figura 2.3: *Salto non condizionato in ROP*

2. attraverso gli statement 2 e 3, il `CF` (Carry Flag) viene copiato all'interno di un registro general purpose (i.e. `%eax`) al fine di poterlo utilizzare per impostare correttamente il valore puntato dallo stack pointer;

3. tramite le istruzioni a righe 4-9 si memorizzano in un'array d'appoggio composto da due entry (e.g. un'area riservata sullo stack frame configurato prima dell'attacco) gli indirizzi dei due possibili target del salto;

4. attraverso l'ultima `pop` a riga 10, si recupera nuovamente l'indirizzo di base dell'array d'appoggio di `dword`;

5. eseguendo l'istruzione a riga 11 quattro o otto volte, rispettivamente per architetture a 32 o 64 bit, si ottiene l'indirizzo dell'entry corretta, relativamente all'esito del confronto memorizzato in `%eax`

6. infine si carica l'indirizzo computato all'interno del registro `%esp` e si esegue la `ret` per saltare.

</> Listato 2.4: Salto condizionale basato sullo stack pointer

```
1  cmp $1, %eax
2  mov $0, %eax
3  adc %eax, %eax
4  pop %ecx
5  pop %ebx
6  mov %ebx, (%ecx)
7  add $4, %ecx
8  pop %ebx
9  mov %ebx, (%ecx)
10 pop %ecx,
11 add %eax, %ecx
12 mov (%ecx), %esp
13 ret
```

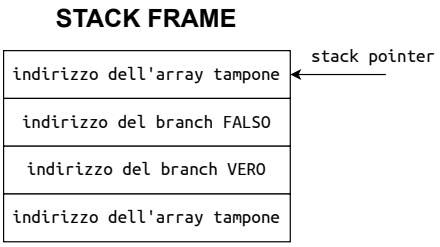

Figura 2.4: *Stack frame da prepare per utilizzare il codice in 2.4*

2.1.4 | Ulteriori considerazioni

Nel paper originale ci si sofferma su alcuni aspetti critici del set di gadget presenti all'interno della `libc` utilizzati per portare avanti attacchi ROP user space. Non è possibile effettuare un'analisi esaustiva e completa in ambito kernel in quanto la superficie di codice binario utilizzabile non è *stabile* poiché vari sottosistemi possono essere compilati o meno nell'immagine del kernel. Di conseguenza qualsiasi risultato ottenuto in termini di set di gadget a disposizione sarebbe limitato ed instabile.

2.2 | Control Flow Integrity

Uno degli elementi di maggiore criticità nell'ambito degli attacchi basati su ROP sono i function pointer. Un'attaccante ha infatti una duplice possibilità di redirezionare il flusso d'esecuzione in modo arbitrario: può infatti modificare l'indirizzo di memoria da chiamare (*forward edge*) o l'indirizzo di ritorno al chiamante (*backward edge*).

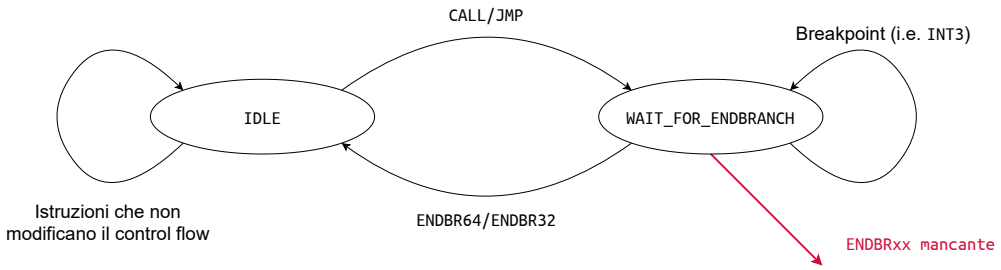

Figura 2.5: *Automa a stati finiti caratterizzante del meccanismo Intel CET [12]*

Il compilatore `Clang LLVM` fornisce un supporto alla CFI (Control Flow Integrity) a livello di compilazione abilitabile tramite flag `-fsanitize=cfi` [9]. Come descritto da Cook nella Linux Conf AU 2020 [10], in ambito *forward edge* la protezione offerta si basa sulla validazione a *call time* della *classe* (i.e. segnatura) di function pointer. In questo ambito il supporto hardware x86 (i.e. Intel CET) è carente rispetto a quanto offre ARM con BTI (Branch Target Identification) e pointer authentication in quanto permette di discriminare un'unica classe di target ammissibili ed inoltre è stato mostrato essere male implementato [11].

Il meccanismo di Intel CET [12] si basa sull'adozione di una specifica istruzione macchina, l'`ENDBR64`[2], per *marcare* i target validi di operazioni di `CALL/JMP`. I byte scelti da Intel per caratterizzare quest'istruzione sono gli stessi che in passato venivano utilizzati per esprimere una `NOP`, in modo tale da rendere i binari compilati compatibili anche su hardware non predisposto al supporto hardware per la CFI. L'istruzione `ENDBR` non ha alcun effetto diretto sullo stato del thread in esercizio, ma viene utilizzata dal processore per aggiornare lo stato di un automa *ad-hoc* descritto nella documentazione ufficiale [12] e riportato per completezza in figura 2.5. In riferimento a quanto descritto nello schema, è interessante osservare come il meccanismo realizzato da Intel sia compatibile con meccanismi, come ad esmepio `kprobe`, basati su `INT3`.

Affinché la validazione a livello software del *forward edge* sia realizzabile è necessario sfruttare la LTO (Link Time Optimization) di `clang`. Questo implica che i file `.o` non

[2] In modo equivalente in sistemi a 32 bit si adotta la `ENDBR32`

sono più semplici file oggetto bensì dei LLVM IR (Intermediate Representation) da gestire attraverso gli script di build propri di LLVM (i.e. l'utilty `bfd` - Binary File Descriptor di `binutils` non è più utilizzabile). Il controllo si basa sull'aggregazione di funzioni afferenti alla stessa classe all'interno di *jump table* dedicate e su controlli effettuati prima di accedervi.

La protezione del *backward edge* si basa sulla gestione dei return address memorizzati sullo stack. In quest'ambito soluzioni *hardware based* sono nettamente migliori (e.g. il meccanismo di protezione in x86 è stato dismesso in quanto lento e problematico rispetto a possibili race conditions). Intel realizza la validazione dell'indirizzo di ritorno affidandosi ad uno SCS (Shadow Call Stack) [12] in cui:

1. a seguito di una `call`, si carica una copia dell'indirizzo di ritorno;
2. all'atto di una `ret`, si preleva quanto caricato in precedenza all'atto della `call` e si confronta con l'indirizzo di ritorno memorizzato nel *main stack*, generando un'eccezione in caso di mancata corrispondenza.

2.3 | RAP: RIP ROP

L'utilizzo della LTO per aggregare le funzioni all'interno di *jump table* e aggiungere una serie di controlli all'accesso impatta in modo considerevole sulla latenza d'esecuzione. Per questo motivo il PaX team, autore nel 2001 del meccanismo ASLR, ha introdotto in letteratura un'alternativa [13] con l'obiettivo di aumentare la protezione sia sul *forward* che sul *backward edge*, tramite rispettivamente Indirect Control Transfer Protection e RAP (Return Address Protection).

Il primo si basa sulla gestione di un CFG (Control Flow Graph) approssimato, come nel caso dell'implementazione di CFI offerta da LLVM in modo nativo. L'approssimazione è legata alla possibilità di avere *false edges* nel grafo, ovvero rami in cui ricadono funzioni

logicamente afferenti a classi distinte, ma che a causa della segnatura vengono aggregate insieme. Questo tipo di problematica viene in parte mitigata attraverso approcci come la rimozione di call indirette e la diversificazione dei tipi ove possibile.

Il vero vantaggio in ambito *forward edge* introdotto dal lavoro del PaX team è un approccio scalabile rispetto alla LTO per la gestione dell'accesso. Questo si basa sullo sfruttamento di funzioni hash, calcolate in modo *language dependent*, su parti specifiche della segnatura e sulla verifica del matching tra la funzione target e la dereferenziazione del puntatore[3]. Nel listato 2.5 è riportato un'esempio di instrumentazione estratto dalla presentazione ufficiale, in cui:

- le costanti numeriche (i.e. `0x11223344` `0x55667788`), oggetto delle operazioni di confronto (i.e. istruzione `cmpq`), sono i valori hash computati sui puntatori;
- le `qword` dichiarate (i.e. istruzione `dq`) in testa alla funzione target func, sono gli hash ammissibili per permetterne l'esecuzione.

</> Listato 2.5: Esempio del PaX Team (*forward edge*)

```
1  cmpq $0x11223344,-8(%rax)
2  jne .error
3  call *%rax
4  ;[...]
5  cmpq $0x55667788,-16(%rax)
6  jne .error
7  call *%rax
8  ;[...]
9  dq 0x55667788,0x11223344
10 func:
```

In ambito *backward edge* il contributo del PaX team è RAP che si basa sulla cifratura degli indirizzi di ritorno. In particolare, nel listato 2.6 tratto dal paper ufficiale, è riportata

[3]Nella presentazione originale si cita che questo tipo di validazione può essere sfruttata anche in ambito backward per i return address, ma in realtà in tale ambito la vera innovazione risiede nella cifratura degli indirizzi di ritorno descritta in seguito.

l'instrumentazione necessaria per la realizzazione della patch, descritta per completezza di seguito:

1. nel prologo, si memorizza nel registro `rbx`, precedentemente pre-salvato, l'indirizzo di ritorno (i.e. `8(%rsp)`) cifrato, utilizzando una chiave mantenuta in un registro apposito (i.e. `r12` su architetture `amd64`);

2. nell'epilogo, si recupera l'indirizzo di ritorno in chiaro dallo stack e lo si confronta con quello decifrato, ottenuto computando `rbx` $\oplus$ `r12`. In caso di successo si procede con il ritorno al chiamante, altrimenti si causa un fault in modo sincrono invocando la `ud2`.

</> Listato 2.6: Esempio del PaX Team (*backward edge*)

```
1      push %rbx
2      mov 8(%rsp),%rbx
3      xor %r12,%rbx
4      ;[...]
5      xor %r12,%rbx
6      cmp %rbx,8(%rsp)
7      jnz .error
8      pop %rbx
9      retn
10 .error:
11     ud2
```

Le alternative alla CFI tradizionale proposte dal PaX team, seppur più performanti, hanno due drawback principali:

- per quanto riguarda RAP, la chiave è vulnerabile in lettura. Tuttavia con ASLR attivo l'attaccante deve *leakare* il return address sia in versione PT che in versione CT;

- Indirect Control Transfer Protection non è utilizzabile in caso di *executable-only memory*, ossia memoria non leggibile ma solo eseguibile la quale, stando all'articolo sopra citato di `lwn.net`, è in procinto di essere inserita in mainline.

2.4 | DROP THE ROP

Uno dei problemi dell' approccio CFI applicato alla codebase del kernel è la granularità con cui si discrimina l'afferenza ad una classe (i.e. approssimazione del CFG); infatti prototipi del tipo:

```
void f(void);   int g(void *);
```

sono estremamente frequenti. Per questo motivo Moreira *et al.* hanno introdotto un nuovo meccanismo [14] in cui si fa uso di un sistema *tag-based*, sempre basato su plugin a tempo di compilazione, per individuare quali indirizzi sono legittimi per una *indirect call*.

Per realizzare il controllo sul *forward edge* occorre instrumentare il codice come mostrato nel listato 2.7 e descritto di seguito:

- in testa al corpo della funzione (box in alto) viene aggiunta una `nopl` che semanticamente non aggiunge alcuna informazione ma permette di specificare i byte di *tag* in memoria da sfruttare in fase di validazione;
- supponendo di aver memorizzato nel registro `rax` l'indirizzo della funzione da invocare, nel box in basso viene mostrato come controllare che il target sia lecito. In particolare si confronta il tag atteso con quanto puntato da `rax` scostandosi di 4 byte per non prendere in considerazione l'opcode della `nopl`; in caso di successo si procede con la call indiretta, altrimenti si invoca un gestore del fault.

Per realizzare invece il controllo sul *backward edge* occorre instrumentare il codice come mostrato nel listato 2.8 e descritto di seguito:

</> Listato 2.7: Instrumentazione per il forward edge (Moreira *et al.*)

Prologo della funzione

```
1 <func>:
2     nopl 0xbcbee9
```

Controllo precedente alla *indirect call*

```
1 cmpl $0xbcbee9,0x4(%rax)
2 je <6>
3 push %rax
4 callq <kcfi_vhndl>
5 pop %rax
6 callq *%rax
7 nopl 0x138395f
```

- subito dopo un'istruzione di `call` viene inserita una `nopl` in cui si specifica la tag, come nel caso precedente;
- nell'epilogo della funzione invocata:
  1. si carica in un registro quanto presente sulla testa dello stack (i.e. l'indirizzo di ritorno);
  2. si confronta la tag attesa con il contenuto del registro caricato al passo precedente spiazzandosi di 4 byte per non considerare l'opcode come al passo precedente;
  3. in caso di successo si procede con la return, viceversa con l'invocazione sincrona del gestore di fault.

2.5 | Adelie

Adelie [6] è un meccanismo di difesa per il kernel Linux proposto da Nikolaev *et al.* che si basa sulla randomizzazione continua del layout dell'address space relativamente all'area

</> Listato 2.8: Instrumentazione per il backward edge (Moreira *et al.*)

Tag inserita subito dopo la `call`

```
1 callq 0xffffff810001eb <func>
2 nopl 0x138395f
```

Controllo precedente alla `ret[q]`

```
1 mov (%rsp),%rdx
2 cmpl $0x138395f,0x4(%rdx)
3 je <7>
4 push %rdx
5 callq <kcfi_vhndl>
6 pop %rdx
7 retq
```

in cui risiedono i moduli, in quanto essi sono tipicamente meno testati rispetto al core e dunque tendenzialmente più vulnerabili.

Affinché sia possibile randomizzare parte dell'address space kernel, è necessario che il codice sia compilato per essere PIC (Position Indipendent Code). A livello utente tipicamente vengono compilate in questo modo le librerie dinamiche (i.e. `.so` in Linux, `.dll` in Windows) affinché possano eseguire correttamente indipendentemente dal loro indirizzo di base di caricamento. A tal proposito si sfrutta in modo massivo la GOT (Global Offset Table) in cui vengono memorizzati gli indirizzi da risolvere, funzione del posizionamento del codice della libreria. In Adelie, la sezione testo originale viene suddivisa in due porzioni distinte:

- la `.text` che può essere continuamente spostata;
- la `.fixed.text` contenente alcuni elementi fondamentali per permettere l'interazione tra la parte *movable* ed il resto del kernel

e si utilizzano quattro GOT per permettere il collegamento tra le due parti:

- nella parte *unmovable*, una contenente i riferimenti alla sezione `.data` ed un'altra utilizzata per per collegarsi alla parte *movable*, la cui posizione è continuamente randomizzata;
- nella parte *movable*, una contenente i riferimenti alla sezione `.rodata`, considerata *unmovable* e a quella `.text` del kernel ed una locale per riferire i simboli interni alla regione in continuo spostamento.

La gestione della randomizzazione, dell'allocazione e del popolamento delle GOT è affidata ad un kernel daemon ad hoc: il *"re-randomizer"*.

Per effettuare questa randomizzazione in modo *safe* rispetto all'esecuzione concorrente di altri task nel sistema si racchiude ciascuna porzione *movable* all'interno di un blocco delimitato da `mr_start` e `mr_end` e si adotta [15] come schema per la gestione dell'*unmapping* della memoria una volta migrato il buffer.

Non è sufficiente randomizzare il codice per innalzare la sicurezza del sistema rispetto al threat model considerato, bensì è necessario randomizzare anche le stack area motivo per cui viene introdotta una pila (LIFO) *lock-free* di stack disponibili *per-CPU*. In definitiva dunque, per ciascuna funzione originale f, se ne costruiscono due nuove g ed h. La prima, posizionata all'interno della sezione `.fixed.text`, è un wrapper in cui ci si procura una nuova area di stack e si invoca il corpo originale h, collocato invece in `.text`, all'interno di `mr_start|end`. Per ultimo il meccanismo di sicurezza adotta anche una cifratura dei return address simile a quanto proposto da [14] e analizzato nella sezione 2.3.

Capitolo 3

Inserimento di un modulo kernel

Il processo di caricamento dei moduli Linux è di particolare interesse nell'ambito del meccanismo di protezione oggetto di questo lavoro di tesi, in quanto sfruttando delle modifiche mirate è stato possibile integrare al suo interno la logica dedicata alla registrazione dei simboli critici da proteggere. Pertanto, al fine di favorire la comprensione di quanto analizzato in sezione 4.6, nelle sezioni seguenti è riportata una disamina delle varie fasi del processo.

3.1 | System call d'accesso

A livello utente, per inserire un modulo M all'interno del kernel Linux, è possibile sfruttare due system call: `init_module` o `finit_module`. Poiché a fini della patch esse sono del tutto equivalenti[1], di seguito per semplicità si analizza soltanto la prima.

L'intero processo di caricamento si basa sull'utilizzo di due strutture dati: `module`, la quale viene utilizzata per rappresentare logicamente il modulo in tutto il suo *ciclo di vita* e `load_info`, utile soltanto nel montaggio per contenere alcuni metadati d'interesse (e.g. indirizzi delle tabelle principali contenute nell'header ELF).

Per prima cosa, all'interno della `init_module`, viene invocata la `may_init_module` in cui si verifica che il thread corrente abbia le capability per effettuare il montaggio e che i moduli siano abilitati. Successivamente, tramite la `copy_module_from_user`:

[1] `finit_module`, a differenza di `init_module`, legge l'ELF da caricare a partire da un file descriptor. Questo è utile per sistemi in cui si vuole permettere il caricamento di un modulo soltanto se questo è memorizzato in uno storage *sicuro* (e.g. read-only `rootfs`) [16, `modutils/modutils.c`, riga 194].

1. si verifica se la lunghezza specificata dall'utente sia non inferiore a quella dell'header di un ELF;
2. si invoca, se presente, l'hook di pre-caricamento dell'LSM;
3. si alloca la memoria adibita a contenere il binario;
4. si copia l'ELF, un chunk alla volta per evitare di *monopolizzare* la CPU;
5. si invoca, se presente, l'hook di post caricamento dell'LSM.

Infine, viene invocata la `load_module` analizzata nella sezione successiva.

3.2 | Funzione load_module

La funzione `load_module` è il *cuore* dell'inserimento del LKM. L'insieme delle attività che vengono svolte al suo interno possono essere logicamente suddivise in *fasi*. La prima è quella dei task preliminari, in cui:

- si controlla la firma del modulo, tramite la `module_sig_check`;
- si effettuano dei validity check sul formato dell'ELF ricevuto dal livello utente, attraverso la `elf_validity_check`;
- si popolano i campi della struttura `load_info`, invocando la `setup_load_info`;
- si verifica che il modulo non appartenga all'elenco di quelli proibiti, sfruttando la funzione *blacklisted*;
- si imposta il campo `sh_addr` per ciascuna section header per evitare ogni volta l'accesso tramite offset e si abbassa il flag di allocazione (i.e. `SHF_ALLOC`) per le sezioni `.modinfo` e `__versions`, grazie alla `rewrite_section_headers`;

- infine, nella `check_modstruct_version`, si effettua un controllo ulteriore in funzione dell'abilitazione del *"module versioning support"*.

La seconda fase è quella dedicata all'allocazione e alla configurazione dell'area di memoria in cui installare il modulo M. Per prima cosa si invoca la `layout_and_allocate` in cui:

1. tramite la `check_modinfo` si imposta la licenza e si effettuano alcuni controlli relativi alle informazioni contenute nella sezione `modinfo` (i.e. sul `vermagic`, sulla possibilità di *contaminazione* nel caso in cui il modulo sia *out-of-tree*, *staging* e/o *live patching* ed infine sulla configurazione delle *retpoline*);

2. si invoca la `module_frob_arch_sections`, una funzione *architecture-specific* non definita per `x86_64`;

3. attraverso la `module_enforce_rwx_sections`, si verifica che nessuna sezione abbia impostato contemporaneamente il flag di scrittura (`SHF_WRITE`) e di esecuzione (`SHF_EXECINSTR`);

4. si abbassa il flag (`SHF_ALLOC`) per `.data..percpu` e si alza `SHF_RO_AFTER_INIT` per `.data..ro_after_init` e `__jump_table`;

5. con la `layout_sections`, si determinano le dimensioni totali delle varie parti del binario (i.e. `text`, `ro`, ...) e si impostano i campi `sh_entsize` per le varie sezioni. In questa fase le operazioni vengono svolte in maniera agnostica rispetto all'architettura target, eventuali ottimizzazioni vengono applicate in seguito;

6. sfruttando la `layout_symtab`, si correggono le dimensioni computate al passo precedente per tenere conto dello spazio necessario a memorizzare la tabella dei simboli *core*;

7. infine invocando la `move_module`, sia per il `core_layout` che per l'`init_layout`:

Stato	Descrizione
`MODULE_STATE_UNFORMED`	Stato iniziale per un modulo. Come suggerisce il nome, il caricamento è ancora incompleto (e.g. non è stata ancora attuata la rilocazione).
`MODULE_STATE_COMING`	Stato in cui il modulo M risiede quando il suo caricamento è quasi terminato. `kallsyms` *vede* le funzioni definite in M, per cui è ad esempio possibile agganciare `k[ret]probe` alle funzioni in esso contenute.
`MODULE_STATE_LIVE`	Stato *normale*, ovvero quello in cui il modulo transita non appena è terminato il suo caricamento e vi rimane fintantoché non viene richiesta la rimozione.
`MODULE_STATE_GOING`	Stato in cui il modulo transita non appena ne viene richiesta la rimozione.

Tabella 3.1: *Possibili stati che può assumere un modulo*

(a) si effettua l'allocazione in modo *architecture-specific*;

(b) si marca il pointer restituito come *non leak*, in quanto puntato da un campo della struttura `module`;

(c) si trasferiscono tutte le sezioni con flag `SHF_ALLOC` all'interno del buffer allocato per l'installazione, aggiornando di volta in volta i campi `sh_addr`;

Lo stato di un modulo, durante il suo ciclo di vita, viene codificato attraverso un apposito enumerato (i.e. `module_state`) che può assumere 4 valori distinti riassunti in tabella 3.1. Tramite la funzione `add_unformed_module`, si inizializza lo stato di M e si inserisce la struttura `module` ad esso associata all'interno della lista RCU e dell'albero RB, nel caso in cui M non sia già presente.

La seconda fase termina con una serie di ulteriori attività di inizializzazione, alcune opzionali (e.g. il controllo sulla firma) altre mandatorie (e.g. inizializzazione di alcuni campi della struttura `module`, risoluzione dei simboli del kernel).

Ai fini della patch che è stata realizzata, è interessante soffermarsi su quanto svolto all'interno della funzione `find_module_sections`, presente in questa *zona* del function call graph. Ogni qual volta si aggiunge un servizio opzionale all'interno del kernel che necessita d'interagire con il sottosistema dei moduli, tipicamente si sfruttano sezioni apposite dell'ELF per memorizzare i metadati d'interesse e reperirli in fase di caricamento, aggiungendo statement alla `find_module_sections`. Per comprendere a fondo il meccanismo, di seguito è stata presa in considerazione l'interazione fra il sottosistema `kprobe` e questa fase di caricamento dei moduli kernel.

`kprobe` è un sottosistema del kernel Linux che permette di invocare specifiche routine macchina prima e dopo l'esecuzione della quasi totalità delle istruzioni presenti nel testo [17]; in poche parole è un sistema di *hot-patching* del kernel. Per impedire di agganciare specifiche procedure, `kprobe` fonisce due API distinte:

- la macro `__kprobes`, da aggiungere alla segnatura, ha l'effetto di posizionare il corpo della funzione all'interno della sezione `.kprobes.text`;
- la macro `NOKPROBE_SYMBOL(fnanme)`, da apporre in calce alla funzione in modo analogo alla `EXPORT_SYMBOL`, ha l'effetto di aggiungere l'indirizzo di `fname` all'interno della sezione `_kprobe_blacklist`.

In entrambi i casi, l'obiettivo è quello di limitare la visibilità del sottosistema `kprobe`.

Ogni qual volta si inserisce un nuovo modulo all'interno del kernel, è possibile che la lista dei simboli *proibiti* debba essere modificata. A tal proposito, in `find_module_sections`, è presente un blocco di codice adibito a questo scopo (listato 3.1) in cui si caricano:

- l'indirizzo della sezione `.kprobes.text` nel campo `kprobes_text_start`;
- l'indirizzo dell'array di simboli inseriti nella blacklist, memorizzato nella sezione `_kprobe_blacklist`, nel campo `kprobe_blacklist`.

In questo modo, tramite un meccanismo di `callback`, al cambiamento di stato del modulo (i.e. `UNFORMED` → `COMING`) si aggiungono i simboli all'interno della blacklist globale.

</> Listato 3.1: Estratto della `kernel/module.c#find_module_sections`

```
#ifdef CONFIG_KPROBES
    mod→kprobes_text_start = section_objs(info, ".kprobes.text", 1,
                                          &mod→kprobes_text_size);
    mod→kprobe_blacklist = section_objs(info, "_kprobe_blacklist",
                                        sizeof(unsigned long),
                                        &mod→num_kprobe_blacklist);
#endif
```

La fase successiva è dedicata al completamento del caricamento del modulo, in particolare per prima cosa si invoca la `apply_relocations` per rilocarlo. Seguono poi una serie di invocazioni a funzioni responsabili a finalizzare il caricamento sotto vari aspetti:

- con la `post_relocation`:
  - si ordina la tabella delle eccezioni, reperite in precedenza dall'ELF con il meccanismo analizzato della `find_module_section`;
  - si copiano le aree *per-cpu* una volta rilocate;
  - si configurano i metadati necessari in seguito al sottosistema `kallsyms`;
  - si attuano delle ottimizzazioni *architecture specific*;
- con la `flush_module_icache`, si libera la *instruction cache* del modulo dopo averla sfruttata in modo intensivo;
- infine, tramite `cfi_init`, `init_build_id` e `dynamic_debug_setup`, si inizializzano rispetettivamente: la CFI (sezione 2.2), alcuni aspetti relativi alla *stack trace* ed i metadati necessari al sottosistema `ddebug`.

Come ultima cosa prima di completare la formazione del modulo, viene inizializzato il sottosistema `ftrace`. Il funzionamento di quest'ultimo si basa sul sostituire, ai primi 5

byte di ciascuna sezione eseguibile, l'invocazione di una propria funzione adibita al monitoraggio. Per evitare di sovrascrivere byte importanti per altre istruzioni, `Kbuild`, in fase di compilazione, inserisce una `CALL` ad una funzione particolare (i.e. `__fentry__`) la quale non fa altro che restituire immediatamente il controllo al chiamante. Questa è a tutti gli effetti un'area di 5 byte che il compilatore riserva affinché in seguito possa venire rimpiazzata dalla `CALL` ad una funzione *significativa*. Poiché il sottosistema `ftrace` tiene traccia dei punti in cui instrumenta il codice, nel realizzare la patch è stato necessario tener conto della sua presenza ed evitare il sollevamento di warning.

La funzione `complete_formation`, come suggerisce il nome, è responsabile di completare la formazione del modulo che si sta installando. Al suo interno, in sezione critica:

- si verifica che i simboli esportati dal modulo non abbiano duplicati;
- si aggiungono i bug presenti nella sezione `__bug_table` alla lista globale;
- si abilitano in modo opportuno i bit di `RO`, `NX` ed `X` per le pagine in cui è stato installato il modulo;
- si muta lo stato del modulo in `MODULE_STATE_COMING`.

Prima di invocare l'ultima funzione della catena di montaggio (i.e. la `do_init_module`, sez. 3.3) vengono svolte le ultime attività di inizializzazione:

- nella `prepare_coming_module`, se presenti, si applicano le *live patch* e si avvia la *notification chain* citata in precedenza nella digressione sulla relazione fra i sottosistemi `kprobe` e `module`;
- si effettua il parsing degli argomenti ricevuti da linea di comando, tramite la funzione `parse_args`;
- si inizializza il sosttosistema `sysfs` tramite `mod_sysfs_setup`,

- nel caso in cui il modulo è *live patching*, si memorizza l'header dell'ELF per sfruttarlo in seguito tramite la `copy_module_elf`, altrimenti si procede a liberare l'area in cui era stato copiato in principio il `.ko` dall'utente, tramite la `free_copy`.

Ai fini della patch realizzata, ha senso soffermarsi sul meccanismo di gestione dei *module parameters*. All'interno della funzione `find_module_sections`, citata in precedenza, per prima cosa si reperisce l'array di strutture `kernel_param` presenti all'interno dell'ELF per effetto della direttiva `__module_param_call`[2] inserita all'interno dei listati C. Analizzando la funzione `parse_args`, è facile comprendere come gli argomenti passati a linea di comando all'atto dell'inserimento sotto forma di `key=value`, ad esempio:

```
insmod mymodule.ko myarg=value
```

vengano utilizzati per inizializzare tali parametri, invocando le operazioni specificate all'interno della struttura stessa nelle `kernel_param_ops`.

All'interno del file `kernel/params.c`, è inoltre presente la logica che permette di comprendere come i valori associati ai parametri vengano effettivamente memorizzati all'interno della struttura citata nel paragrafo precedente. In particolare, per ciascun tipo di parametro è necessario definire ed esportare gli handler per le operazioni `get` e `set` e la struttura `kernel_param_ops` che ne racchiude i puntatori. Al fine di minimizzare le linee di codice, nella maggior parte dei casi a tal proposito si sfrutta il template definito dalla macro `STANDARD_PARAM_DEF` (listato 3.2). In essa è facile dedurre quanto desiderato, ovvero che il valore effettivo del parametro viene memorizzato all'interno del campo `arg` della struttura `kernel_param`.

3.3 | Funzione do_init_module

La `do_init_module` è la funzione responsabile del completamento dell'installazione del nuovo modulo. Per prima cosa al suo interno, tramite la `do_mod_ctors`, vengono invocati

[2] Questa direttiva è presente nell'espansione delle macro `module_param*` presenti nel kernel.

</> Listato 3.2: Macro `STANDARD_PARAM_DEF`

```
1  #define STANDARD_PARAM_DEF(name, type, format, strtolfn)            \
2  int param_set_##name(const char *val, const struct kernel_param *kp) \
3  {                                                                   \
4      return strtolfn(val, 0, (type *)kp→arg);                        \
5  }                                                                   \
6  int param_get_##name(char *buffer, const struct kernel_param *kp)   \
7  {                                                                   \
8      return scnprintf(buffer, PAGE_SIZE, format "\n",                \
9              *((type *)kp→arg));                                     \
10 }                                                                   \
11 const struct kernel_param_ops param_ops_##name = {                  \
12     .set = param_set_##name,                                        \
13     .get = param_get_##name,                                        \
14 };                                                                  \
15 EXPORT_SYMBOL(param_set_##name);                                    \
16 EXPORT_SYMBOL(param_get_##name);                                    \
17 EXPORT_SYMBOL(param_ops_##name)
```

vari costruttori definiti all'interno del binario, reperiti tramite la `find_module_sections` all'interno della sezione `.ctors` oppure `.init_array`. Dopo di che, sfruttando il meccanismo delle *initcall* [18, Concepts/"The initcall mechanism"], si invoca la funzione di inizializzazione del modulo e si imposta il suo stato a `MODULE_STATE_LIVE` (tabella 3.1).

Successivamente vengono svolte alcune attività *secondarie*, non importanti ai fini della comprensione della patch, come ad esempio::

- avviare la catena di notifiche basata su meccanismo di *callback* a seguito del cambiamento di stato del modulo (i.e. `COMING` → `LIVE`);
- notificare all'utente, tramite `uevent` del sottosistema `sysfs`, l'avvenuto inserimento del modulo;
- attendere il completamento delle attività asincrone, nel caso in cui sia stato spe-

cificato di voler gestire “`async_probe`” [19] nella linea di comando, sfruttando la `async_syncronize_full`.

Capitolo 4

Path-Based Authorization

4.1 | Descrizione ad alto livello dell'architettura

PBA (Path-Based Authorization) è una patch di sicurezza per il kernel Linux che introduce la possibilità di monitorare l'esecuzione dei thread in esercizio in un sistema, al fine di decidere se concedergli o meno la possibilità di eseguire specifiche funzioni critiche. Come anticipato nel capitolo introduttivo, l'obiettivo è quello di evitare che un attaccante, ad esempio tramite ROP, possa eseguire una di queste procedure, bypassando eventuali controlli presenti in funzioni che, lungo un'esecuzione *legittima*, verrebbero attivate prima di esse (figura 4.1).

Il meccanismo di sicurezza ha i moduli kernel come target, motivo per cui in sezione 4.4 è analizzata nel dettaglio l'interfaccia fornita e le scelte implementative orientate all'utilizzabilità. Inoltre si basa:

- sull'interazione di due entità presenti nel sistema: i thread applicativi T_i ed i demoni di validazione D_j;
- sul concetto di *checkpoint* C_k definito, nel contesto di PBA, come una qualunque funzione facente parte di un percorso d'esecuzione valido per l'attivazione di un simbolo critico.

Ogni qual volta un thread applicativo T_i effettua una `CALL` ad un checkpoint C_k, in maniera automatica e trasparente allo sviluppatore, viene attivata una funzione definita nella patch sfruttando il sottosistema `kprobe`. Al suo interno si interagisce con uno dei demoni di

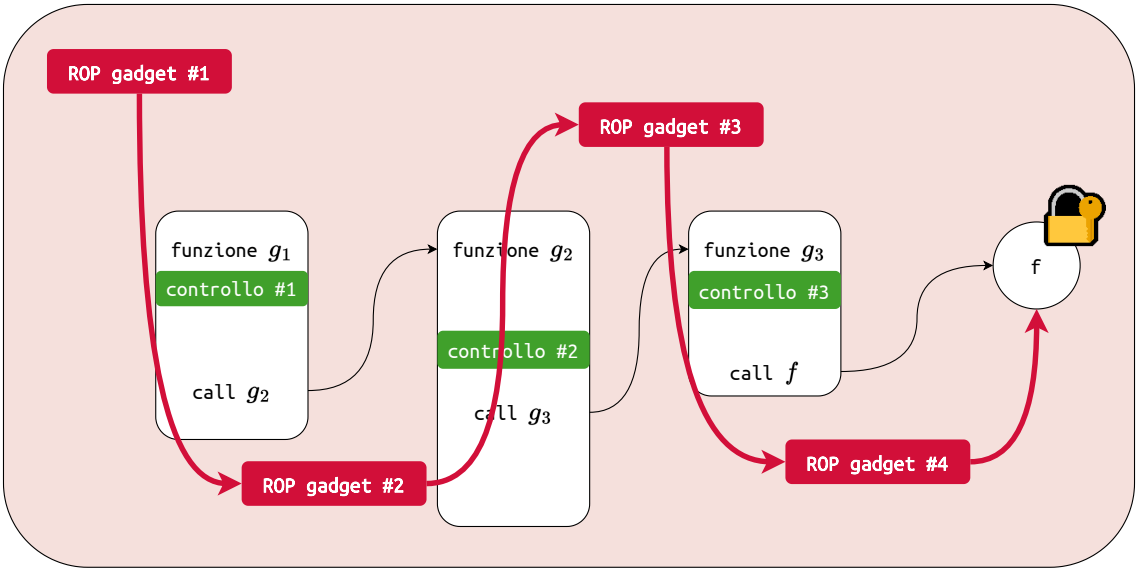

Figura 4.1: *Esempio d'attacco*

validazione D_j, il quale una volta attivato registra nelle proprie strutture dati l'evento:

$$E = \{\text{“Il thread } T_i \text{ è passato per } C_k\text{”}\}$$

Nel momento in cui il thread T_i tenta d'accedere al simbolo critico f, si sfrutta ancora una volta il sottosistema `kprobe` per attivare un altro gestore, che ha l'obiettivo di avviare nuovamente un'interazione con D_j. Il demone, una volta risvegliato, verifica che la lista ordinata dei checkpoint previsti per accedere a f (i.e. Σ) sia contenuta in quella degli attraversati da T_i (i.e. Σ_i) ed, in caso affermativo, concede l'accesso, altrimenti lo proibisce.

Per comprendere al meglio il meccanismo si consideri l'esempio seguente: siano

$$\begin{cases} \Sigma_1 = [C_1, C_3, C_4, C_9, C_2, C_5] \\ \Sigma_2 = [C_7, C_4, C_9, C_6, C_5] \\ \Sigma_3 = [C_4, C_6, C_1, C_{11}, C_4, C_5] \end{cases}$$

le liste di checkpoint collezionati rispettivamente da T_1, T_2 e T_3. Nel caso in cui la

sequenza di checkpoint attesi per f fosse $\Sigma = [C_1, C_4, C_5]$, il demone:

- concederebbe l'accesso a T_1;
- proibirebbe l'accesso a T_2 in quanto $C_1 \in \Sigma$, ma $C_1 \notin \Sigma_2$;
- proibirebbe l'accesso a T_3 in quanto C_4 precede C_1 in Σ_3 mentre lo segue in Σ.

È evidente che lo schema finora descritto non sia sufficiente ad innalzare il livello di sicurezza del sistema. Infatti se il *corpo* del simbolo critico f rimanesse nella sua posizione originale, un attaccante potrebbe semplicemente redirezionare il flusso ad `<f+5>` piuttosto che `<f>` per bypassare completamente il sottosistema `kprobe` ed eseguire in modo illegittimo la funzione critica. Per questo motivo, in fase di registrazione, le istruzioni macchina di f vengono migrate in una nuova area, il cui indirizzo è noto unicamente ai demoni. In definitiva, una volta completata la verifica di Σ_i rispetto a Σ, il demone D_j fornisce l'indirizzo valido a T_i per procedere con la sua esecuzione.

4.2 | Struttura della patch

La patch è stata sviluppata in riferimento alla versione 5.15.75 del kernel linux e supporta unicamente l'architettura x86 a 64 bit. Tuttavia, l'impatto sui file esistenti è limitato così come il numero delle porzioni di codice *architecture specific*, motivo per cui risulta essere facilmente portabile verso altre versioni e/o architetture.

La patch PBA interessa principalmente il sottosistema `security`, in cui è stato inserito un nuovo modulo omonimo. Poiché il funzionamento del meccanismo di sicurezza è invasivo nel processo di caricamento modulo (sezione 4.6) all'interno del file KBuild si è preferito utilizzare un booleano (i.e. `y|n`) come variabile di configurazione piuttosto che un tristate (i.e. `y|m|n`).

Per facilitare la comprensione di come sfruttare il servizio offerto è stata inoltre aggiunta una serie di esempi ad-hoc in `samples/pba/` descritti nella sotto sezione 4.13.2 e nel capitolo 5.

Infine l'interfaccia fornita agli sviluppatori per usufruire del servizio realizzato è stata completamente sintetizzata all'interno dell'header file `include/linux/pba.h`

Le modifiche alla codebase esistente sono limitate al file `kernel/module.c`, agli header `kprobes.h` e `module.h` contenuti nella directory di default (i.e. `include/linux/`) ed infine ai file di configurazione del sottosistema di sicurezza (i.e `KBuild` e `Makefile`), tutte descritte accuratamente nelle sezioni che seguono.

4.3 | Analisi delle strutture dati

Per comprendere al meglio quanto presentato nelle successive sezioni, è vantaggioso analizzare a priori le strutture dati necessarie al funzionamento della patch. Ciascuna di esse è stata *marcata* con la direttiva `__randomize_layout` per randomizzarne il layout al fine di aumentare l'efficacia del meccanismo di protezione come descritto in sezione 4.12.

4.3.1 | Struttura `chkp`

La struttura `chkp` (listato 4.1) permette l'astrazione del concetto di *checkpoint* all'interno del codice. Al suo interno sono presenti:

- il puntatore alla probe associata al checkpoint (`probe`);
- un contatore dei riferimenti (`ref_counter`) che:
  - viene incrementato, nel momento in cui si tenta di registrare nuovamente il medesimo checkpoint, evitando dunque di allocare più memoria di quella necessaria;

  - viene decrementato, nel momento in cui si tenta di rimuovere il checkpoint che è attualmente necessario ad altri simboli da proteggere, evitando dunque di rompere lo schema di protezione;

- la struttura che permette di collegare il checkpoint ad una hash list (`links`).

</> Listato 4.1: Struttura `chkp`

```
1 struct chkp {
2     struct kprobe *probe;
3     unsigned int ref_counter;
4     struct hlist_node links;
5 } __randomize_layout;
```

4.3.2 | Struttura `protected_symbol`

La struttura `protected_symbol` (listato 4.2) contiene l'insieme dei metadati necessari alla protezione di un simbolo critico nel kernel quali:

- il puntatore alla probe associata al simbolo da proteggere (`target_probe`);
- il vettore di puntatori a strutture `chkp` (`expected_chkps`), una per ciascun checkpoint previsto per il simbolo in questione;
- la dimensione del vettore precedente (`nr_checkpoints`);
- l'indirizzo in cui è stato migrato il codice del simbolo critico (`new_addr`);
- la dimensione del buffer in cui è stato migrato il simbolo critico (`size`);
- il nonce generato dal demone all'atto della registrazione del simbolo da utilizzare per validare le richieste di rimozione (`nonce`);
- la struttura che permette di collegare il simbolo all'interno di una hash list (`links`).

</> Listato 4.2: Struttura `protected_symbol`

```
1 struct protected_symbol {
2     struct kprobe *target_probe;
3     struct chkp **expected_chkps;
4     size_t nr_checkpoints;
5     unsigned long new_addr;
6     unsigned long size;
7     unsigned long nonce;
8     struct hlist_node links;
9 } __randomize_layout;
```

4.3.3 | Struttura `collected_chkp`

La struttura `collected_chkp` (listato 4.3) viene utilizzata per codificare l'evento:

$$E = \{\text{“Il thread } T_i \text{ è passato per } g_i\text{”}\}$$

citato nella sezione 4.1. In definitiva dunque rappresenta l'informazione mantenuta dal validatore relativa al passaggio del thread applicativo T_i attraverso uno specifico checkpoint. Al suo interno sono presenti:

- l'identificativo di T_i (`pid`);
- il nome del checkpoint (`symbol`);
- il blocco che permette di collegare il simbolo all'interno di una hash list (`links`).

4.3.4 | Struttura `validator_work`

La struttura `validator_work` (listato 4.4) viene utilizzata per raccogliere il set di informazioni necessarie per l'interazione tra thread richiedenti e demoni di validazione. È costituita dai seguenti campi:

</> Listato 4.3: Struttura `collected_chkp`

```
1 struct collected_chkp {
2     pid_t pid;
3     const char *symbol;
4     struct hlist_node links;
5 } __randomize_layout;
```

- la struttura necessaria per collegare la richiesta alla coda di quelle che il demone D_i deve ancora processare (`links`);
- il puntatore al TCB del richiedente (`requestor`) per permetterne il risveglio;
- una variabile atomica (`syncro`) per la sincronizzazione con il richiedente, nel caso in cui questo faccia attesa attiva.

e da altri legati al tipo di lavoro che si intende far svolgere al validatore, i quali vengono descritti nell'elenco puntato che segue:

- per registrare un nuovo simbolo sono necessari:
  - il nome (`target_symbol`);
  - la lista dei nomi dei checkpoint previsti (`expected_checkpoints`);
  - la lunghezza della lista precedente (`nr_checkpoints`);
  - l'indirizzo di memoria in cui è stato migrato il corpo del simbolo (`new_addr`);
  - la taglia del nuovo buffer allocato (`size`);
  - il valore di ritorno (i.e. `0`/`-E`) della funzione di registrazione eseguita dal validatore (`retval`):
  - il campo utilizzato dal validatore per restituire il nonce da utilizzare eventualmente in seguito al fine di interrompere la protezione (`result`).

- per interrompere la protezione di un simbolo sono necessari:
  - il nome (`target_symbol`);
  - il nonce da utilizzare per validare la richiesta (`nonce`);
  - il campo utilizzato dal validatore per segnalare eventuali tentativi d'attacco (`result`).
- per collezionare un checkpoint sono necessari:
  - il puntatore alla probe agganciata alla funzione (`kp`);
  - il campo utilizzato dal validatore per segnalare eventuali tentativi d'attacco (`result`).
- per validare una richiesta d'accesso ad un simbolo critico sono necessari:
  - il puntatore alla probe agganciata alla funzione (`kp`);
  - il campo in cui il validatore restituisce l'indirizzo in cui è stata migrata la funzione oppure 0 per segnalare un tentativo d'attacco (`result`).
- per rimuovere checkpoint raccolti da un thread in procinto di terminare la sua esecuzione, non sono necessarie informazioni ulteriori.

4.3.5 | Struttura `request_area`

La struttura `request_area` (listato 4.5) viene utilizzata per rappresentare il punto di contatto fra demoni di validazione e thread applicativi. In particolare al suo interno sono presenti:

- il puntatore al TCB del demone di validazione per poterci interagire (`validator`);
- la lista *lock-less* a cui ciascun thread applicativo T_i può accodare la propria richiesta di lavoro (`requests`).

</> Listato 4.4: Struttura `validator_work`

```
1  struct validator_work {
2      struct llist_node links;
3      struct task_struct *requestor;
4      atomic_t syncro;
5      int type;
6      unsigned long result;
7      union {
8          struct {
9              const char *symbol;
10             unsigned long nonce;
11         };
12         struct kprobe *kp;
13         struct {
14             const char *target_symbol;
15             const char **expected_checkpoints;
16             size_t nr_checkpoints;
17             unsigned long new_addr;
18             unsigned long size;
19             int retval;
20         };
21     };
22 } __randomize_layout;
```

4.3.6 | Struttura `val_worker_args`

La struttura `val_worker_args` (listato 4.6) rappresenta logicamente l'insieme delle informazioni comuni ai vari demoni validatori e necessarie al loro esercizio. Al suo interno sono presenti:

- un identificativo per discriminare il *ruolo* del demone (`index`);
- i puntatori alla tabelle hash dei checkpoint (`chkps_htable`) e dei simboli da proteggere gestiti dai demoni (`prot_symbs_htable`);

</> Listato 4.5: Struttura `request_area`

```
1 struct request_area {
2     struct task_struct *validator;
3     struct llist_head requests;
4 } __randomize_layout;
```

- i puntatori ai lock che servono a sincronizzare l'accesso alle tabelle citate sopra (`chkps_lock` e `prot_symbs_lock`);
- il puntatore ad un contatore atomico (`finish`), utile insieme al primo campo, per l'orchestrazione di un'ipotetica fase di rimozione del meccanismo di sicurezza, qualora venisse implementato come modulo *standalone* (capitolo 6).

</> Listato 4.6: Struttura `val_worker_args`

```
1 struct val_worker_args {
2     int index;
3     struct hlist_head *chkps_htable;
4     struct hlist_head *prot_symbs_htable;
5     struct mutex *chkps_lock;
6     struct mutex *prot_symbs_lock;
7     atomic_t *finish;
8 } __randomize_layout;
```

4.3.7 | Struttura `pba_decl_entry`

La struttura `pba_decl_entry` (listato 4.7) è sfruttata per la realizzazione dell'interfaccia d'uso della patch fornita agli sviluppatori dei moduli Linux. Al suo interno sono presenti:

- i nomi del simbolo da proteggere (`target`)
- la lista dei nomi dei checkpoint attesi (`chkps`);

- il nonce da utilizzare per validare la richiesta di interruzione di protezione (`nonce`).

Listato 4.7: Struttura `pba_decl_entry`

```
1 struct pba_decl_entry {
2     const char *target;
3     const char *chkps[CONFIG_MAX_CHKPS];
4     unsigned long nonce;
5 } __randomize_layout;
```

Come si evince dal listato 4.7, è possibile esprimere al più `CONFIG_MAX_CHKPS` per ciascun simbolo critico. Tale valore può essere configurato a *compile-time* attraverso un opportuno parametro di KBuild specificato all'interno del file `Kconfig` presente nella directory `security/pba/`.

4.3.8 | Struttura `pba_target`

La struttura `pba_target` viene sfruttata unicamente nella patch del file `kernel/module.c` ed il suo scopo è quello di permettere la costruzione di una lista di informazioni necessarie a migrare i vari simboli critici definiti all'interno del modulo che si sta per montare. Al suo interno dunque sono presenti:

- il nome del simbolo che si vuole proteggere (`name`);
- il puntatore al buffer in cui è stato migrato il corpo della funzione (`buff`);
- la taglia in byte del codice macchina migrato (`size`);
- la struttura che permette il collegamento delle informazioni all'interno di una lista dedicata (`links`).

</> Listato 4.8: Struttura `pba_target`

```
1 struct pba_target {
2     const char *name;
3     void *buff;
4     unsigned long size;
5     struct list_head links;
6 } __randomize_layout;
```

4.4 | Interfaccia d'uso

L'interfaccia fornita agli sviluppatori di moduli kernel per proteggere l'accesso a simboli critici definiti all'interno dei loro artefatti si basa sullo sfruttamento del sezionamento offerto dallo standard ELF. In particolare, al fine di identificare una funzione per cui si vuole specificare un percorso d'attivazione valido, è possibile utilizzare la macro `PBA(fn)`. Questa, come evidente dal listato 4.9, permette di collocare il corpo di `fn` all'interno della sezione "`__pba_target_<fn>`" sfruttando la direttiva `__section(section)`, la quale:

- è definita all'interno del file `include/linux/compiler_attributes.h`;
- espande nell'attributo `__attribute__((__section__(section)))`.

Questo accorgimento è alla base delle attività di migrazione dell'area critica descritte nel dettaglio in sezione 4.6.

</> Listato 4.9: Macro `PBA`

```
1 #define PBA(fn) __section("__pba_target_" #fn) fn
```

Per indicare invece il percorso d'attivazione valido per ciascun simbolo critico definito all'interno di un modulo, è necessario utilizzare diverse macro:

- `START_PBA_DECL` e `END_PBA_DECL` per delimitare la zona di dichiarazione dei simboli critici;
- `GRANT_ACCESS_TO` e `IF_PASSED_IN` per specificare gli execution path per ciascuna funzione da proteggere.

Per capire come combinare tra loro le direttive presentate, è utile prendere in considerazione l'estratto dell'esempio presente nella patch (i.e. `samples/pba/exposer.c`), riportato per comodità nel listato 4.10.

</> Listato 4.10: Macro per definire i percorsi d'attivazione validi

```
1 START_PBA_DECL
2     GRANT_ACCESS_TO("test_A", IF_PASSED_IN("test_B", "test_C"))
3 END_PBA_DECL
```

Anche questo set di macro in realtà, maschera lo sfruttamento delle sezioni ELF, in particolare nel listato 4.11 è mostrata la sua espansione. Il posizionamento di queste informazioni all'interno della `"__pba_decl"`, viene sfruttato nel processo di caricamento dei moduli descritto nel capitolo 3 ed in particolare nella `find_module_section` per agganciare la struttura `pba_decl_entry` alla `module`. Questo meccanismo è alla base dell'inizializzazione delle strutture dati necessarie alla protezione dei simboli critici descritta in sezione 4.6.

</> Listato 4.11: Espansione delle macro nel listato 4.10

```
1 static struct pba_decl_entry
2 __pba_decl__[] __used __section("__pba_decl") = {
3     { .target = "test_A", .chkps = {"test_B", "test_C"}, .nonce = 0UL}
4 };
```

All'interno della soluzione realizzata è stata anche introdotta la possibilità di specificare a *load-time*, per ciascuna funzione critica, il valore di una variabile locale (e.g. una chiave segreta). L'interfaccia fornita a tale scopo è sintetizzata nel listato 4.12, in cui:

- la macro `SECURE_VALUE(name)` serve a dichiarare, all'interno del corpo di una funzione critica, la variabile `name` la cui inizializzazione avviene a *load-time* sfruttando gli argomenti a linea di comando. Il modificatore `volatile` viene utilizzato per evitare che il compilatore attui ottimizzazioni indesiderate.

  In essa si fa uso della direttiva `PBA_SV_PLACEHOLDER` al fine di introdurre una sequenza *riconoscibile* nel binario, per i motivi descritti in sezione 4.6;

- la macro `DECLARE_SECURE_VALUE_FOR(fn)` viene utilizzata per dichiarare la volontà di utilizzare una variabile sicura nella funzione critica `fn`.

</> Listato 4.12: Interfaccia per la gestione di valori sicuri

```
1 #define PBA_SV_PLACEHOLDER (12415237193757133664UL)
2 #define SECURE_VALUE(name)                              \
3     volatile unsigned long name = PBA_SV_PLACEHOLDER
4 #define DECLARE_SECURE_VALUE_FOR(fn)                    \
5     unsigned long pba_sv_##fn = 0UL;                    \
6     module_param(pba_sv_##fn, ulong, S_IRUGO)
```

4.5 | Relazioni tra le entità

Per motivi prestazionali (i.e. per evitare di introdurre un *single point of failure*) l'architettura sviluppata prevede lo sfruttamento di un pool di demoni di validazione (i.e 64) creati allo startup.

Questo implica che i metadati necessari alla protezione vengono organizzati in strutture dati che possono essere comuni a tutti i validatori o private per il singolo. Per limitare la latenza sperimentata dai thread T_i, a livello progettuale è stato deciso di associarli lungo l'intero ciclo di vita ad uno ed un solo demone D_j, sfruttando una funzione hash del puntatore al TCB per il mapping.

È opportuno sottolineare come, per la tabella dei checkpoint collezionati, è stato necessario utilizzare una chiave diversa per l'indicizzazione. Questo perché ciascun thread T_i interagisce con il demone $j-$esimo, dove l'indice j è calcolato come funzione hash dell'indirizzo del TCB. Nel caso in cui il demone organizzasse le strutture `collected_chkp` rispetto al puntatore al TCB, la tabella hash si ridurrebbe ad una sola lista concatenata.

In definitiva dunque, in relazione alle strutture dati analizzate nella sezione precedente, un thread applicativo T_i nell'eseguire il gestore citato nella sezione 4.1 compie i seguenti passi:

1. individua la `request_area` ad esso associato sfruttando la funzione `hash_ptr` per mappare un puntatore arbitrario sull'intervallo $[0, \texttt{HASH\_TABLE\_BITS} - 1] \equiv [0, 63]$;
2. popola una struttura `validator_work` inserendo i metadati opportuni per il lavoro da richiedere;
3. aggancia tale struttura alla lista *lock-less* presente all'interno della `request_area`;
4. risveglia il demone D_j ed attende il completamento del task richiesto;
5. eventualmente esegue specifiche attività (i.e. invocare una `BUG_ON` o modificare il program counter) in funzione del valore assunto dal campo di output `result` contenuto nella struttura `validator_work`.

I demoni di validazione devono necessariamente condividere le strutture dati per la gestione della protezione dei simboli critici, mentre per via del mapping 1:1 tra T_i e D_j possono mantenere lo storico dei passaggi attraverso i checkpoint (i.e. le strutture `collected_chkp`) privatamente.

Per motivi prestazionali (i.e. per minimizzare i tempi d'accesso medi) è stato scelto di organizzare le varie informazioni all'interno di hash table con liste di trabocco. Queste sono realizzabili sfruttando alcune facility offerte dal kernel Linux, come descritto nel elenco seguente:

- per rappresentare la tabella è stato sfruttato un array di dimensione $2^{\texttt{HASH\_TABLE\_BITS}}$ di strutture `hlist_head`;
- per rappresentare le liste di trabocco si utilizza una catena di `hlist_node` inserite all'interno delle strutture descritte nella sezione precedente (i.e. `protected_symbol`, `chkp`, `collected_chkp`).

Inoltre, per limitare ulteriormente la latenza introdotta dalla patch, le hash list sono state realizzate sfruttando il paradigma RCU (Read Copy Update), il quale garantisce la possibilità di traversare le liste di trabocco in modo concorrente senza necessità d'utilizzo di lock. Questi ultimi vengono utilizzati unicamente in fase di modifica, quindi all'atto della registrazione (sezione 4.6) ed interruzione (sezione 4.9) della protezione per un simbolo per sincronizzare l'accesso dei vari demoni di validazione. Queste attività tuttavia vengono svolte di rado ed, in particolare la prima, tipicamente all'avvio del sistema, finestra temporale in cui l'aspetto prestazionale conta in maniera ridotta.

Per sintetizzare quanto presentato in questa sezione, in figura 4.2 è riportato uno *sguardo d'insieme* relativo all'organizzazione delle strutture dati gestite dai demoni e necessarie per le varie interazioni.

4.6 | Registrazione di un simbolo da proteggere

La registrazione di un nuovo simbolo da proteggere può essere scomposta in due attività principali: la migrazione dell'area di memoria contenente il codice macchina e l'allocazione delle strutture dati necessarie alla sua gestione.

Per portare avanti entrambe è stato necessario per prima cosa reperire, all'interno del processo di caricamento dei moduli, le informazioni relative alla protezione contenute nelle sezioni `__pba_target_*` (sezione 4.4). A tal proposito è stata modificata la funzione `find_module_sections`, descritta nel capitolo 3, aggiungendo il blocco di codice

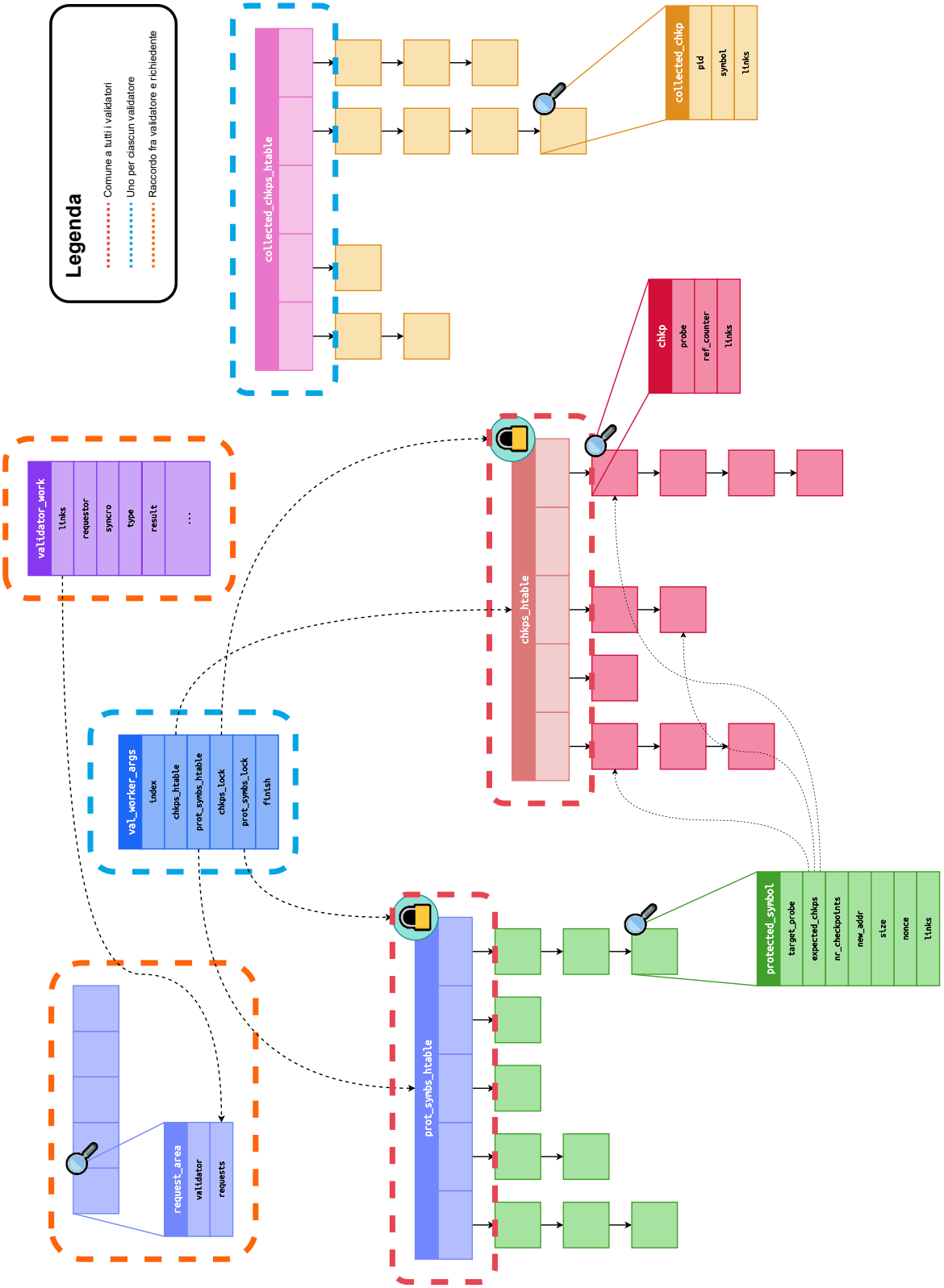

Figura 4.2: *Relazioni fra principali strutture dati*

ad-hoc riportato nel listato 4.13. Questo ha lo scopo di caricare l'indirizzo di base dell'array di strutture `pba_decl_entry` (listato 4.7) e la sua taglia rispettivamente nei campi `pba_entries` e `num_pba_entries` entrambi aggiunti alla struttura `module`.

</> Listato 4.13: Modifica della `find_module_sections`

```
1 #ifdef CONFIG_PBA
2     mod→pba_entries = section_objs(info, "__pba_decl",
3                                    sizeof(*mod→pba_entries),
4                                    &mod→num_pba_entries);
5 #endif
```

L'attività di migrazione inizia nella `allocate_pba_targets`, aggiunta alla `load_module` (sezione 3.2) subito dopo l'invocazione alla `post_relocation`. In essa si scorre la lista delle sezioni dell'ELF alla ricerca di quelle il cui nome inizia per `".rela__pba_target_"`.

Queste sezioni sono presenti all'interno del binario in quanto, essendo il modulo un oggetto rilocabile compilato per un'architettura x86 a 64 bit, il compilatore per ogni sezione testo ne genera automaticamente una di rilocazione il cui nome inizia per `.rela` [20]. Per ciascuna di esse si alloca una struttura `pba_target` (listato 4.8) e se ne popolano i campi. In particolare:

1. si reperisce l'indirizzo della sezione `"__pba_target_"` associata, sfruttando il campo `sh_info` del section header;
2. si alloca un nuovo buffer di taglia pari a $\min\{\texttt{PAGE\_SIZE}, s\}$, dove con s si indica il numero di byte di cui è composto il codice da migrare;
3. si copia il contenuto della sezione `"__pba_target_"` all'interno della nuova area di memoria;
4. si sovrascrive l'area originale con una serie di `0x90` (i.e. `NOP`) per rendere del tutto inoffensivi tentativi di bypass del meccanismo di sicurezza.

Un attaccante infatti potrebbe, tramite ROP, iniziare ad eseguire la funzione critica a 5 byte dal suo indirizzo di base eludendo del tutto il sottosistema `kprobe`. *Annullando* il buffer originale si rende inutile questo tipo di offensiva.

Il passo successivo alla migrazione del codice è la sua rilocazione, di cui è responsabile la `relocate_new_buffer`, la quale è un adattamento della `__apply_relocate_add` definita all'interno di `arch/x86/kernel/module.c`. Per prima cosa al suo interno si reperisce l'indirizzo dell'array (`rel`) di strutture `Elf64_Rela` (listato 4.14) memorizzato all'interno della sezione `".rela__pba_target_*"`. Dopo di che, per ciasucna entry:

1. si calcola l'indirizzo (`loc`) da rilocare come `base` (i.e. l'indirizzo dell'area di memoria allocata in precedenza) + `offset` (i.e. quello specificato in `rel[i].r_offset`);
2. si reperisce l'indirizzo (`sym`) della struttura `Elf64_Sym` (listato 4.14), ancora una volta come `base` (i.e. l'indirizzo della tabella dei simboli) + `offset` (i.e. i 32 bit più significativi della maschera `r_info` a 64);
3. si calcola il valore di rilocazione come `sym→st_value + rel[i].r_addend`;
4. a seconda del tipo di rilocazione, discriminato tramite gli 8 bit meno significativi del campo `r_info`, si procede a sovrascrivere il contenuto dell'indirizzo di memoria determinato in precedenza.

   È opportuno osservare che rispetto alla `__apply_relocate_add` non sono presenti controlli in quanto la patch è inserita dopo l'invocazione della `apply_relocations` ed eventuali *malformazioni* sarebbero già emerse.

Infine, la struttura `pba_target` viene inserita all'interno di una lista la cui testa è definita come variabile locale della `load_module`.

Gli elementi che popolano la lista di cui sopra, vengono sfruttati in una seconda funzione aggiunta nella patch alla `load_module`, ovvero la `setup_pba_for_module`, inserita successivamente alla `parse_args`. In essa, per ciascuna entry dell'array `pba_entries`:

</> Listato 4.14: Strutture `Elf64_*`

```
1  typedef struct elf64_rela {
2      Elf64_Addr r_offset;    /* Loc. at which to apply the action */
3      Elf64_Xword r_info;     /* Index and type of relocation */
4      Elf64_Sxword r_addend;  /* Const. addend used to compute value */
5  } Elf64_Rela;
6
7  typedef struct elf64_sym {
8      Elf64_Word st_name;     /* Symbol name, index in string tbl */
9      unsigned char st_info;  /* Type and binding attributes */
10     unsigned char st_other; /* No defined meaning, 0 */
11     Elf64_Half st_shndx;    /* Associated section index */
12     Elf64_Addr st_value;    /* Value of the symbol */
13     Elf64_Xword st_size;    /* Associated symbol size */
14 } Elf64_Sym;
```

1. si ricerca l'elemento della lista associato al simbolo da proteggere;
2. si gestiscono i valori sicuri (sotto sezione 4.6.1);
3. si impostano i bit di sola lettura (i.e. `RO`) e esecuzione (i.e. `X`) per le entry della *page table* associate alle pagine in cui è stata migrata la funzione critica;
4. si invoca una delle funzioni *core* della patch: la `start_protect_access_to`.

In quest'ultima funzione, il demone validatore una volta risvegliato:

- alloca una nuova istanza della struttura `protected_symbol`;
- la configura registrando la `kprobe` al simbolo critico ed i vari `chkp` necessari agganciandoli alla tabella hash, previa acquisizione del lock;
- infine, in sezione critica, inserisce il simbolo critico nella hash table, se non era già stato fatto in precedenza.

Questo controllo viene svolto a valle delle operazioni necessarie alla configurazione delle strutture dati, per minimizzare la durata della sezione critica sempre al fine di ridurre il più possibile la latenza introdotta dalla patch.

Come ultima cosa è opportuno osservare che sarebbe stato possibile modificare in un unico punto la `load_module`, sfruttando il registro `CR0` presente sull'architetttura hardware x86. Questo contiene al suo interno una serie di bit che permettono di alterare le funzionalità del processore. In particolare, il sedicesimo bit (listato 4.15) permette di abilitare e/o disabilitare la protezione della memoria in scrittura. In definitiva dunque sarebbe stato possibile inserire la logica di migrazione interamente all'interno della `setup_pba_for_module`, invocando la `native_write_cr0` per sovrascrivere il contenuto del buffer originale. Questo approccio tuttavia avrebbe avuto due drawback:

- la portabilità rispetto ad altre architetture che non forniscono in ISA la possibilità di disabilitare la protezione in scrittura della memoria;
- il fornire all'attaccante una finestra temporale Δt in cui la funzione critica f sarebbe stata ancora nella posizione originale e facilmente individuabile attraverso il sottosistema `kallsyms` (e.g. invocando la `kallsyms_lookup_name`).

</> Listato 4.15: Bit per la protezione della memoria in scrittura su x86

```
1 #define X86_CR0_WP_BIT 16 /* Write Protect */
2 #define X86_CR0_WP     _BITUL(X86_CR0_WP_BIT)
```

4.6.1 | Gestione dei valori sicuri

Per ciascun valore sicuro, dichiarato attraverso la `DECLARE_SECURE_VALUE_FOR(fn)`, viene creata una variabile globale all'interno del binario associata ad un parametro, il cui nome è `pba_sv_<fn>` (e.g. se la funzione fosse `crit_fn` verrebbe aggiunta la variabile

`pba_sv_crit_fn`). La funzione `handle_secure_values`, invocata prima di finalizzare il nuovo buffer allocato, è responsabile di caricare all'interno della variabile locale dichiarata tramite `SECURE_VALUE`, il valore fornito a *load time* attraverso tale parametro. Per questo motivo, per ciascun simbolo critico f, nella funzione:

1. si preleva il valore associato al parametro `pba_sv_f`, registrato all'interno dei metadati del modulo dalla `parse_args` invocata in precedenza;
2. si inserisce il valore all'interno della variabile locale di f adibita a tale scopo, dichiarata attraverso la direttiva `SECURE_VALUE`;
3. si azzera il valore del parametro per evitare *leak* di informazioni (e.g. un'attaccante potrebbe altrimenti leggere la chiave dal filesystem `sys`).

L'attività di *patch* del corpo originale di f è senza dubbio la più interessante dal punto di vista realizzativo, motivo per cui viene approfondita di seguito. Lo statement in cui viene espansa la direttiva `SECURE_VALUE` (listato 4.12), una volta compilato per un'architettura x86 a 64 bit, genera l'istruzione macchina[1] riportata nel listato 4.16

</> Listato 4.16: Output della compilazione di `SECURE_VALUE`

```
48 b8 60 8b 1c f0 72 c8 4b ac    movabs $0xac4bc872f01c8b60,%rax
; note that HEX(12415237193757133664) = 0xac4bc872f01c8b60
; where 12415237193757133664 is PBA_SV_PLACEHOLDER
```

L'idea del meccanismo realizzato è quella di ricercare, all'interno del buffer in cui è presente il corpo della funzione critica, l'operando dell'operazione di memorizzazione (i.e. la sequenza di byte `60 8b 1c f0 72 c8 4b ac`) e sovrascriverci il valore specificato a *load time* attraverso il parametro apposito. L'implementazione di tale meccanismo si basa sullo sfruttamento delle `union`, in particolare attraverso una variabile come quella riportata nel

[1] Potrebbe generare anche una scrittura sullo stack, il che sarebbe equivalente dal punto di vista della modifica realizzata, in quanto ciò che conta è l'operando.

listato 4.17 è stato possibile interpretare un'area di memoria di 8 byte secondo due regole differenti (i.e. `unsigned long` per la definizione, `unsigned char [8]` per la ricerca), ovviando dunque al problema dell'*endianess* in entrambe le fasi.

</> Listato 4.17: Definizione della union utilizzata per la patch

```
1 union {
2     unsigned long ul;
3     unsigned char uc[8];
4 } foo;
```

4.7 | Collezione di un checkpoint

La collezione di un checkpoint da parte di un thread T_i è un'attività critica dal punto di vista della sicurezza. Infatti, poiché l'adversary model è costituito da un attaccante che:

- può utilizzare ROP per costruire execution flow arbitrari;
- ha libero accesso al codice sorgente del sistema operativo, moduli inclusi;

è necessario proteggersi anche da tentativi di attacco al sistema difensivo stesso.

La protezione da questo tipo di offensiva si basa sull'introduzione di un meccanismo di *anti-forgery* relativo all'invocazione dei gestori `kprobe`. La segnatura di quest'ultimi è conforme a quanto definito all'interno del header file `include/linux/kprobes.h` (listato 4.18) per cui un attaccante potrebbe:

1. individuare quali checkpoint sono previsti per l'attivazione di una specifica funzione f;
2. costruire, nella propria stack area, una serie di strutture `kprobe` contenenti le informazioni necessari alla collezione;

3. invocare il gestore, tramite una catena di gadget ROP, una volta per ciascun checkpoint da collezionare;

4. saltare infine ad eseguire f riuscendo ad eludere il sistema di protezione in quanto ha collezionato in modo fraudolento i checkpoint previsti.

</> Listato 4.18: Segnatura del gestore `kprobe`

```
1 typedef int (*kprobe_pre_handler_t) (struct kprobe *,
2                                      struct pt_regs *);
```

Per questo motivo è stato introdotto un meccanismo anti-contraffazione basato sul puntatore alla struttura `kprobe` passato al gestore. Al suo interno infatti, il richiedente T_i fornisce attraverso la struttura `validator_work` tale indirizzo al validatore. I vari D_j sono gli unici thread nel sistema a conoscere l'indirizzo della `kprobe` registrata per il checkpoint C_k, in quanto è stato uno di essi ad allocarla ed inserirla nei loro metadati comuni di gestione.

Per questo motivo il demone associato a T_i, una volta risvegliato, cerca all'interno della tabella hash dei checkpoint registrati l'indirizzo ricevuto come parametro. Nel caso in cui questo non sia presente, viene immediatamente dedotto che il thread T_i sta tentando di attaccare il sistema e se ne richiede la terminazione immediata.

Una volta analizzata l'implementazione del meccanismo di anti-contraffazione, la restante parte della logica per la collezione di un checkpoint è di facile comprensione. In particolare il demone alloca una struttura `collected_chkp`, ne popola i campi e la inserisce all'interno della propria tabella hash privata per utilizzarla in seguito in fase di validazione.

Il richiedente T_i, una volta risvegliato dal demone a lavoro ultimato, invoca la `BUG_ON` sul valore restituitogli per richiedere l'interruzione sincrona della sua esecuzione nel caso in cui sia stato valutato una minaccia ed infine prosegue ad eseguire la funzione a cui era stato agganciato il gestore.

Come ultimo aspetto è opportuno sottolineare come, in alternativa alla terminazione forzata del thread giudicato malevolo, sarebbe stato possibile invocare in modo sincrono la `panic` e mandare in halt l'intero sistema. È stato preferito evitare questo approccio in quanto, seppur conservativo, esporrebbe ad attacchi di tipo DoS (Denial of Services).

4.8 | Validazione del percorso d'esecuzione

La validazione del percorso d'esecuzione si basa su una verifica della storia pregressa dell'esecuzione dei thread nel sistema, volta a stabilire se concedere o meno la possibilità di eseguire specifiche funzioni dichiarate critiche a *compile-time*.

Quando un thread T_i invoca una di queste, in automatico parte un gestore installato tramite il sottosistema `kprobe` che avvia un'interazione con il demone di validazione D_j. Anche in questo caso, prima dello svolgimento del lavoro richiesto, il validatore deve verificare l'autenticità dell'invocazione al gestore applicando il meccanismo di anti contraffazione descritto nella sezione precedente. Questo è necessario in quanto altrimenti sarebbe possibile portare avanti un attacco di tipo DoS, inondando il thread validatore di richieste.

Una volta accertata la validità del puntatore alla `kprobe` ricevuto dal thread richiedente come metadato per svolgere il lavoro richiesto, il demone scorre le istanze di struttura `collected_chkp` presenti nella lista di trabocco della propria hash table e verifica quanto descritto nella sezione 4.1; ovvero che la lista dei checkpoint attesi sia contenuta, nell'ordine corretto, in quella dei collezionati.

È interessante osservare come, per motivi d'efficienza, le `hlist` nel kernel Linux permettano unicamente l'inserimento in testa in $O(1)$, motivo per cui la verifica viene svolta a ritroso (i.e. partendo dall'elemento in posizione `nr_checkpoints-1` dell'array dei checkpoint attesi fino ad arrivare allo `0`-esimo, in riferimento alla struttura dati `protected_symbol` descritta nel listato 4.2).

Al termine delle attività di validazione viene caricato nel campo `result` della struttura `validator_work` l'indirizzo a cui è stata migrata la funzione in caso di succcesso, 0 altrimenti. Nella coda del gestore, ovvero una volta che il richiedente viene sbloccato dal demone, si sfrutta una facility offerta dal sottosistema `kprobe` per redirezionare il flusso verso l'indirizzo configurato dal validatore.

Come si evince dalla documentazione, o in alternativa da una analisi diretta del codice sorgente, ogni qual volta si aggancia una `kprobe` ad una funzione del kernel, il sottosistema omonimo sostituisce al primo byte il valore `0xCC` che corrisponde all'*opcode* di un'interruzione da debug (i.e. `INT3`), memorizzando nel campo `opcode` della struttura `kprobe` (listato 4.19) il valore originale. Quando tale byte viene eseguito, in automatico viene generato, dal sottosistema per la gestione delle interruzioni, uno snapshot dello stato dei registri nel formato stabilito dalla struttura `pt_regs`. Il puntatore a tale area di memoria viene fornito al gestore della `kprobe` che viene ad essere attivato, in accordo alla sua segnatura riportata nel listato 4.18.

</> Listato 4.19: Struttura `kprobe`

```
1  struct kprobe {
2      struct hlist_node hlist;
3      struct list_head list;
4      unsigned long nmissed;
5      kprobe_opcode_t *addr;
6      const char *symbol_name;
7      unsigned int offset;
8      kprobe_pre_handler_t pre_handler;
9      kprobe_post_handler_t post_handler;
10     kprobe_opcode_t opcode;
11     struct arch_specific_insn ainsn;
12     u32 flags;
13 };
```

Questo implica che all'interno del gestore, prima dell'esecuzione della funzione stessa

(o in alternativa a valle se si registra un *post-handler*) è possibile modificare lo stato del thread ad esempio modificando i parametri passati. Quindi questa facility è stata sfruttata all'interno della patch per modificare l'*instruction pointer* (i.e. `regs→ip`).

All'interno della documentazione è presente però un *warning* [17, sez. Changing Execution Path] perché nel momento in cui si manipola l'instruction pointer è necessario fare attenzione allo stack frame. Tuttavia il thread in esercizio ha invocato una funzione f e passa ad eseguire f' che è soltanto una copia della prima modificata opportunamente per via della rilocazione. Questo implica che lo stack frame di partenza e quello d'arrivo sono compatibili, motivo per cui è possibile cambiare *in sicurezza* l'indirizzo.

Sempre all'interno della documentazione infine è descritto come far sì che il cambio del program counter abbia effetto. È infatti necessario restituire un valore diverso da `0` affinché si interrompa l'esecuzione *single-step* e si passi direttamente ad eseguire quanto specificato in `regs→ip`.

4.9 | Interruzione della protezione per un simbolo

L'interruzione della protezione di un simbolo è integrata all'interno del processo di rimozione di un modulo in Linux, in modo analogo a quanto avviene con la registrazione (sezione 4.6). Per svolgere questa attività, è stata modificata la system call `delete_module`, la quale è l'*entry point* del processo di rimozione. In particolare è stata aggiunta l'invocazione alla funzione della patch `end_pba_for` subito prima della `try_stop_module`, al cui interno viene cambiato lo stato del modulo (i.e. `LIVE → GOING`).

All'interno della funzione aggiunta, per ciascuna entry dell'array `pba_entries` presente nella struttura `module` (sezione 4.6), si invoca una funzione responsabile di avviare un'interazione con il validatore al fine di liberare le aree di memoria allocate per la protezione ed il monitoraggio dei percorsi d'esecuzione dei thread applicativi.

È opportuno osservare che anche in questo caso è stato necessario introdurre un meccanismo di validazione dell'invocazione in quanto, una funzione del tipo:

```
void end_protect_access_to(const char *name)
```

avrebbe permesso ad un attaccante di invocarla in modo arbitrario, tramite ad esempio ROP, al fine di rimuovere del tutto il meccanismo di protezione. Per ovviare a questo scenario non è stato possibile sfruttare il puntatore alla `kprobe` come nei casi descritti in precedenza, in quanto le funzioni di registrazione ed interruzioni sono le uniche della patch ad essere invocate in modo sincrono dal richiedente.

Il meccanismo realizzato si basa sull'adozione di un *nonce*, in particolare:

1. all'atto della registrazione di un nuovo simbolo da proteggere, il demone di validazione genera una sequenza casuale invocando la `get_random_long` offerta dal kernel linux;

2. memorizza tale maschera di bit all'interno delle struttura dati `protected_symbol` e `validator_work`, rispettivamente nel campo dedicato ed in quello utilizzato per comunicare l'esito del lavoro (sezione 4.3)

3. il richiedente, una volta svegliato dal demone, memorizza il *nonce* ricevuto all'interno del campo apposito della struttura `pba_decl_entry` presente nell'array puntato dalla struttura `module`;

4. infine, passa questo valore random alla funzione responsabile di avviare l'interazione con il validatore all'atto dell'interruzione della protezione.

In questo modo il demone, come prima cosa nella sua routine per l'interruzione, confronta il *nonce* ricevuto con quello memorizzato nelle proprie strutture dati ed in caso di *mismatch* causa l'interruzione forzata del richiedente.

4.10 | Rimozione dei metadati associati ai thread in uscita

Ogni qual volta un thread attraversa un checkpoint, come analizzato in sezione 4.7, viene avviata un'interazione con il demone di validazione che porta, salvo imprevisti, all'allocazione di una struttura dati `collected_checkpoint` per il monitoraggio del suo percorso d'esecuzione ai fini di un'ipotetica futura validazione.

Onde evitare *memory leak* è stato predisposto un meccanismo di recupero della memoria non più necessaria che viene ad essere attivato automaticamente ogni volta che un thread in esercizio avvia l'iter per terminare la sua esecuzione. In particolare, a tal proposito, è stata agganciata tramite `kprobe` la funzione `do_exit`. All'interno del gestore si avvia l'interazione con il demone D_j il quale semplicemente effettua il cleanup di tutti i metadati associati al PID del thread uscente.

4.11 | Rimozione del modulo di sicurezza

Nel capitolo 6 è stato individuato un possibile scenario di sviluppo futuro per la patch realizzata, ovvero quello della trasformazione del meccanismo in un a modulo *out-of-tree*. A tal proposito, sarebbe necessario implementare un meccanismo d'orchestrazione per la terminazione dei validatori al fine di regolamentare il rilascio della memoria allocata per la protezione. Poiché, come descritto in sezione 4.2, il *cuore* del meccanismo sviluppato è già contenuto all'interno di un LKM, per completezza la logica necessaria a regolamentare l'uscita dei demoni è stata già implementata e descritta di seguito.

In sezione 4.5 viene descritto come, per motivi d'efficienza, si prevede l'utilizzo di un pool di validatori anziché di un singolo e di come questi necessitino di condividere talune strutture dati (i.e. la tabella hash dei simboli da proteggere e dei checkpoint).

Nel caso in cui fossero state utilizzate variabili globali per memorizzare gli indirizzi delle

strutture condivise, sarebbe stato più facile individuarle per un'attaccante per via della minore entropia degli indirizzi rispetto a quelli relativi alle aree di stack. Per questo motivo è stata introdotta la struttura dati `val_worker_args` (listato 4.6), come *contenitore* delle informazioni comuni ai validatori. Allo startup del modulo core della patch:

1. si inizializza un'istanza di `val_worker_args` da utilizzare come *modello*;
2. per ciascun demone, si clona tale modello e se ne fornisce l'indirizzo all'atto dello spawn tramite l'apposito argomento della `kthread_run`.

All'interno di questa struttura `val_worker_args` sono presenti anche due informazioni necessarie per orchestrare il *cleanup* della memoria in fase di rimozione del modulo di sicurezza, descritti di seguito:

- tramite `index`, si discrimina il *ruolo* del validatore, ovvero master o worker a seconda del valore assunto dal campo rispettivamente pari o diverso da 0;
- tramite il contatore atomico `finish`, il master è in grado di determinare quando liberare le aree di memoria comuni.

In particolare la logica di terminazione prevede le fasi seguenti:

- all'atto della richiesta di rimozione del modulo di sicurezza, viene invocata la funzione d'uscita in cui si richiede la terminazione dei demoni tramite la `ktread_stop`;
- ciascun validatore worker, una volta ricevuta la richiesta di terminazione, incrementa il contatore atomico, risveglia il master e termina;
- il master attende che tutti abbiano terminato per poi procedere al cleanup della memoria comune.

4.12 | Sfruttamento della randomizzazione

L'architettura descritta nelle sezioni precedenti sfrutta due meccanismi anti contraffazione per validare le invocazioni alle funzioni presenti all'interno della patch. Entrambi si basano sull'utilizzo di informazioni (i.e. l'indirizzo della `kprobe` ed il nonce generato) che diventano dunque critiche dal punto di vista della robustezza dello schema.

Nel caso in cui l'attaccante riuscisse infatti a pervenire a tali informazioni potrebbe bypassare completamente il meccanismo di protezione ed eseguire arbitrariamente come se la patch non fosse installata. La soluzione a questo problema risiede nel modo in cui queste informazioni sono memorizzate, ovvero la randomizzazione della loro posizione.

Si considerino gli scenari della collezione di un checkpoint e della validazione del percorso d'esecuzione. Un attaccante può *facilmente* ottenere l'indirizzo della tabella hash in cui il sottosistema `kprobe` mantiene le strutture omonime registrate, in quanto è un simbolo globale. A questo punto potrebbe utilizzare opportuni gadget per traversare la lista ed individuare le `kprobe` da passare al validatore per eludere il controllo tra quelle registrate. Questo sarebbe possibile perché, conoscendo il codice sorgente, l'attaccante saprebbe con precisione di quanto spiazzarsi dalla testa per raggiungere il campo `hlist` (listato 4.19).

Una situazione analoga si avrebbe nel caso dell'interruzione della protezione, in cui in assenza di randomizzazione, un attaccante una volta ottenuto l'indirizzo di un campo qualsiasi della struttura `module` potrebbe spiazzarsi in modo relativo e pervenire ai *nonce* in essa memorizzati.

Analizzando il codice sorgente della versione del kernel Linux di riferimento, si scopre che la struttura `module` è già di per se randomizzata, a differenza della `kprobe`, motivo per cui è stato necessario modificare l'header file `include/linux/kprobes.h` ed aggiungere la direttiva `__randomize_layout` in calce alla sua dichiarazione.

4.12.1 | Dettagli sul plugin `randstruct`

La randomizzazione delle strutture dati è una facility offerta dal plugin `randstruct`, disponibile per il compilatore `gcc` e facente parte del progetto KSPP (Kernel Self Protection Project). L'obiettivo di quest'ultimo è quello di innalzare il livello di sicurezza della mainline linux, ad esemprio attraverso il porting del progetto PaX analizzato in sezione 2.3. Come descritto nell'articolo pubblicato su LWN [21], il plugin permuta i campi di una struttura a partire da un seed fissato a *compile-time*.

È proprio questo seme il punto critico della randomizzazione, in quanto per permettere la compilazione di moduli *out-of-tree*, le distribuzioni Linux (e.g. Ubuntu, Fedora) devono fornirlo agli utilizzatori finali, fra cui anche i potenziali attaccanti. Poiché il seed è unico per binario rilasciato, una volta individuato è possibile vanificare completamente gli effetti della permutazione dei campi su tutte le macchine in cui esso è installato. Per questo motivo lo stesso Torvalds in una mailing list [22] ha affermato: *"So it's imnsho a pretty questionable security thing. It's likely most useful for one-off 'special secure installations' than mass productions."* a cui Cook (i.e. uno dei membri del KSPP) ha replicato: *"Well, Facebook and Google don't publish their kernel builds. :)"*.

4.13 | Ulteriori dettagli

In questa sezione vengono presentati dettagli minori relativi all'implementazione dello schema difensivo oggetto di questo lavoro di tesi.

4.13.1 | Posizionamento delle funzioni della patch nell'ELF

Nel caso in cui un attaccante riuscisse ad agganciare, tramite `k[ret]probe`, le funzioni proprie del meccanismo di sicurezza potrebbe monitorare delle esecuzioni benevole al fine

di individuare i parametri corretti da utilizzare per eludere il meccanismo di sicurezza.

Per evitare questo scenario, ogni funzione della patch è stata *marcata* con la direttiva `__kprobes` introdotta nella digressione presente nel capitolo 3 per posizionare il codice all'interno della sezione `.kprobes.text` che è una zona non accessibile al sottosistema `kprobe`.

4.13.2 | Esempi d'uso del meccanismo di protezione

Per aiutare a comprendere come utilizzare la patch di sicurezza descritta in questo documento, sono stati aggiunti degli esempi *ad hoc* all'interno della directory `samples/pba/`. Il primo (i.e. `exposer`), come suggerisce il nome, espone 3 funzioni e specifica un percorso d'attivazione valido per uno di essi, come descritto in sezione 4.4. Nella procedura critica si mostra:

1. l'effettiva migrazione dell'area contenente il codice macchina, stampando, tramite `printk`, i primi 15 byte dell'area di memoria originale;
2. la corretta inizializzazione di un valore segreto specificato a *load-time*.

Nel secondo (i.e. `happy_scenario`) e nel terzo (i.e. `unhappy_scenario`) modulo d'esempio si mostra l'effettivo funzionamento del meccanismo di sicurezza, in particolare:

- nel secondo, viene concesso il permesso di eseguire il simbolo critico al thread in esercizio, in quanto passa precedentemente in tutti i checkpoint nell'ordine previsto;
- nel terzo, si nega l'accesso in quanto non viene invocato uno dei checkpoint attesi.

4.13.3 | Gestione dell'audit

Per motivi prettamente prestazionali, il meccanismo di sicurezza produce *by default* un set minimale di informazioni nel buffer circolare adibito alle operazioni di log, per di più in casi sporadici (e.g. registrazione di un nuovo simbolo critico). Tuttavia in ottica di sviluppo e debugging è comodo poter abilitare un'esecuzione *verbosa*, motivo per cui è stato introdotto un parametro di configurazione apposito (i.e. `[CONFIG_]PBA_DEBUG`). Questo viene utilizzato per abilitare o meno il flag di compilazione `-DDEBUG`, il quale incide sull'effettivo comportamento della macro `pr_devel` adottata all'interno del codice. Come si evince dalla documentazione infatti, nel caso in cui `DEBUG` è definita, questa espande in `printk(KERN_DEBUG …)`, altrimenti in una `nop` (i.e. `no_printk`).

Inoltre per facilitare il rispetto dei vincoli di *code style* previsti per lo sviluppo kernel[2] (i.e. il rispetto delle 80 colonne) è stata ridefinita, come spesso avviene, la macro `pr_fmt`, la quale permette di passare ad esempio da `pr_info(KBUILD_MODNAME ": whatever\n")` a `pr_info("whatever\n")`.

[2] A tal proposito è stato utilizzato il tool `clang-format` offerto da LLVM per formattare il codice realizzato in riferimento al set di regole stabilite all'interno del file `.clang-format`.

Capitolo 5

Valutazione

5.1 | Use case d'applicazione di PBA

Per esemplificare l'utilizzabilità del meccanismo di sicurezza sviluppato è stato ideato uno use-case *minimale* apposito, descritto in questo capitolo.

Lo scenario di riferimento è quello del char device driver `r2s` (root-to-secret) realizzato per il kernel Linux. Esso permette ai thread T_i in esercizio di accedere in lettura specifici nodi del virtual file system al fine di reperire un'informazione segreta S, fornita unicamente nel caso in cui lo UID di T_i sia pari a `0`. Dallo pseudo-codice 1, relativo all'implementazione della `read`, risulta evidente come, nel caso in cui non si utilizzasse PBA, un attaccante una volta ottenuto l'indirizzo della funzione di lettura, potrebbe reperire l'informazione in modo illegittimo, sfruttando ad esempio un attacco basato su ROP.

Algoritmo 1 Lettura in `r2s`

```
if T is root then
    print error message
    kill T
else
    call read_secret
end if
```

Per questo motivo nell'implementazione proposta nel file `samples/pba/r2s.c` è stata utilizzata l'interfaccia analizzata in sezione 4.4 per sfruttare il meccanismo di protezione descritto in questo documento. In particolare, per sperimentare la robustezza dello schema, è possibile *costruire* la POC (Proof Of Concept) seguente, rispettando gli step riportati in elenco:

1. installare la versione del kernel *patchata* per poter utilizzare le facility di sicurezza descritte in questo documento, impostando `CONFIG_SAMPLE_PBA=m` in `.config` per generare i kernel object degli esempi;

2. scegliere un segreto S e specificarlo all'atto del montaggio del modulo, tramite:

```
/path/to/r2s # insmod r2s.ko pba_sv_read_secret=S
```

3. creare il nodo del file system da pilotare con il driver inserito e renderlo leggibile a tutti, attraverso:

```
/path/to/r2s # export MAJOR=$(cat /sys/module/r2s/parameters/major)
/path/to/r2s # mknod user/secret c $MAJOR 0
/path/to/r2s # chmod 444 user/secret
```

4. compilare l'applicativo utente, ovvero un programma C elementare che apre il file, richiede la lettura di al più `4k` byte e termina, eseguendo:

```
/path/to/r2s $ cd user && gcc poc.c -o poc
```

5. provare ad eseguire il binario come utente normale o `root`:

```
/path/to/r2s/user $ [sudo] ./poc
```

Finora è stato descritto il meccanismo di simulazione dell'*happy scenario*, ovvero quello di un thread benevolo che esegue la funzione critica `read_secret` passando per l'execution path atteso, ossia invocando prima la `vfs_read` e successivamente la `r2s_read`. Per simulare l'attività malevola di un attaccante, è stato sviluppato un altro modulo *ad hoc*, ovvero `rop_simulator`. Per facilitare la realizzazione dell'attacco oggetto di questa demo, nella funzione di inizializzazione di `r2s` viene emesso, sul buffer circolare adibito al log, l'indirizzo logico della funzione critica `read_secret`. Questo può essere letto e passato, all'atto del montaggio, a `rop_simulator` sfruttando l'apposito parametro `address`.

Attraverso l'esempio descritto in questo capitolo è possibile osservare e sperimentare direttamente i possibili *outcome*, funzione dell'operato più o meno legittimo degli utenti nel sistema, in particolare:

1. nel caso in cui si esegue il comando `sudo ./poc`, viene mostrato a schermo il segreto fornito dal driver, per effetto dell'implementazione del binario;
2. viceversa, se lo stesso comando viene eseguito senza `sudo`, il thread termina in modo anomalo la propria esecuzione;
3. infine, se si simula uno scenario d'attacco tramite `rop_simulator` viene a generarsi un'eccezione tracciabile che porta all'interruzione sincrona del thread in esercizio.

5.2 | Analisi delle prestazioni

Il meccanismo di sicurezza sviluppato prevede l'introduzione all'interno del sistema di una serie di demoni di validazione responsabili di monitorare l'esercizio dei thread applicativi. Questo monitoraggio si basa sullo sfruttamento del sottosistema `kprobe` e dunque sull'*agganciare* specifiche funzioni del kernel al fine di eseguire attività prima della loro effettiva invocazione. Per questo motivo è stato necessario portare avanti un'analisi prestazionale al fine di determinare l'overhead effettivo introdotto dalla patch PBA. L'obiettivo di questa fase è stato quello di valutare i coefficienti α_{chkp} e α_{tgt} dell'equazione seguente:

$$\begin{aligned}\Delta T &= T_0 + n_{chkp}T_{chkp} + n_{tgt}T_{tgt} + T_{PBA} \\ &= T_0 + n_{chkp}T_{chkp} + n_{tgt}T_{tgt} + n_{chkp}\alpha_{chkp} + n_{tgt}\alpha_{tgt} \\ &= T_0 + n_{chkp}\left(\alpha_{chkp} + T_{chkp}\right) + n_{tgt}\left(\alpha_{tgt} + T_{tgt}\right)\end{aligned}$$

che rappresenta il tempo d'esecuzione di un thread che:

Numero di CPU	2
Memoria RAM	4 GB
Versione del kernel Linux	5.15.75
Architettura hardware	x86_64
Virtualizzatore	Oracle VM VirtualBox (v. 7.0.4 r154605)

Tabella 5.1: *Caratteristiche tecniche della macchina adottata per i test prestazionali*

- impiega T_0 per eseguire funzioni il cui *wall clock time* è indipendente dalla presenza o meno della patch;
- passa per n_{chkp} checkpoint e per n_{tgt} simboli critici, impiegando per ciascuno di essi α_X per eseguire attività della patch e T_X per eseguire il corpo effettivo della funzione con $X \in \{chkp, tgt\}$.

Per valutare in modo empirico i coefficienti α_{chkp} e α_{tgt} sono stati eseguiti i test descritti di seguito in un ambiente le cui caratteristiche sono riportate in tabella 5.1. In particolare per stimare α_{chkp} è stato utilizzato un modulo apposito di test:

- contenente una funzione critica ed un checkpoint (i.e. `chkp`), entrambe con una logica minimale;
- dichiarando, tramite l'interfaccia descritta in sezione 4.4, il percorso d'attivazione valido.

Per catturare con precisione i tempi d'esercizio è stata utilizzata la facility `ktime` offerta dal kernel Linux, la quale permette di lavorare alla sensibilità dei nanosecondi. In tabella 5.2 sono riportati i risultati delle 10000 invocazioni fatte alla funzione `chkp`, con e senza la patch attiva. Analizzando i tempi medi e le deviazioni standard raccolte è possibile osservare come l'overhead introdotto dalla patch PBA sia minimale (i.e. $\alpha_{chkp} \sim 8\ \mu s$), il che è in linea con quanto atteso poiché l'implementazione della collezione di un checkpoint (sez. 4.7) è non bloccante e ottimizzata per via dell'utilizzo di hash table. La maggiore

Patch	Media (ns)	Deviazione Standard (ns)
Abilitata	8189.0239	4478.9672
Non abilitata	27.6335	2.4061

Tabella 5.2: *Media e deviazione standard del tempo d'esecuzione di un checkpoint (dimensione campione: 10000)*

Patch	Media (ns)	Deviazione Standard (ns)
Abilitata	8057.9911	5101.7633
Non abilitata	32.3777	0.9879

Tabella 5.3: *Media e deviazione standard del tempo d'esecuzione di un simbolo critico (dimensione campione: 10000)*

variabilità dei tempi nel caso di patch attiva è dovuta al fatto che le attività di protezione richiedono l'interazione con un demone, il quale può essere già sveglio quando viene contattato, oppure in attesa sulla propria wait queue privata.

In modo analogo è stata portata avanti l'analisi sui tempi d'esercizio di un simbolo critico. In questo caso è stato adottato un modulo kernel apposito contenente due[1] checkpoint associati ad una funzione critica (i.e. `crit`). Una volta eseguiti i primi è stata stimolata `crit` per 10000 volte, come nel caso precedente, e sono stati raccolti i risultati in tabella 5.3. Analizzando quanto ottenuto è possibile notare come, anche in questo caso, l'overhead sia limitato (i.e. $\alpha_{tgt} \sim 8\ \mu s$) e circa equivalente a quello dei checkpoint. In linea di principio l'attività di validazione dovrebbe impegnare il demone per un tempo maggiore rispetto a quella di collezione, in quanto è necessario scorrere la lista di trabocco della tabella hash per controllare i checkpoint collezionati dal richiedente. Tuttavia i risultati di quest'attività di test mostrano come utilizzare una *hash table* abbia permesso di minimizzare la profondità delle liste di trabocco, rendendo circa costante il tempo di lookup. Rimangono infine valide le considerazioni sulla differenza delle deviazioni standard in funzione dell'abilitazione della patch.

[1] Ne sono stati scelti 2 come stima del numero di checkpoint medio che uno sviluppatore di LKM potrebbe adottare per i propri simboli critici.

Capitolo 6

Conclusioni e sviluppi futuri

L'obiettivo di questa tesi era lo sviluppo di una soluzione difensiva contro l'esecuzione illecita di funzioni critiche nel kernel. Il meccanismo realizzato è funzionale e a basso impatto prestazionale. Seppure il lavoro svolto può definirsi completo rispetto al set di requisiti funzionali stabilito ad inizio lavori, può di certo essere ampliato e migliorato al fine di renderlo interessante rispetto ad un numero maggiore di scenari d'applicazione.

Il meccanismo di protezione realizzato, oggetto di questo lavoro di tesi, ha un invasività limitata relativamente alla codebase esistente. Basti pensare che è stato modificato un unico sorgente (i.e. `kernel/module.c`) ed alcuni file d'intestazione a differenza di meccanismi come Adelie e RAP che richiedono modifiche più sostanziali come instrumentazioni e compilazione PIC. Tuttavia, per sua natura, una patch non è qualcosa di applicabile a runtime in un'installazione già esistente, bensì necessita di un processo di build *ex-novo*.

Si consideri ora uno scenario in cui l'amministratore di un certo sistema S necessita di aggiungere a runtime un modulo M contenente un simbolo per cui sarebbe vantaggioso specificare un percorso d'attivazione valido. Il sistema operativo in esercizio su S tuttavia è stato configurato ed installato prima del rilascio della patch PBA. Allo stato attuale, l'amministratore dovrebbe riconfigurare l'intero ambiente per poter aggiungere la facility di protezione. Sarebbe decisamente più vantaggioso poter:

1. montare a runtime un modulo P *equivalente* alla patch sviluppata in termini di funzionalità offerte;

2. inserire il modulo M, dichiarando il percorso d'esecuzione valido per il simbolo

critico, sfruttando la libreria descritta nella sezione 4.4.

Questo è proprio uno dei possibili sviluppi futuri di questo lavoro di tesi. In particolare sarebbe necessario affidare la responsabilità di gestire l'inizializzazione e l'interruzione della protezione dei simboli rispettivamente alle funzioni di startup e shutdown del modulo in cui sono contenuti. Tuttavia, come analizzato in sezione 4.6, per individuare con precisione l'area di memoria da migrare contenente il simbolo critico, si sfruttano specifiche sezioni dell'ELF.

Nel capitolo 3 viene mostrato come *in fondo* alla `load_module` viene invocata la `free_copy` responsabile di liberare l'area di memoria contenente il binario originale e che solo in seguito, nella `do_init_module`, viene chiamata la funzione `init` del modulo. Per questo motivo, per rendere P *equivalente* alla patch sviluppata, sarebbe necessario realizzare un plugin di post compilazione per inserire, nella sezione `.rodata` dei kernel object, il `.ko` stesso così da aver accesso all'header e alle informazioni di rilocazione contenute nelle sezioni `".rela*"`.

Sarebbe inoltre possibile aumentare il livello di protezione offerto, introducendo un meccanismo di randomizzazione continua delle aree critiche. A tal fine sarebbe necessario introdurre un demone ulteriore al pool, il cui obiettivo sarebbe quello di migrare periodicamente le aree protette in modo *safe* rispetto all'esercizio degli altri validatori. Per garantire questa *safeness* sarebbe possibile adottare un registro atomico wait-free come ARC [23], per mantenere in modo *packed* il numero di validatori afferenti all'epoca $i-$esima del posizionamento in memoria degli oggetti critici e l'indirizzo dell'area contenente il codice da proteggere. In questo modo sarebbe possibile variare in modo efficiente e scalabile la posizione in memoria del simbolo critico rendendo più complesse eventuali attività di discovery degli indirizzi dilazionate nel tempo per un attaccante.

Bibliografia

[1] RedHat. «Che cos'è il malware?» (Giu. 2018), indirizzo: https://www.redhat.com/it/topics/security/what-is-malware.

[2] J. P. Anderson, «Computer security technology planning study,» ANDERSON (JAMES P) e CO FORT WASHINGTON PA FORT WASHINGTON, rapp. tecn., 1972.

[3] A. Jajoo, «A study on the Morris Worm,» *CoRR*, vol. abs/2112.07647, 2021. arXiv: 2112.07647. indirizzo: https://arxiv.org/abs/2112.07647.

[4] R. Roemer, E. Buchanan, H. Shacham e S. Savage, «Return-Oriented Programming: Systems, Languages, and Applications,» *ACM Trans. Info. & System Security*, vol. 15, n. 1, mar. 2012.

[5] A. Bansal e D. Mishra, «A practical analysis of ROP attacks,» *CoRR*, vol. abs/2111.03537, 2021. arXiv: 2111.03537. indirizzo: https://arxiv.org/abs/2111.03537.

[6] R. Nikolaev, H. Nadeem, C. Stone e B. Ravindran, «Adelie: Continuous Address Space Layout Re-randomization for Linux Drivers,» *CoRR*, vol. abs/2201.08378, 2022. arXiv: 2201.08378. indirizzo: https://arxiv.org/abs/2201.08378.

[7] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual: Instruction Set Reference, A-Z, Volume 2 (2A, 2B, 2C & 2D)*, Operating System, 2022.

[8] L. Erdődi, «Conditional Gadgets for Return Oriented Programming,» set. 2013. doi: 10.13140/2.1.2179.2646.

[9] C. LLVM. «Control Flow Integrity.» (2022), indirizzo: https://clang.llvm.org/docs/ControlFlowIntegrity.html.

[10] J. Edge. «Control-flow integrity for the kernel.» (gen. 2020), indirizzo: https://lwn.net/Articles/810077/.

[11] grsecurity. «Close, but No Cigar: On the Effectiveness of Intel's CET Against Code Reuse Attacks.» (dic. 2016), indirizzo: https://grsecurity.net/effectiveness_of_intel_cet_against_code_reuse_attacks.

[12] O. B. B. S. Ittai Anati. «Control Flow Enforcement Technology, Compiler Architecture and Tools Conference (CATC).» (dic. 2017), indirizzo: https://www.intel.com/content/dam/develop/external/us/en/documents/catc17-introduction-intel-cet-844137.pdf.

[13] P. Team. «RAP: RIP RIP.» (ott. 2015), indirizzo: https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf.

[14] J. Moreira, S. Rigo, M. Polychronakis e V. P. Kemerlis, «DROP THE ROP fine-grained control-flow integrity for the Linux kernel,» *Black Hat Asia*, 2017.

[15] R. Nikolaev e B. Ravindran, «Hyaline: Fast and Transparent Lock-Free Memory Reclamation,» in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, ser. PODC '19, Toronto ON, Canada: Association for Computing Machinery, 2019, pp. 419–421, isbn: 9781450362177. doi: 10.1145/3293611.3331575. indirizzo: https://doi.org/10.1145/3293611.3331575.

[16] B. developers. «BusyBox v. 1.35.0.» (2022), indirizzo: https://elixir.bootlin.com/busybox/1.35.0/source.

[17] M. H. Jim Keniston Prasanna S Panchamukhi, *Kernel Probes (Kprobes)*, Kernel Linux manual, 2022. indirizzo: https://docs.kernel.org/trace/kprobes.html.

[18] 0xAX, *Linux Insides*, Kernel Linux manual, 2022. indirizzo: https://0xax.gitbooks.io/linux-insides/content/.

[19] K. development community, *The kernel's command-line parameters*, Kernel Linux manual, 2022. indirizzo: https://www.kernel.org/doc/html/v5.15/admin-guide/kernel-parameters.html.

[20] M. Matz, J. Hubička, A. Jaeger e M. Mitchell, *System V Application Binary Interface AMD64 Architecture Processor Supplement*, Manual, lug. 2012. indirizzo: https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf.

[21] N. Hussein. «Randomizing structure layout.» (mag. 2017), indirizzo: https://lwn.net/Articles/722293/.

[22] L. T. Kees Cook. «Mailing list ARChives - Linux Kernel.» (mar. 2017), indirizzo: https://marc.info/?l=linux-kernel&m=149083865202302&w=2.

[23] HPDCS. «ARC - Wait free atomic register.» (2018), indirizzo: https://github.com/HPDCS/ARC.

Ringraziamenti

Ci tengo a dedicare uno spazio nell'elaborato alle persone che mi hanno sostenuto durante questo percorso. In primis, un ringraziamento sentito va ai miei relatori, i professori Francesco Quaglia e Alessandro Pellegrini, persone per le quali nutro una grande stima e con cui ho avuto l'onore di collaborare per realizzare questo lavoro di tesi. Ringrazio di cuore la mia famiglia per avermi trasmesso i valori che mi hanno reso l'uomo che sono oggi. A mamma, una persona pura, semplice ed amorevole. Grazie per i messaggi e le chiamate dopo ogni esame, per essere stata così presente in questo lungo viaggio. A papà, un uomo onesto, leale, un modello a cui ispirarmi. Grazie per avermi trasmesso il significato di determinazione e la passione per il mondo dell'informatica. A mia sorella Sofia, la mia migliore amica, una persona in grado di ascoltare e capire nel profondo gli altri. Grazie per la tua capacità di alleggerire le giornate più dure con la tua allegria. Un ringraziamento speciale a Giulia, la mia costante. Grazie per ogni singolo gesto, dal semplice revisionare con cura ogni mia email, al venire da me all'improvviso la sera tardi soltanto per calmare le mie ansie. Le piccole cose che abbiamo condiviso in cinque anni le porterò sempre con me. Menzione d'onore ai migliori ingegneri che conosco, i compagni con cui ho condiviso questo percorso, la vostra presenza ha reso il tutto più interessante e leggero. Grazie perché confrontandomi con voi ho avuto l'opportunità di crescere dal punto di vista personale e professionale. Un grazie in particolare a Cristiano, più di un semplice collega per me. Un vero professionista, un eccellente compagno di lavoro, ma soprattutto un amico speciale. Infine a chi c'è stato, a chi anche solo con un pensiero mi è stato accanto in questo capitolo della mia vita. Grazie del supporto, vi voglio bene.