UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

**DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE**

**CORSO DI LAUREA IN INGEGNERIA INFORMATICA**

# Meccanismi di difesa del kernel Linux

**Relatore: Prof. Mauro Migliardi**

**Laureando: Jacopo Lazzarin**

**ANNO ACCADEMICO 2021 – 2022**

**Data di laurea 21/09/2022**

*A mia madre Cipriana e sorella Letizia*
*agli amici: Elena, Anna, Manuel, Giacomo ed Enrico*
*grazie per aver adornato questi tre anni di ricordi preziosi*

# INDICE

**Sommario**

Il kernel Linux è l'elemento più critico e complesso di un moderno sistema operativo UNIX-like e richiede un elevato grado di sicurezza viste le sue molteplici applicazioni nel mondo informatico. In modo da proteggere il kernel da attacchi malevoli, e anche da sé stesso, sono stati introdotti numerosi meccanismi di difesa atti a risolvere o mitigare le vulnerabilità più comuni e i modi per sfruttarle. In questo lavoro si andrà a studiare lo stato attuale dei principali meccanismi di difesa della memoria del kernel Linux. In particolare, si andranno ad analizzare nel dettaglio il funzionamento degli stack canaries per quanto riguarda la memoria statica; e i costi di overhead causati dalla forzata inizializzazione della memoria dinamica.

# INTRODUZIONE

Ormai nella nostra vita quotidiana, sia nella sfera personale che in quella lavorativa, siamo assistiti da una qualche forma di dispositivo informatico. Personal computer, server, cellulari e apparati di rete sono tutti esempi di macchine pensanti che regolano i propri componenti elettronici utilizzando del software, e l'integrazione di processori all'interno di altri dispositivi come orologi, televisioni e automobili sta rendendo questa pervasione dei sistemi informatici una realtà sempre più scontata. L'elemento indispensabile che unisce tutti questi dispositivi è la presenza di un sistema operativo che ne gestisca le risorse, e il nucleo di un sistema operativo, chiamato kernel, ne è l'elemento più critico, che va a gestire direttamente l'hardware offrendo, allo stesso tempo, le fondamenta per costruire del nuovo software su di esso. È interessante poter osservare che, nonostante le numerose differenze nei dispositivi informatici e nelle loro applicazioni, vi sia un kernel molto usato che riesca ad adattarsi alle più diverse situazioni, e il suo nome è Linux.

In questo lavoro si è interessati alla sicurezza che il kernel Linux è in grado di offrire, con particolare attenzione ad una categoria di difese del kernel di basso livello, generalmente mirate alla protezione della memoria e dei suoi utilizzi impropri, ai quali ci si riferirà con il nome di meccanismi di difesa. Il motivo di tanto interesse verso queste mitigazioni è dovuto al fatto che, ad oggi, tra le vulnerabilità più comuni nel software [1] sono ancora presenti errori nella gestione della memoria, come scritture e letture al di fuori di un buffer, Use After Free e dereferenziazione di puntatori nulli. Per motivi storici e di efficienza, il kernel Linux è scritto in linguaggio C, un linguaggio che ripone la responsabilità del corretto utilizzo della memoria sulle spalle del programmatore. Considerate le dimensioni impressionanti del progetto e del numero di persone differenti che ve ne prendono parte, è inevitabile che in Linux, così come in altri programmi, siano presenti delle vulnerabilità che possono essere sfruttate da un attaccante per compromettere la confidenzialità, l'integrità o la disponibilità delle informazioni memorizzate nel sistema. Il ruolo svolto dai meccanismi di difesa del kernel è quello di contrattaccare comportamenti malevoli o instabili annullando la presenza di intere classi di vulnerabilità, oppure rendendone estremamente più difficile lo sfruttamento da parte di un ataccante nonostante le vulnerabilità rimangano presenti nel codice.

I principi di funzionamento dei meccanismi di difesa del kernel Linux sono strettamente legati alla memoria che vanno a proteggere, chiamata kernel space, e per essere compresi occorre prima comprendere come viene gestita questa memoria e quali sono le principali differenze con il corrispettivo user space.

Parafrasando quanto scritto nella documentazione di Linux [10], un meccanismo di difesa ideale dovrebbe essere efficace, sempre attivo di default, che non impatti negativamente le prestazioni del sistema e il funzionamento di altre funzionalità del kernel. Utilizzando questa descrizione come modello di riferimento, nella parte finale di questo elaborato si andranno ad analizzare due piccoli meccanismi di hardening della memoria, chiamati *init_on_alloc* e *init_on_free*. L'analisi di questi due meccanismi ha un duplice scopo: il primo, è quello di andare a fornire un caso di studio rigoroso, la cui metodologia possa essere applicata ed adattata anche allo studio di altri meccanismi di difesa del kernel simili; il secondo è quello di ottenere dei dati a supporto dell'abilitazione o disabilitazione di queste due funzionalità.

## ORGANIZZAZIONE DEL LAVORO

La versione del kernel Linux presa in considerazione per tutta la durata di questo lavoro è la 5.15.0. Al momento, l'ultima versione stabile del kernel Linux è la 5.19.8; tuttavia i meccanismi qui descritti sono tutti già consolidati da tempo all'interno del kernel e non dovrebbero aver subito sostanziali modifiche tali da rendere obsolete le argomentazioni riportate in questo documento. Per l'analisi del codice sorgente è stata utilizzata una pubblica istanza del navigatore di codice Elixir.* L'utilizzo di un navigatore ha agevolato notevolmente l'analisi del codice sorgente, favorendone la lettura, la ricerca delle definizioni di funzioni, macro e i loro riferimenti all'interno del codice. Per quanto riguarda il lato hardware, gli esperimenti sono stati eseguiti su di un processore Intel Core i7-3537U.

Il resto dell'elaborato è suddiviso in quattro capitoli, qui brevemente descritti:

### *Capitolo 1 - Una breve storia di Linux*

Il primo capitolo ha lo scopo di contestualizzare il kernel Linux all'interno del mondo informatico raccontandone brevemente le sue origini. Le prime due sezioni del capitolo sono dedicate, rispettivamente, alla descrizione del sistema UNIX di Ken Thompson e alla filosofia del software libero di Richard Stallman, entrambi elementi fondamentali nella nascita di Linux. La terza sezione raccoglie alcuni miglioramenti di sicurezza che il kernel ha apportato rispetto ai sistemi UNIX tradizionali.

### *Capitolo 2 - Gestione della memoria*

Il secondo capitolo discute la gestione della memoria in due parti: stack e heap. La gestione di queste due aree di memoria nel kernel è molto diversa rispetto allo user space e, a volte, può risultare più complessa. Nella parte dedicata alla memoria stack vengono discussi gli attacchi caratteristici a cui può essere assoggettata e viene trattata una delle prime mitigazioni di sicurezza ancora in uso oggi: i canarini dello stack. Nella seconda parte vengono descritte le caratteristiche principali degli allocatori della memoria heap, così come i loro meccanismi di debug.

### *Capitolo 3 - Hardening della memoria*

Questo capitolo si concentra sullo studio dei meccanismi di difesa *init_on_alloc* e *init_on_free*. La prima sezione descrive il loro funzionamento e come questi meccanismi si integrino con il resto del kernel; nella seconda viene descritta la metodologia adottata per il loro studio; e nella terza vengono riportati i risultati ottenuti.

### *Capitolo 4 - Osservazioni finali*

Nel capitolo finale vengono discussi i risultati riportati nei capitoli precedenti e vengono fatte delle considerazioni sullo stato attuale dei meccanismi di difesa che il kernel Linux è in grado di offrire. In questa parte conclusiva vengono anche esposti i principali problemi riscontrati nello studio e si accenna a dei possibili miglioramenti che favorirebbero l'intera comunità del kernel.

---

* https://elixir.bootlin.com/linux/v5.15/source

# 1 UNA BREVE STORIA DI LINUX

Linux è un kernel monolitico e modulare UNIX-like, libero ed open source, che negli ultimi decenni ha guadagnato una certa popolarità all'interno del mondo informatico. Più impropriamente, il nome Linux è comunemente utilizzato per riferirsi ad una vasta famiglia di sistemi operativi che trovano la loro applicazione in numerosi settori diversi tra loro, tra i quali possiamo citare quello delle reti di calcolatori, dei sistemi real-time, dei dispositivi embedded e, seppur in modo meno diffuso, dei personal computer. Da un punto di vista storico, Linux ha rappresentato un grosso contributo per l'affermazione e sviluppo del movimento per il software libero e del progetto GNU. Tuttavia, è opportuno ricordare che Linux non nasce dal nulla: ci sono almeno 20 anni di storia, precedenti alla sua creazione, che influenzano il suo design e le direzioni che il kernel ha preso.

In questo primo capitolo andremo a tracciare un breve riassunto di questi 22 anni, descrivendo gli eventi più salienti, i sistemi che li hanno caratterizzati e i loro autori. L'obiettivo è quello di contestualizzare il kernel Linux descrivendone l'origine di alcune delle funzionalità che lo compongono e di mettere in luce questa parte affascinante di storia informatica.

## 1.1 LE ORIGINI: UNIX

UNIX è un sistema operativo scritto da Ken Thompson nel 1969, all'epoca ricercatore presso i Bell Labs di proprietà dell'American Telephone and Telegraph Company (AT&T). In quell'anno l'AT&T si era appena ritirata da un progetto congiunto per la creazione di un nuvo sistema operativo, chiamato MULTICS, al quale Thompson aveva contribuito come programmatore. Grazie all'esperienza guadagnata, Thompson iniziò a progettare e scrivere un file system per il disco di un calcolatore PDP-7 pressoché inutilizzato dai laboratori. Tuttavia, per poter sfruttare appieno le funzionalità di un file system occorrevano programmi che interagissero con esso, così Thompson implementò una *shell* e il concetto di *processo* all'interno dei propri sorgenti, avvicinando il proprio programma ad un sistema operativo rudimentale.

Non ci volle molto perché altri ricercatori interessati si unissero a Thompson, proponendo nuove idee ed implementando funzionalità aggiuntive, che stabilizzarono la direzione del progetto in un sistema operativo a kernel monolitico. Tra i componenti di questo gruppo possiamo trovare menti come quelle di Dennis Ritchie, Brian Kernighan e Douglas McIlroy. Alla fine, il sistema prese il nome di Unics (Uniplexed Information Computing Service), poi UNIX, come scherzoso riferimento a MULTICS.

Nel 1971, il codice assembly di UNIX venne riadattato per un PDP-11 e vennero introdotte nuove funzionalità e strumenti ancora in uso oggi (e.g. *sh*, *ls*, *cat*, *find*, *chown*). Nel 1973, il codice venne riscritto in linguaggio C, creato da Ritchie, rendendo il programma più facile da mantenere e portabile per diverse architetture. Questa terza versione di UNIX presentava una struttura fondamentale che ancora oggi compare nei suoi derivati, Linux compreso, e della quale citiamo alcuni elementi caratteristici [19]:

- Un file system ad albero formato da *inodes* che comprende anche le periferiche esterne, gestite come *file speciali*. Questo permette di realizzare un modello universale di I/O che gestisce tutti i possibili tipi di file tramite le stesse chiamate di sistema (e.g. *open*, *read*, *write*, *close*).

- Un'implementazione del concetto di *pipe* che permette di collegare gli stream di input e output dei programmi di una shell in un unico flusso di lavoro. Questo permette di concentrarsi sullo sviluppo di programmi con funzionalità specifiche, lasciando ampia libertà nel collegarli tra loro per realizzare un compito complesso.
- Un veloce tasso di creazione di nuovi processi, gestiti tramite le chiamate *fork*, *execve*, *wait* e *exit*. È interessante osservare come la creazione di un nuovo spazio di indirizzi e l'esecuzione di un nuovo programma siano realizzati da due chiamate distinte, un comportamento atipico se confrontato con quello di altri sistemi.
- L'utilizzo di dieci bit di protezione per l'accesso in lettura, scrittura ed esecuzione dei file. I primi nove bit sono utilizzati, a gruppi di tre, per segnalare i permessi concessi al possessore del file (*owner*), al gruppo al quale esso appartiene (*group*) e agli utenti rimanenti (*other*). L'ultimo bit, chiamato *set-user-ID bit*, è utilizzato nei file eseguibili per rendere le funzionalità di un programma privilegiato accessibili ad altri utenti. Questo bit è spesso abilitato per programmi dediti alla gestione delle impostazioni del sistema (e.g. *passwd* per il cambio della password).
- Un super utente, tradizionalmente chiamato *root*, che possiede il massimo controllo sulla macchina e che si contrappone al normale utente non privilegiato. Per avere una gestione intermedia dei privilegi in un sistema UNIX tradizionale occorre fare un buon utilizzo dei bit di protezione dei file descritti al punto precedente.

Nel 1974 UNIX godeva ormai di una certa stabilità: oltre a contare degli utilizzi amministrativi all'interno dei laboratori Bell si era guadagnato anche la fama di essere un sistema semplice, elegante ed efficiente [19]. Questa maturazione colse l'attenzione dell'AT&T e riaccese l'interesse dell'azienda nell'offrire un sistema operativo commerciale al pubblico, in particolare alle università, con tanto di sorgenti al costo nominale di una licenza di utilizzo. Questa diffusione su larga scala ebbe due grandi risvolti sulla comunità informatica dell'epoca: da un lato creò la domanda per sistemi UNIX, o simili, sul mercato; dall'altro permise ai dipartimenti informatici delle università di poter studiare e modificare il codice di un vero sistema operativo.

Per ulteriori informazioni dello sviluppo di UNIX negli anni '70 si rimanda a [18].

### 1.1.1 La nascita di nuove varianti

La diffusione dei sorgenti di UNIX all'interno del mondo accademico diede spazio al codice di poter migliorare e includere nuove funzionalità, nate da individui e necessità differenti. Quando queste funzionalità raggiungono un livello notevole, è possibile parlare di una variante di UNIX. Di questi derivati dal sistema dei Bell Labs ce ne sono due di notevole importanza con i quali Linux mantiene tuttora un'elevata compatibilità: BSD e System V [12].

#### *BSD*

Il Berkeley Software Distribution (BSD) è una variante di UNIX sviluppata da Thompson e da un gruppo di studenti presso l'università di Berkeley nella seconda metà degli anni '70. Tra le modifiche più significative troviamo l'editor di testo *vi*; il Berkeley Fast File System (BFFS); e, rilasciata nel 1983, una completa implementazione dello stack dei protocolli TCP/IP. BSD venne distribuito assieme ai propri

codici sorgenti e si diffuse molto in ambito accademico, ricevendo ulteriori modifiche e miglioramenti indipendenti al punto di poter parlare di una famiglia di sistemi BSD.

#### *System V*

Lo System V dello UNIX Support Group (USG) di proprietà dell'AT&T è una versione commerciale di UNIX rilasciata nel 1983 largamente diffusa nel mondo aziendale.* In questo sistema troviamo i primi meccanismi avanzati per la Inter-Process Communication, in particolare: *message queues* e *memoria condivisa* per la comunicazione dei processi; *semafori* per la loro sincronizzazione. System V era disponibile per una grande varietà di architetture hardware e a costi moderati, favorendolo nella competizione con altri sistemi già presenti nel mercato.

### 1.1.2 Gli standard

La presenza di numerose versioni di UNIX, tutte quasi compatibili tra loro ma con le loro differenze caratteristiche, creò la necessità di una standardizzazione di questa grande famiglia di sistemi operativi. Verso la fine degli anni '80 nacquero due standard: Portable Operating System Interface (POSIX) e la Single UNIX Specification (SUS). Il primo è un gruppo di standard sviluppato dall'Institute of Electrical and Electronic Engineers (IEEE) che definisce l'insieme di interfacce software che un sistema operativo deve offrire in modo da garantire una compatibilità tra i programmi al livello di codice sorgente. Il secondo è invece una specifica che caratterizza un sistema come UNIX, fortemente basato su POSIX. Al giorno d'oggi, POSIX e SUS sono riuniti nella SUSv4 e sono ancora un importante riferimento per lo sviluppo di nuovi sistemi UNIX e UNIX-like, Linux compreso.

## 1.2 GNU E IL MOVIMENTO PER IL SOFTWARE LIBERO

Nel 1985, Richard Stallman, un abile programmatore che aveva lavorato presso il MIT, pubblicò un manifesto che rivoluzionò il modo di pensare al software: *Il manifesto GNU* [20]. Secondo Stallman, ogni utente deve avere il diritto di poter comprendere, modificare e condividere il software che utilizza: in questo modo si va a creare un ambiente collaborativo, e non competitivo, per i programmatori che scrivono il codice ed è l'utente ad essere effettivamente in controllo del proprio software, non il contrario. Un software che rispetta le libertà dell'utente viene chiamato *software libero*† (in opposizione al software proprietario) e nell'arco del tempo sono state caratterizzate quattro libertà essenziali che lo definiscono (numerate da zero a tre) [22]:

**0** La libertà di eseguire il programma come si desidera, per qualsiasi scopo.

**1** La libertà di studiare come funziona il programma e di cambiarlo per adattarlo alle proprie necessità.

**2** La libertà di distribuire delle copie del software originale per aiutare gli altri.

---

* Il motivo per il quale ci è voluto così tanto per una versione commerciale di UNIX è per questioni di antitrust. L'AT&T possedeva il totale controllo delle telecomunicazioni americane e vi erano accordi con il governo che limitavano l'approdo dell'azienda all'interno di altri mercati. Nel 1982, l'AT&T venne divisa dai propri laboratori di ricerca, così facendo perse il monopolio sulle telecomunicazioni ma si aprirono le porte al mondo dei sistemi operativi.

† In inglese, *free* è un termine ambiguo in quanto può significare sia *gratuito* che *libero*. Purtroppo questa ambiguità è sufficiente a confondere, anche all'interno della stessa comunità informatica, le questioni mosse da Stallman come questioni economiche invece che morali. Per chiarire ogni dubbio è diventata celebre la seguente frase: *"Free as in freedom, not as in beer"*.

3 La libertà di distribuire delle copie del software contenente le proprie modifiche per dare la possibilità alla comunità di poter beneficiare dai cambiamenti eseguiti.

Nell'ottobre di quello stesso anno, venne fondata la Free Software Foundation (FSF), presieduta da Stallman, e si diede il via ad un movimento informatico per la creazione e diffusione di software libero.

Il manifesto GNU è stata anche un'ottima occasione per Stallman per chiamare a sé programmatori e aziende intenzionate a supportarlo nella creazione del suo ultimo progetto: GNU is Not UNIX (GNU). GNU è un sistema operativo concettualmente basato su UNIX completamente libero e rappresenta il primo passo in avanti per il movimento. È importante notare come GNU non sia un diretto derivato da UNIX, bensì si tratti di uno UNIX-like: il codice sorgente è stato scritto da zero e alcuni principi architetturali sono completamente differenti dall'originale (in particolare il kernel che sarà descritto a breve). Nel 1990 era quasi stato completato e poteva contare strumenti come l'editor testuale *Emacs*; il compilatore C *gcc*; la shell interattiva *bash*; e la libreria *glibc*, per nominarne alcuni. Nel 1991 si cominciò a scrivere anche il kernel di GNU, chiamato HURD: un microkernel basato su Mach considerato architetturalmente superiore ad un design monolitico. Purtroppo, questa superiorità architetturale viene pagata tuttora come complessità nello sviluppo del codice, con la prima versione stabile di HURD rilasciata, per soli processori i386, nel 2013.

## 1.3 LINUX E IL SISTEMA GNU/LINUX

Inizialmente, Linux era un progetto sviluppato da Linus Torvalds, allora studente presso l'università di Helsinki, fortemente basato su di una variante di UNIX largamente diffusa per scopi didattici: Minix del professor Andrew Tanenbaum. Nell'ottobre del 1991, Torvalds annunciò la versione 0.02 del proprio kernel: una versione stabile (anche se non stand-alone) in grado di compilare ed eseguire numerose utilità del progetto GNU. Seppur il progetto non rappresentava nulla di grosso (a detta del suo autore), molti programmatori videro del potenziale nel lavoro di Torvalds e si unirono a lui nel migliorare ed estendere le funzionalità del kernel. Tra questi, il kernel attirò anche l'attenzione della FSF e, visto che HURD era distante dall'essere stabile, cominciarono i primi sforzi per integrare Linux con il progetto GNU, dando vita al primo sistema operativo GNU/Linux.

Ad oggi, dopo più di trent'anni di costante sviluppo, il kernel Linux risulta essere dotato di tutte le funzionalità richieste da un moderno sistema general purpose per architetture multiprocessore, usufruendo dei benefici lasciati in eredità dal sistema UNIX di Thompson. La sua struttura modulare permette di estendere le funzionalità del kernel a run-time e, dove un modulo non dovesse essere sufficiente, è sempre possibile configurare e installare versioni modificate per andare incontro alle proprie necessità.

Da un punto di vista di sicurezza, Linux migiora la suddivisione binaria dei privilegi dei sistemi UNIX tradizionali andando ad introdurre, dalla versione 2.2, il concetto di *capabilities*. Una capability è un permesso che specifica una funzionalità privilegiata che un utente può eseguire, e il super utente è l'unico a possedere tutte le capabilities disponibili nel sistema. Questo meccanismo permette di avere una maggiore granularità nella gestione dei privilegi e ne facilita la loro gestione.

Un altro grande miglioramento, che sfrutta l'architettura modulare del kernel, è quello dell'introduzione del framework Linux Security Module (LSM) [23]. Questo meccanismo introduce nel kernel delle funzioni di aggancio (*hook*) posizionate nei punti più critici e sensibili alla sicurezza del kernel, spesso ritrovabili nel passaggio tra user space e kernel space nelle chiamate di sistema, ma non solo. A queste funzioni si agganciano dei moduli esterni al kernel per l'esecuzione dei controlli contro

possibili attacchi definiti sotto forma di controllo degli accessi. Questa struttura bipartita, in contrapposizione ad una logica integrata nel kernel, favorisce lo sviluppo di diversi moduli di sicurezza, ognuno ideato per andare incontro alle necessità di un determinato gruppo di utenti. Tra questi moduli di sicurezza possiamo trovare SELinux, AppArmor, Smack e TOMOYO, per citarne alcuni. Oltre a questi sistemi di sicurezza di alto livello, il kernel offre numerose tecniche per individuare e contrastare la presenza di vulnerabilità di memoria a basso livello.

Tutti questi meccanismi di sicurezza, performance e debugging sono individuabili nei file di configurazione del kernel, chiamati *Kconfig*, che vengono letti e interpretati durante la compilazione del sistema tramite il programma *make*, lo strumento standard per la configurazione, compilazione e installazione di un kernel Linux.

### 1.3.1 Standard UNIX e Linux

Al momento, Linux non soddisfa ufficialmente la SUSv4. Tuttavia, l'intero sviluppo del kernel è incentrato sullo standard POSIX e si è generalmente d'accordo che Linux soddisfi gli standard UNIX *de facto* [2]. Il kernel presenta inoltre estensioni funzionali che lo rendono parzialmente compatibile con sistemi BSD e System V, oltre ad altre specifiche del sistema GNU.

In aggiunta agli standard UNIX, nel 2001 venne rilasciata la Linux Standard Base, un insieme di specifiche che garantiscono una compatibilità al livello di codice binario tra sistemi Linux e alla quale molte distribuzioni aderiscono.

# 2 GESTIONE DELLA MEMORIA

In generale, ogni programma possiede due aree dedicate alla gestione della memoria dinamica: uno stack dedicato all'allocazione delle variabili automatiche; e uno heap per la per la gestione di blocchi di oggetti di diversa dimensione. Sotto questo aspetto, il kernel Linux struttura la propria memoria in modo analogo ai programmi in user space. Tuttavia, vi sono delle grosse differenze e limitazioni nella gestione di queste due aree che rendono la programmazione del kernel diversa da quello che ci si potrebbe aspettare normalmente.

Di seguito si andranno a delineare le forti limitazioni legate all'uso dello stack del kernel Linux e andranno descritti i diversi modi per la gestione della memoria, dalla gestione delle pagine ai blocchi di oggetti. Si presterà una particolare attenzione ad alcuni dei meccanismi di difesa e di debug impiegati per mitigare o individuare la presenza di errori, con una particolare attenzione posta sui canarini dello stack.

## 2.1 GLI STACK DEL KERNEL

Lo stack è una zona di memoria dedicata all'allocazione delle variabili locali di una funzione in *stack frames* che vengono gestiti tramite operazioni di *pop* e di *push*. Nel kernel sono presenti numerosi stack durante la sua esecuzione: uno per ogni processo attivo nel sistema e uno per ogni CPU. I primi sono gli stack utilizzati dal kernel per gestire l'invocazione di una chiamata di sistema e sono vuoti quando un processo compie operazioni in user space. Invece, per quanto riguarda gli stack delle CPU, essi sono dedicati alla gestione di ogni interrupt handler che quel processore è in grado di ricevere. Al contrario dello user space, la dimensione di questi stack è fissata in fase di compilazione e può variare tra i 4KB e i 16KB a seconda dell'architettura. Inoltre, gli stack associati ad un processo memorizzano le informazioni principali riguardanti il suo contesto tramite la struttura *thread_info* posizionata subito dopo la loro fine. Grazie alla loro dimensione fissa e a questa posizione nota, i calcoli per raggiungere questa struttura si fanno più semplici, ma si aprono delle nuove vulnerabilità quando lo stack cresce eccessivamente rischiando di andare a corrompere silenziosamente le strutture dati del kernel. Questo tipo di vulnerabilità prende il nome di *stack depth overflow* ed è un esempio di problematica specifica del kernel Linux della quale non si trovano equivalenti nello user space. Nella scrittura di codice lato kernel lo stack viene considerato come una risorsa molto limitata: allocazione di array e ricorsione, specie se profonda, sono considerate come delle cattive pratiche di programmazione ed è bene che la somma della dimensione di tutte le variabili locali di una funzione si mantenga al di sotto delle centinaia di byte [14].

Storicamente, lo stack ha rappresentato un punto debole per la difesa di un programma e nella sicurezza informatica esistono numerosi attacchi mirati a quest'area di memoria. Il motivo di tanto interesse è dovuto al fatto che lo stack rappresenta una zona di memoria facile da raggiungere, dal comportamento prevedibile, che contiene puntatori alla sezione di codice di un programma, meglio noti come *return address*. Uno degli attacchi più semplici, quanto versatili, è quello dello *stack smashing* (distruzione dello stack) [4]. Nello stack smashing si va a scrivere al di fuori dei limiti di un buffer sullo stack in modo da corrompere lo stack frame corrente. Nei casi meno articolati, un programma attaccato tende ad andare in crash o ad ave-

re un comportamento indefinito, mentre nelle situazioni più gravi è possibile andare a sovrascrivere il return address più vicino per farlo puntare a codice controllato dall'attaccante. Questo ultimo caso, dove il flusso di esecuzione di un programma viene dirottato da un attaccante, rappresenta lo scenario peggiore nell'attacco ad un kernel poiché lascia ampio controllo sul come compromettere il sistema. Il payload di uno stack smashing può essere memorizzato sullo stesso stack compromesso, su blocchi di memoria heap oppure, per attacchi al kernel, in un'area di memoria marcata come eseguibile in user space.

Un altro attacco che fa leva sullo stack per dirottare il flusso di esecuzione del programma è la Return Oriented Programming (ROP), originariamente nota come Borrowed Code Chunk Exploitation [13]. ROP nasce per aggirare Write Xor Execute (W^X), una delle prime difese contro la famiglia di attacchi precedenti introdotta per la prima volta nel sistema operativo OpenBSD 3.3 [15], che rende la possibilità di scrivere su una pagina di memoria (*W*) e di poter eseguire il suo contenuto (*X*) operazioni mutualmente esclusive. L'idea alla base di ROP è quella di concatenare piccoli frammenti del codice attaccato (chiamati *gadget*), ognuno in grado di fare piccole operazioni elementari, per far emergere un nuovo codice complessivo controllato dall'attaccante. Supponendo che il codice attaccato sia abbastanza grande, non ci sono limitazioni imposte da questa tecnica di programmazione rispetto ad una più tradizionale, anche se, nella pratica, è più plausibile che la programmazione ROP venga utilizzata per ottenere un maggiore controllo sul sistema ottenendo i privilegi di super utente.

Ad oggi, il metodo migliore per difendere il kernel dagli attacchi precedenti consiste ancora nell'eseguire un corretto controllo dei limiti dei buffer di memoria utilizzati e di non fidarsi di ciò che può arrivare dallo user space. Nel kernel sono presenti due funzioni che racchiudono al loro interno la logica di questi controlli, chiamate *copy_from_user* e *copy_to_user*, spesso utilizzate all'interno di moduli e chiamate di sistema. Il loro compito è quello di trasferire una certa quantità di dati da e verso lo user space verificando che gli indirizzi di memoria appartengano al processo chiamante e che gli opportuni permessi di scrittura o lettura (dove necessari) siano abilitati. Una volta utilizzate, è possibile verificare se il valore di ritorno, che rappresenta il numero di byte trasferiti, coincida con quello atteso per individuare e gestire eventuali errori.

Tuttavia, nel momento in cui queste funzioni vengano mal utilizzate o non vengano impiegate affatto, esiste comunque un ulteriore livello di sicurezza offerto da numerose mitigazioni con lo scopo di rendere estremamente più complesso lo sfruttamento di una vulnerabilità da parte di un attaccante. Tra queste mitigazioni ve ne sono alcune che basano il loro funzionamento sull'introduzione di entropia all'interno del sistema, come è il caso della Kernel Address Layout Randomization (KASLR) che somma un offset generato casualmente agli indirizzi di ogni stack del kernel ad ogni avvio del sistema. Altri meccanismi fanno invece uso di memoria addizionale per eseguire dei controlli di ridondanza. Una delle mitigazioni più note ed utilizzate per la difesa specifica degli stack, sia del kernel che in user space, consiste nell'impiego dei canarini dello stack.

### 2.1.1 I Canarini dello stack

I canarini dello stack (*stack canaries*), noti anche con il nome di *stack protector* oppure *stack cookies*, sono una mitigazione introdotta in fase di compilazione che fa la sua prima comparsa nel 1998 come una patch di sicurezza al compilatore gcc, chiamata *StackGuard* [6]. Un canarino è un valore generato casualmente grande quanto una parola di sistema che viene controllato sistematicamente per determinare la presenza di corruzioni di memoria. Il suo principio di funzionamento è semplice: se il valore del canarino dovesse cambiare rispetto a quello originale durante l'esecuzione di un programma, allora è possibile individuare e gestire una corruzione di

memoria.* Nel kernel ogni stack ha un suo canarino ad esso associato ed è estremamente improbabile che due o più stack condividano canarini dallo stesso valore. I canarini per un processo vengono generati durante la sua creazione nella funzione *dup_task_struct* con la chiamata a *get_random_canary* e vengono memorizzati all'interno del membro *stack_canary* della struttura *task_struct*. Per quanto riguarda gli interrupt handler, i loro canarini vengono inizializzati in fase di avvio del kernel, dalla funzione *boot_init_stack_canary* e vengono memorizzati nel contesto del processo *idle*: un processo critico del sistema, non terminabile, vincolato ad eseguire sul processore su cui è stato inizialmente creato. Ogni volta che un processo (o un interrupt) prende il controllo di una CPU, il suo canarino viene copiato dal suo contesto (o dal processo idle di quella CPU) e memorizzato tra le variabili del processore, facilmente accessibili in Linux tramite l'interfaccia dedicata *percpu*.

Durante l'utilizzo dello stack, le funzioni protette da questo meccanismo di difesa possiedono del codice aggiuntivo nel loro prologo ed epilogo che esegue le seguenti azioni:

- Nel prologo: viene prelevato il valore del canarino dalle variabili del processore e viene posizionato tra il return address memorizzato e le variabili locali del frame.
- Nell'epilogo: si confronta il valore memorizzato nello stack con quello del processore.

Il codice che implementa queste due funzionalità viene inserito automaticamente dal compilatore ed è composto da poche righe di codice assembly. La determinazione dell'eventuale presenza di corruzioni di memoria avviene nell'epilogo e l'utilità di questo controllo si manifesta una volta considerata la natura di un normale buffer overflow. In generale, un buffer overflow corrompe la memoria in modo contiguo, per una certa quantità di byte nota, senza conoscere i valori memorizzati nelle variabili compromesse. Ponendo un valore casuale tra il buffer e il return address si va a complicare la situazione per un attaccante obbligandolo a dover ottenere, come informazione aggiuntiva, il valore esatto del canarino per poter manipolare il flusso di esecuzione del programma. Nel caso il canarino risulti corrotto, il codice inserito dal compilatore invoca una funzione dedicata, chiamata *__stack_chk_fail* che gestisce una corruzione di memoria. Nei programmi in user space, questa funzione non fa altro che chiedere al sistema operativo di terminare il processo corrente, in quanto il flusso di esecuzione potrebbe essere compromesso e non ci sono modi per ristabilirlo. Analogamente, nel kernel viene offerta una nuova definizione di questa funzione che forza la terminazione del sistema tramite un *kernel panic* (la versione Linux del *Blue Screen of Death* del sistema operativo Windows). Questo comportamento estremo permette di rendere vano ogni tentativo a forza bruta per individuare il valore del canarino, ma non risolve tutti quegli attacchi dedicati a minare la disponibilità di un sistema mandandolo in crash. Sotto questo aspetto si può solo constatare che i canarini difendono il kernel, e un programma in generale, dal peggiore dei casi: l'esecuzione di codice controllato da un attaccante.

Per quanto riguarda l'overhead causato dall'utilizzo dei canarini, esso tende ad essere moderato: il numero di istruzioni assembly totale (nel prologo ed epilogo) non supera la decina e rimane fisso per ogni funzione protetta. In [6] viene delineato un upperbound ottenuto analizzando funzioni estremamente elementari e, a detta degli autori, per funzioni più complesse e realistiche, dove il codice assembly originale è in netta maggioranza rispetto a quello aggiunto, il costo dei canarini dovrebbe essere notevolmente ammortizzato. In uno studio più recente [7] viene

* Il nome *canarino* deriva dall'impiego di veri canarini nelle miniere durante il Novecento per la determinazione di fughe di gas tossico inodore e incolore. La presenza di una minima fuga di gas avrebbe portato alla morte l'animale molto prima di danneggiare la salute dell'uomo, permettendo ai minatori di evacuare la galleria compromessa in tempo.

analizzato il costo di implementazioni differenti di un'altra tecnica di difesa, lo *shadow stack*, e viene studiato, per completezza, l'overhead temporale causato dai canarini dello stack implementati dal compilatore *gcc*, trovando un overhead medio del 2.54%.

Per concludere questa discussione sui canarini è opportuno accennare al fatto che non tutte le funzioni necessitano di uno stack protetto. I compilatori moderni tendono ad utilizzare approcci euristici per determinare quali funzioni proteggere e quali invece mantenere intatte. I dettagli possono variare da compilatore a compilatore, ma gcc offre due modalità. La prima modalità, considerata di default in fase di compilazione, va a difendere gli stack delle sole funzioni che contengono array di caratteri con dimensioni superiori agli 8 byte. Questo approccio è uno dei più deboli in quanto ricerca solamente funzioni che potrebbero avere una cattiva gestione di stringhe. Per quanto sia vero che molti buffer overflow nascano dal problema precedente, e forse ne possono anche essere la maggioranza, questo metodo esclude tutti gli altri possibili casi meno comuni a priori e dunque non garantisce la completa difesa di un programma. Il secondo metodo, attivabile con l'opzione *-fstack-protector-strong*, esegue una ricerca più approfondita andando ad aggiungere la logica di protezione ad ogni funzione che possieda un array o che utilizzi almeno una delle sue variabili locali come destinazione per un assegnamento. Quest'ultimo metodo è il più sicuro in quanto garantisce di andare a coprire l'intera superficie d'attacco escludendo le funzioni che certamente non sono vulnerabili. L'utilizzo di uno dei due metodi (o di nessuno) è controllato nei file di configurazione Kconfig del kernel, nelle opzioni dai nomi, rispettivamente, *Stack protector buffer overflow detection* e *Strong Stack Protector*. Siccome i canarini necessitano di registri dedicati in un processore, queste configurazioni non sono tra le opzioni di sicurezza ma risultano essere collocate tra quelle dipendenti dall'architettura.

## 2.2 ALLOCAZIONE DELLA MEMORIA DINAMICA

Ci sono altri metodi all'interno del kernel per prendere il possesso della memoria durante l'esecuzione del sistema operativo. Generalmente questi approcci vengono definiti dinamici, in quanto soltanto durante l'esecuzione del sistema è possibile determinare con esattezza la quantità di memoria utilizzata e si contrappongono alle zone statiche (e.g. le sezioni per le variabili globali e i vari stack) di dimensione prestabilita in fase di compilazione. Gli allocatori sono l'insieme di funzioni dedicate alla gestione della memoria dinamica e, di principio, sono sempre formati da una o più coppie di funzioni dalla natura duale: una metà dedicata alla suddivisione della memoria (*alloc*) e un'altra per la sua liberazione (*free*).

Nelle moderne architetture le pagine formano l'unità atomica per la suddivisione della memoria e, generalmente, la loro grandezza rientra nell'ordine dei 4KB e 8KB. Nel kernel è presente un allocatore di basso livello, chiamato *buddy*, che restituisce all'utilizzatore un insieme di pagine libere specificando degli opportuni flag GFP (Get Free Pages) alle funzioni *alloc_pages* e *get_free_pages*. L'utilizzo di questi flag permette di influenzare il comportamento delle funzioni a seconda del tipo di memoria richiesta, ad esempio: *GFP_KERNEL* è utilizzato per ottenere memoria per il kernel; *GFP_DMA* per l'I/O; e *GFP_USER* per ricavare memoria per conto dello user space. Per evitare il rilascio involontario di informazioni sensibili del kernel allo user space, la memoria marcata *GFP_USER* è l'unica ad essere azzerata durante l'allocazione, mentre per ottenere un comportamento simile con la memoria del kernel è responsabilità del programmatore specificare ulteriormente il flag *__GFP_ZERO*. In genere, questo allocatore non è utilizzato direttamente nel codice ma forma la base per altre due famiglie di allocatori più sofisticati: la famiglia di allocatori SLAB e quelli malloc-like.

### 2.2.1 La famiglia di allocatori SLAB

La famiglia di allocatori SLAB (dall'inglese *slab*: lastra) è un insieme di funzioni, dall'interfaccia simile, che promuove il riutilizzo della memoria e ne riduce la sua frammentazione interna allocando oggetti della stessa dimensione all'interno di lastre di memoria. L'allocatore SLAB, dalla quale la famiglia prende il nome, è stato ideato per la prima volta nel 1994 per essere utilizzato nella gestione della memoria del kernel del sistema operativo SunOS della Sun Microsystem (ora Oracle) ed è stato pubblicamente documentato in [5]. Linux offre la sua implementazione dell'allocatore SLAB; una sua variante, chiamata SLUB, che riduce l'impatto sulla memoria; ed infine un allocatore semplificato basato sull'implementazione di Kernighan e Ritchie [11] ritrovabile nei sistemi UNIX tradizionali. Durante la configurazione del kernel Linux è possibile scegliere soltanto una delle tre implementazioni possibili degli allocatori della memoria a lastre, con SLOB ormai caduto in disuso e relegato a rari usi nei dispositivi embedded, mentre SLUB è ormai l'opzione di default nonostante non sia il più performante [9].

L'idea alla base di questi allocatori è quella di suddividere la memoria in cache (con l'accezione di deposito, non di memoria intermedia fra due entità) di oggetti dello stesso tipo (e dunque dimensione). Le cache sono a loro volta formate da un insieme di una o più pagine di memoria contigue, chiamate lastre, ed ogni lastra andrà a memorizzare un certo numero di oggetti. L'allocatore tiene traccia della quantità di oggetti in uso e suddivide le lastre in tre categorie: piene, vuote e parzialmente riempite. L'allocatore cercherà di riutilizzare lo spazio all'interno delle lastre parzialmente riempite per l'allocazione di nuovi oggetti e richiederà nuove pagine di memoria soltanto quando non è più possibile riciclare lo spazio. In condizioni di necessità, le lastre completamente libere possono essere rilasciate in modo da aumentare le risorse disponibili al resto del sistema.

Un'altra caratteristica in comune a questi allocatori è quella di memorizzare in una lista (chiamata *free list*) tutti i blocchi di memoria non utilizzati in una lastra parzialmente riempita, in modo da velocizzare l'allocazione di nuovi oggetti. Generalmente, la free list è allocata sulla stessa lastra ed utilizza gli spazi di memoria liberi per memorizzare i propri puntatori e contatori. Inoltre, la collocazione degli oggetti all'interno delle lastre viene eseguita prestando attenzione al loro allineamento con i blocchi delle cache hardware, andando a riempire lo spazio tra un oggetto e l'altro con dei byte di padding ed utilizzando un meccanismo di colorazione delle cache, qui non descritto, per limitare i fenomeni di trashing degli oggetti.

Per creare nuove cache di oggetti ed ottenere un riferimento alla loro struttura è possibile utilizzare la funzione *kmem_cache_create*, dove viene specificato il nome della cache e la dimensione del tipo di oggetti che andrà a contenere. Il puntatore ottenuto potrà poi essere utilizzato in *kmem_cache_alloc* per l'allocazione di nuovi oggetti. Analogamente, i duali delle due funzioni precedenti sono *kmem_cache_destroy* e *kmem_cache_free*. È interessante osservare che in fase di creazione di una cache è possibile specificare un puntatore ad una funzione di costruzione denominata *ctor*. Questo costruttore, se non è nullo, viene sempre invocato durante l'allocazione di un oggetto e garantisce un'appropriata inizializzazione della sua memoria. Sfortunatamente, secondo Love nel suo libro sullo sviluppo del kernel Linux [14], questo parametro è raramente utilizzato.

Infine, gli allocatori SLAB e SLUB sono progettati per agevolare l'individuazione degli errori nella gestione della memoria manipolando le lastre allocate. Queste tecniche non sono propriamente dei meccanismi di sicurezza del kernel ma alcune di loro sono impiegate per determinare la presenza di corruzioni della memoria e dunque svolgono comunque un ruolo, seppur minore, nel rendere il kernel più sicuro. Per SLAB è necessario ricompilare il kernel, abilitando *SLAB_DEBUG*, per attivare queste impostazioni di configurazione, mentre per SLUB basta utilizzare *slub_debug* come argomento da riga di comando. SLUB offre inoltre un grado maggiore di libertà in quanto può tracciare anche cache di oggetti specifiche, permettendo ad

**Figura 1:** cache utilizzate dall'allocatore *kmalloc* del kernel

```
kmalloc-8k           341    352   8192    4    8 : tunables    0    0    0 : slabdata     88     88      0
kmalloc-4k           640    656   4096    8    8 : tunables    0    0    0 : slabdata     82     82      0
kmalloc-2k          2870   2912   2048   16    8 : tunables    0    0    0 : slabdata    182    182      0
kmalloc-1k          2213   2304   1024   32    8 : tunables    0    0    0 : slabdata     72     72      0
kmalloc-512         3831   3904    512   32    4 : tunables    0    0    0 : slabdata    122    122      0
kmalloc-256         3629   3968    256   32    2 : tunables    0    0    0 : slabdata    124    124      0
kmalloc-192         2058   2058    192   21    1 : tunables    0    0    0 : slabdata     98     98      0
kmalloc-128         1504   1504    128   32    1 : tunables    0    0    0 : slabdata     47     47      0
kmalloc-96          2982   2982     96   42    1 : tunables    0    0    0 : slabdata     71     71      0
kmalloc-64         15787  16832     64   64    1 : tunables    0    0    0 : slabdata    263    263      0
kmalloc-32         12800  12800     32  128    1 : tunables    0    0    0 : slabdata    100    100      0
kmalloc-16         15938  17408     16  256    1 : tunables    0    0    0 : slabdata     68     68      0
kmalloc-8          18432  18432      8  512    1 : tunables    0    0    0 : slabdata     36     36      0
```

un suo utilizzatore di andare a ridurre il campo di ricerca per la determinazione di errori. Tra questi meccanismi di debug ve ne sono due di maggiore interesse:

**SLAB POISONING:** la memoria degli oggetti di una cache viene avvelenata con dei valori costanti ripetuti per tutta la lunghezza della sezione di memoria interessata. Tra i possibili valori troviamo *POISON_FREE* (0x6B) che viene controllato durante la fase di allocazione di un nuovo oggetto e permette di determinare vulnerabilità di tipo Use After Free e Double Free. Il secondo valore ben noto è *POISON_INUSE* (0x5A) utilizzato per marcare oggetti appena allocati e le sezioni di padding tra di loro. I controlli su quest'ultimo veleno vengono eseguiti solamente sul padding della cache e rientrano tra i controlli di integrità di una free list.

**RED ZONES:** sono dei valori grandi quanto una parola di sistema posti all'inizio e alla fine di un nodo in una cache. Queste zone permettono la determinazione di overflow e underflow degli oggetti e assumono due valori differenti a seconda che l'oggetto sia stato allocato (*SLUB_RED_ACTIVE*, 0xCC) o risulti libero (*SLUB_RED_INACTIVE*, 0xBB).

Nell'eventualità che uno di questi controlli dovesse fallire, il sistema non entra in uno stato di panico, come nel caso dei canarini dello stack, ma si limita a stampare un rapporto sullo stato della memoria corrotta (accessibile tramite il programma *dmesg*) e continua la sua normale esecuzione anche se potrebbe essere gravemente compromesso.

### 2.2.2 Gli allocatori malloc-like

La terza e ultima categoria di allocatori nel kernel Linux è quella degli allocatori malloc-like, chiamati così perché dotati di un'interfaccia intenzionalmente simile a quella messa a disposizione dalla libreria *libc* nello user space. Questi allocatori permettono di occupare blocchi di memoria di dimensione almeno uguale, spesso superiore, a quella richiesta e anch'essi sono influenzati nel comportamento dall'uso dei flag GFP. Anche queste funzioni sono molto utilizzate nel kernel e vanno a ricoprire i casi lasciati scoperti dagli strumenti precedenti: l'allocazione oggetti di dimensione generalmente inferiore ad una pagina nota durante l'esecuzione del sistema.

Il funzionamento di *kmalloc* e dei suoi simili si basa sulle cache di oggetti descritte precedentemente. In particolare, durante l'avvio del sistema vengono create delle cache del tipo *kmalloc-N*, dove N è la dimensione degli oggetti contenuti, utilizzando gli allocatori della famiglia SLAB. Quando un blocco di memoria deve essere allocato, non si fa altro che accedere alla più piccola cache in grado di ospitare il blocco e si alloca un nuovo oggetto. La figura 1 mostra un estratto del file di sistema */proc/slabinfo* (che contiene informazioni su tutte le cache attive nel sistema) e mette in evidenza le cache utilizzate dalle funzioni malloc-like.

# 3 HARDENING DELLA MEMORIA

In questo capitolo verrà considerata l'inizializzazione forzata della memoria: una tecnica molto efficace nel contrastare molte vulnerabilità legate ad un cattivo utilizzo della memoria ma tendenzialmente costosa. In Linux esistono due meccanismi complementari che vanno ad implementare questa tecnica e di entrambi verranno studiati funzionamento, overhead e interazione con altri meccanismi del kernel.

Infine, si andrà a descrivere la metodologia utilizzata negli esperimenti e verranno riportati i risultati ottenuti.

## 3.1 INIT_ON_ALLOC E INIT_ON_FREE

In Linux, l'inizializzazione forzata della memoria è affidata ad un paio di opzioni chiamate *init_on_alloc* e *init_on_free*. Queste due configurazioni obbligano gli allocatori del kernel ad inizializzare la memoria in fase, rispettivamente, di allocazione e liberazione, fatta eccezione per casi in cui sono definiti altri comportamenti (e.g. costruttori e avvelenamento della memoria). Queste due opzioni possono essere attivate sia in fase di configurazione del kernel (in *security/Kconfig.hardening*), che in fase di avvio del sistema tramite due opzioni da riga di comando omonime booleane [17]. All'interno del codice sorgente è possibile poi fare riferimento alle macro *CONFIG_INIT_ON_ALLOC_DEFAULT_ON* e *CONFIG_INIT_ON_FREE_DEFAULT_ON* per verificare la loro abilitazione. Nonostante i nomi possano suggerire un'idea diversa, queste opzioni sono disabilitate di default nelle configurazioni per architetture Intel x86.

Prima dell'introduzione di queste due funzionalità, l'unico controllo eseguito dal kernel per la pulizia della memoria era la presenza del flag *__GFP_ZERO* in fase di allocazione, mentre non esistevano meccanismi corrispondenti nella fase di liberazione. Con la loro introduzione, e la progettazione di altri meccanismi analoghi, la complessità delle condizioni per l'inizializzazione è stata spostata all'interno di quattro nuove funzioni booleane: due per l'allocatore delle pagine (*want_init_on_alloc, want_init_on_free*) e due per gli allocatori della famiglia SLAB (*slab_want_init_on_alloc, slab_want_init_on_free*). Le condizioni più complesse sono racchiuse nelle funzioni booleane per gli allocatori SLAB, in particolare nel codice riservato all'allocazione.

**Listing 1:** slab_want_init_on_alloc

```
1  static inline bool slab_want_init_on_alloc(gfp_t flags,
2                                              struct kmem_cache *c)
3  {
4      if (static_branch_maybe(CONFIG_INIT_ON_ALLOC_DEFAULT_ON,
5                  &init_on_alloc)) {
6          if (c->ctor)
7              return false;
8          if (c->flags & (SLAB_TYPESAFE_BY_RCU |
9                          SLAB_POISON))
10             return flags & __GFP_ZERO;
11         return true;
12     }
13     return flags & __GFP_ZERO;
```

```
14 }
```

Dal codice in 1 possiamo notare come l'inizializzazione forzata della memoria in fase di allocazione sia rifiutata nei casi in cui venga fatto uso di memoria avvelenata oppure sia disponibile un costruttore. Questo permette al meccanismo di integrarsi con il resto del sistema e di non intralciare eventuali condizioni di debug.

Una volta determinata la necessità di inizializzare la memoria, viene eseguito un un azzeramento tramite *memset* su tutto l'oggetto da allocare o liberare. La logica di cancellazione si può ritrovare nelle funzioni *slab_post_alloc_hook* e *slab_free_hook*. Come si può osservare in 2, l'array di puntatori *p* rappresenta un gruppo di oggetti da allocare di una cache *s* comune. Sia nel codice in 2 che in 3 la variabile booleana *init* rappresenta il risultato dell'invocazione delle funzioni booleane descritte al paragrafo precedente.

Un ulteriore caso particolare è rappresentato dall'abilitazione di un altro meccanismo di difesa del sistema chiamato Kernel Address Sanitizer (KASAN) [21]. Lo scopo di KASAN è quello di sanificare e tenere traccia delle inizializzazioni della memoria utilizzando dell'hardware dedicato. Le modalità in cui KASAN può operare sono tre: generica; software tag-based; e hardware tag-based. Al momento, soltanto le architetture arm64 con tecnologia Memory Tagging Extension (MTE) sono in grado di supportare le ultime due modalità. La modalità generica è invece indipendente dall'architettura, tuttavia è caratterizzata da un forte overhead ed è pensata per essere utilizzata solo in condizioni di debug di un sistema. Quando sia KASAN che le funzionalità *init_on_alloc* e *init_on_free* sono abilitate, il kernel favorisce le funzioni di azzeramento integrate di KASAN poiché modificano anche i tag utilizzati dal programma per tenere traccia delle zone di memoria occupate e libere.

**Listing 2:** Frammento di slab_post_alloc_hook

```
1 for (i = 0; i < size; i++) {
2     p[i] = kasan_slab_alloc(s, p[i], flags, init);
3     if (p[i] && init && !kasan_has_integrated_init())
4         memset(p[i], 0, s->object_size);
5     kmemleak_alloc_recursive(p[i], s->object_size, 1,
6                              s->flags, flags);
7 }
```

Per quanto riguarda il rilascio della memoria, i *memset* utilizzati sono due: il primo per la pulizia dell'oggetto appena liberato e il secondo per la cancellazione dei metadati associati, posti ad una certa distanza fissa dall'oggetto. Sempre per non compromettere eventuali situazioni di debug, il kernel è in grado di determinare la presenza di red zones e di preservarle.

**Listing 3:** Frammento di slab_free_hook

```
1 if (init) {
2     int rsize;
3
4     if (!kasan_has_integrated_init())
5         memset(kasan_reset_tag(x), 0, s->object_size);
6     rsize = (s->flags & SLAB_RED_ZONE) ? s->red_left_pad : 0;
7     memset((char *)kasan_reset_tag(x) + s->inuse, 0,
8            s->size - s->inuse - rsize);
9     }
```

Infine, è opportuno fare una precisazione sul rapporto tra *init_on_free* e gli allocatori SLAB. Come espresso precedentemente, gli allocatori della famiglia SLAB utilizzano lo spazio non utilizzato di una lastra per memorizzare le informazioni

associate alla free list degli oggetti. Quando *init_on_free* è abilitato, la gestione della free list viene spostata off-slab, ovvero al di fuori della lastra in uso e questo controllo viene eseguito ogni volta che una nuova cache viene creata all'interno della funzione *set_objfreelist_slab_cache*.

## 3.2 ANALISI DEI COSTI DI OVERHEAD

I costi di overhead causati dall'utilizzo delle due funzionalità in esame sono brevemente descritti all'interno del file di configurazione *security/Kconfig.hardening*. Per *init_on_alloc* viene descritto un overhead inferiore all'1%, con un picco al 7% per carichi di lavoro estremi; mentre per *init_on_free* si parla di un costo variabile tra il 3% e il 5% in condizioni normali, 8% per i carichi peggiori. Sfortunatamente, nel file non si accenna minimamente al procedimento eseguito per ottenere questi risultati, né viene indicato un qualche riferimento esterno da poter consultare. Questo è comprensibile visto che il testo indicato svolge il ruolo di presentazione delle funzionalità e dunque si pone enfasi sui risultati ottenuti, ma nega la possibilità di riprodurre gli esperimenti eseguiti. In un messaggio di commit di Alexander Potapenko [17] si possono trovare delle tempistiche associate all'inizializzazione forzata della memoria per due benchmark: la compilazione del kernel Linux e l'esecuzione del programma *hackbench*. Questi due carichi di lavoro sono abbastanza comuni nella comunità di sviluppatori del kernel e hanno degli utilizzi per l'analisi delle prestazioni nella gestione della memoria.

La compilazione del kernel Linux è un processo dispendioso, classificabile sia come memory-bound per la grande quantità di operazioni sul disco eseguite (prevalentemente letture da file) che CPU-bound per l'interpretazione e compilazione di grandi quantità di codice sorgente. Questo carico di lavoro è dunque adeguato a mettere in evidenza dei rallentamenti nel kernel space e può fungere da modello per processi che fanno pieno uso delle risorse del sistema.

Per quanto riguarda *hackbench*, esso non è uno strumento dedicato allo stress della memoria, almeno in linea di principio. Questo programma è di solito incluso in una suite di test per sistemi real time, nota come *rt-tests*, tuttavia possiede degli utilizzi anche nei sistemi general purpose in quanto è dedicato allo stress dello schedulatore di sistema. Nel suo funzionamento di base, *hackbench* procede a creare un certo numero di coppie di processi che comunicano tramite *socket* e misura il tempo complessivo impiegato nello scambio dei messaggi. I suoi utilizzi nell'analisi delle performance della memoria sono dovuti alla rapida creazione e distruzione di processi e sono in grado di dare un'idea sulla degradazione delle prestazioni dello schedulatore.

In questo lavoro, l'analisi dei costi di *init_on_alloc* e *init_on_free* è stata eseguita utilizzando due approcci differenti:

- Una *macro-benchmark* che ripete gli esperimenti eseguiti in [17].
- Una *micro-benchmark* che analizza i tempi di esecuzione delle singole chiamate a *kmalloc* e *kfree*. L'analisi è eseguita modificando il codice sorgente del kernel e introducendo una nuova chiamata di sistema.

Tutti gli esperimenti sono stati eseguiti durante l'avvio della macchina in modo tale da poter ritrovare il sistema in uno stato prevedibile e sono stati ripetuti per cinque volte. Per pianificare i test all'avvio è stato utilizzato il programma *cron*, spesso già installato in numerose distribuzioni GNU/Linux. Le funzionalità di hardening sono state controllate specificando le opportune opzioni da riga di comando durante il boot del kernel, come specificato in [17]. La modifica delle opzioni di boot del kernel Linux è un'operazione dipendente dal boot loader utilizzato. In questo caso, dove si fa uso di GRUB come boot loader di sistema, è possibile andare a modificare il file di configurazione *etc/default/grub* e aggiungere alla variabile

*GRUB_CMDLINE_LINUX_DEFAULT* le opzioni desiderate, per poi andare a eseguire lo script *update-grub* per fare in modo che le modifiche abbiano effetto. Una volta avviato un sistema GNU/Linux è possibile andare a leggere i contenuti del file di sistema */proc/cmdline* per poter constatare se le opzioni da riga di comando siano state specificate. In assenza di queste opzioni, il comportamento riguardante l'inizializzazione forzata della memoria dipende dalla configurazione originale del kernel.

Di seguito, verranno descritte in dettaglio le due metodologie.

### 3.2.1 Macro-benchmark

La compilazione del kernel Linux e l'esecuzione di *hackbench*, sebbene appartenenti alla stessa categoria, sono stati trattati come due esperimenti differenti.

Nel primo benchmark si va a compilare un kernel Linux con le minime impostazioni di default per architetture x86. Prima di ogni compilazione viene eseguito *make clean* per cancellare i file di supporto generati nella compilazione precedente. Per velocizzare la compilazione è stato fatto uso dell'opzione *-j8* per parallelizzare il lavoro tenendo conto del numero di processori disponibili nel sistema. Il tempo di esecuzione è stato misurato utilizzando il comando *time* che permette di specificare una stringa di formattazione per la stampa dei dati misurati, facilitando così la fase di analisi, e un file di output per il loro raccoglimento. Per gli utilizzatori della shell *bash* occorre fare una nota di attenzione: *bash* mette a disposizione un'omonima funzionalità molto più limitata che non permette di redirezionare il proprio output. Per evitare di utilizzare la funzionalità della shell occorre dunque specificare il percorso completo del programma per la sua invocazione (e.g. *usr/bin/time*). Il tempo di esecuzione di ogni compilazione si è mantenuto al di sotto dei 15 minuti.

Per quanto riguarda *hackbench*, esso è stato parametrizzato in modo tale da creare 15 coppie di processi e ogni coppia si scambia messaggi da 512 byte per 2000 volte su 25 socket per processo. L'invocazione completa del comando è la seguente:

```
hackbench --datasize 512 -f 25 -g 15 -l 2000
```

I tempi ottenuti si sono sempre mantenuti al di sotto dei 5 minuti per ogni esecuzione.

### 3.2.2 Micro-benchmark

Per misurare i tempi di esecuzione delle funzioni *kmalloc* e *kfree* in kernel space, per poi riportare questi tempi in user space, si è deciso di introdurre del nuovo codice all'interno del kernel sotto forma di una nuova chiamata di sistema dedicata. In termini di codice, una chiamata di sistema è una normale funzione C che deve essere segnalata andando a modificare degli opportuni file sorgente come riportato, in gran dettaglio, in [3]. I quattro passaggi fondamentali sono i seguenti:

1. Definire il prototipo della chiamata di sistema in *include/linux/syscalls.h*
2. Aggiungere una nuova entry ad un file di testo che funge da tabella delle chiamate di sistema. Questo passaggio è dipendente dall'architettura e potrebbe richiedere molteplici modifiche a seconda delle piattaforme che si desidera raggiungere. Per architetture x86, i file d'interesse sono *syscall_32.tbl*, per sistemi a 32 bit, e *syscall_64.tbl*, per processori a 64 bit, presenti entrambi nella cartella *arch/x86/entry/syscalls*.
3. Scrivere il codice della nuova chiamata di sistema all'interno dei file già presenti nel kernel oppure aggiungendone di nuovi.
4. Aggiungere una definizione di fallback in *kernel/sys_ni.c* in modo da andare a coprire le architetture che non implementano la nuova chiamata di sistema.

Il terzo punto è sicuramente quello più importante in quanto va a racchiudere il codice per l'esecuzione dell'esperimento e viene riportato in appendice a pagina 22. Il codice è stato aggiunto al file *kernel/sys.c*, dove sono presenti altre definizioni di chiamate di sistema, in modo da non dover mettere mano ai *Makefile* di supporto per la compilazione. La chiamata di sistema restituisce, sotto forma di *unsigned long*, il tempo di esecuzione di un ciclo di allocazioni (o liberazioni) della memoria in nanosecondi.

Una volta compilato ed avviato il kernel modificato è possibile scrivere un programma che invochi direttamente la nuova chiamata di sistema utilizzando la funzione *syscall* offerta dalla libreria *libc*. In fase di compilazione, *syscall* produce il codice assembly necessario per la corretta invocazione della chiamata di sistema desiderata; gli unici requisiti sono l'indice della funzione all'interno della tabella delle chiamate di sistema e i parametri da poi passare in kernel space. Anche il codice del programma per la raccolta dei dati può essere trovato in appendice a pagina 23.

Nella fase di test, la chiamata di sistema è stata parametrizzata in modo da allocare e liberare 1000 oggetti da 512 byte. L'invocazione è stata ripetuta per 10.000 volte. Come descritto nel capitolo precedente, *kmalloc* fa internamente uso degli allocatori a lastre e l'allocazione di oggetti di 512 byte permette di mettere sotto pressione la cache *kmalloc-512*.

In conclusione, è bene accennare al fatto che l'introduzione di una nuova chiamata di sistema non è l'unico modo per introdurre del nuovo codice nel kernel, né può sempre risultare essere il migliore. Uno dei punti di forza di Linux è quello di essere un kernel modulare e un altro approccio poteva essere quello di andare a scrivere un modulo di test dedicato. Inoltre, tutte le funzioni in esame risultano esportate al di fuori del kernel, e dunque possono essere utilizzate normalmente all'interno di un modulo, cosa non sempre possibile. Il motivo per il quale si è preferita una chiamata di sistema rispetto ad un modulo è stato di pura convenienza nella raccolta dei dati: anche un modulo può comunicare con lo user space ma i mezzi a disposizione noti (messaggi di log o creazione di file dedicati nel file system */proc*) sono di difficile automazione, rendendo la scelta più sfavorevole.

## 3.3 RISULTATI OTTENUTI

Di seguito si andranno a riportare i risultati ottenuti dall'esecuzione degli esperimenti descritti alla sezione precedente in forma tabellare. I valori d'interesse sono il valore massimo e minimo di esecuzione, la media delle cinque misurazioni, la deviazione standard associata e il costo di overhead. Per gli esperimenti di micro-benchmark, ognuno caratterizzato da una propria deviazione standard, verrà esposta una media rappresentativa delle deviazioni calcolate. La prima riga di ogni tabella va a rappresentare il caso base rispetto al quale fanno riferimento i costi di overhead stimati. I tempi misurati negli esperimenti di macro-benchmark sono espressi in secondi, mentre i valori riportati nelle tabelle di micro-benchmark sono dell'ordine dei nanosecondi.

### 3.3.1 Risultati per i test di macro-benchmark

I risultati ottenuti dall'esecuzione della compilazione di Linux evidenziano un peggioramento delle prestazioni dovute all'azzeramento della memoria liberata.

La tabella 1 mostra il tempo effettivamente trascorso dal compilatore in kernel space, noto anche come *system time*. Con l'opzione *init_on_alloc* abilitata, il kernel mantiene un overhead inferiore all'1%, come atteso; mentre l'opzione *init_on_free* è decisamente predominante in quanto raggiunge il 9%. Secondo il file di configurazione delle opzioni di hardening (*security/Kconfig.hardening*), i peggioramenti causati da *init_on_free* sono dovuti ad una cattiva gestione della cache hardware du-

rante la liberazione, fatto probabilmente attribuibile alla forzata gestione della lista di oggetti liberi off-slab.

| Opzioni attive | Massimo | Minimo | Media | Deviazione Standard | Overhead (%) |
|---|---|---|---|---|---|
| nessuna | 207,93 | 206,01 | 207,06 | 0,62 | - |
| *init_on_alloc* | 209,99 | 208,43 | 208,91 | 0,57 | 0,89% |
| *init_on_free* | 226,88 | 224,88 | 225,76 | 0,72 | 9,03% |
| *init_on_** | 229,48 | 226,48 | 228,45 | 1,07 | 10,33% |

**Tabella 1:** Tempi di compilazione del kernel Linux (*system time*)

Se confrontiamo invece i tempi totali trascorsi per la compilazione, riportati nella tabella 2, i costi di overhead risultano attutiti dai tempi di accesso al disco, con *init_on_alloc* praticamente trascurabile.

Si può dunque concludere che il caso peggiore per queste due opzioni di sicurezza è rappresentato da processi CPU-bound che fanno un uso intensivo della memoria del sistema. Numerosi componenti del kernel Linux rientrano al di sotto di questa categoria, ed è possibile aspettarsi una degradazione del servizio sensibile all'utente anche nel caso di ambienti desktop. Le esecuzioni di *hackbench*, mirate a mettere sotto sforzo lo schedulatore del sistema (tabella 3), dimostrano un overhead non trascurabile nonostante il programma sia stato parametrizzato per lo scambio di numerosi messaggi su memoria già allocata. Nel caso dello schedulatore, sembra che il costo delle due funzionalità di sicurezza sia equamente diviso, con un picco poco al di sotto dell'8% quando entrambe sono abilitate.

| Opzioni attive | Massimo | Minimo | Media | Deviazione Standard | Overhead (%) |
|---|---|---|---|---|---|
| nessuna | 727,32 | 723,32 | 725,29 | 1,67 | - |
| *init_on_alloc* | 726,25 | 724,7 | 725,36 | 0,56 | 0,01% |
| *init_on_free* | 741,34 | 735,91 | 738,35 | 1,93 | 1,80% |
| *init_on_** | 744,92 | 738,98 | 741,47 | 2,09 | 2,23% |

**Tabella 2:** Tempi di compilazione del kernel Linux (*wall clock time*)

| Opzioni attive | Massimo | Minimo | Media | Deviazione Standard | Overhead (%) |
|---|---|---|---|---|---|
| nessuna | 55,318 | 54,637 | 54,984 | 0,233 | - |
| *init_on_alloc* | 59,505 | 56,864 | 57,953 | 0,989 | 5,40% |
| *init_on_free* | 57,222 | 56,893 | 57,038 | 0,120 | 3,74% |
| *init_on_** | 59,565 | 58,368 | 59,139 | 0,421 | 7,56% |

**Tabella 3:** Tempi di esecuzione del programma *hackbench*

### 3.3.2 Risultati per i test di micro-benchmark

Le tabelle 4 e 5 mostrano i risultati ottenuti dall'invocazione della chiamata di sistema addizionale *microbenchmark*. I costi misurati sono elevati ma non necessariamente rispecchiano il rallentamento effettivo sul sistema, come provato negli esperimenti di macro-benchmark precedenti. Quello che queste misurazioni sono riuscite a catturare è il comportamento del kernel a fronte di una rapida allocazione e liberazione di oggetti, in particolare si riconferma il costo maggiore dell'azzeramento della memoria liberata rispetto a quella allocata.

Inoltre, è possibile osservare che i valori massimi ottenuti si discostano di molto dalla media, comportamento opposto ai valori minimi che invece rientrano all'interno dell'intervallo descritto dalla deviazione standard. Questi valori massimi sono dei casi limite raggiunti dall'aggiunta di una nuova lastra di memoria alla cache di oggetti in fase di allocazione, e dal rilascio di una lastra libera nel caso complementare.

| *init_on_alloc* | Massimo | Minimo | Media | Deviazione Standard | Overhead (%) |
|---|---|---|---|---|---|
| disabilitato | 660,112 | 94,444 | 107,675 | 18,908 | - |
| abilitato | 906,871 | 182,303 | 199,370 | 29,724 | 85,159% |

**Tabella 4:** Tempo stimato per l'esecuzione di *kmalloc*

Quando i meccanismi di inizializzazione forzata della memoria sono attivi, i casi limite precedenti sono ulteriormente peggiorati in quanto la memoria viene inizializzata due volte: un azzeramento iniziale per la pagina intera e poi uno minore per ogni oggetto da posizionare (o rimuovere). Purtroppo, l'allocatore delle pagine non è in grado di determinare se una pagina è stata richiesta da un allocatore superiore oppure da un'altra funzione del kernel e dunque è costretto ad inizializzare la memoria in ogni caso. La presenza di un flag GFP aggiuntivo dedito a segnalare una richiesta di pagine da parte di un allocatore a lastre risolverebbe il problema della doppia inizializzazione durante l'allocazione, ma non è detto che ne migliorerebbe la situazione generale visto che questi allocatori sono progettati per riutilizzare la loro memoria il quanto più possibile. Per quanto riguarda la liberazione della memoria, il problema posto dalla doppia inizializzazione non è di immediata risoluzione.

| *init_on_free* | Massimo | Minimo | Media | Deviazione Standard | Overhead (%) |
|---|---|---|---|---|---|
| disabilitato | 214,912 | 131,458 | 137,446 | 8,209 | - |
| abilitato | 555,093 | 266,204 | 299,321 | 33,693 | 117,773% |

**Tabella 5:** Tempo stimato per l'esecuzione di *kfree*

# 4 OSSERVAZIONI FINALI

Per poter essere utilizzato, un buon meccanismo di difesa non deve solo essere efficace nel contrastare una o più classi di vulnerabilità software, ma deve anche integrarsi correttamente con il resto delle funzionalità del kernel, e questa è la fonte principale di complessità che uno sviluppatore deve fronteggiare nello sviluppo di nuove mitigazioni della memoria. Se un meccanismo è caratterizzato da un costo di overhead troppo alto allora non verrà utilizzato; oppure, se possibile, verrà relegato ad essere uno strumento di debug per sistemi ancora instabili, com'è il caso della modalità generale di KASAN. Nella fase d'integrazione il meccanismo deve essere consapevole dell'abilitazione di eventuali strumenti di debug, della presenza di meccanismi di difesa analoghi già consolidati e non dovrebbe attivarsi nei casi in cui la memoria venga correttamente utilizzata.

Se si va a considerare l'inizializzazione forzata della memoria dinamica durante l'allocazione sotto questa luce, allora la funzionalità *init_on_alloc* è un buon meccanismo di difesa che dovrebbe essere abilitato di default. I costi di overhead associati, descritti nel capitolo precedente, sono quasi trascurabili e sono nulli nel caso gli oggetti allocati facciano uso di un costruttore della memoria, caratteristica preferita nell'inizializzazione di un oggetto. Inoltre, la classe di vulnerabilità legata all'utilizzo di memoria dinamica non inizializzata, oppure parzialmente inizializzata, viene completamente contrastata da questo meccanismo.

Una considerazione opposta può invece essere eseguita su *init_on_free*. Oltre ai costi elevati, una grande limitazione posta da questa funzionalità è quella di non risolvere completamente gli errori della memoria che va ad indirizzare, come la divulgazione di informazioni sensibili. Certo, l'utilizzo di *init_on_free* riduce al minimo la persistenza in memoria di tali informazioni, cancellandole appena lo spazio in cui risiedono venga rilasciato e dunque esse non siano più richieste. Tuttavia, questi benefici vengono a mancare nel momento in cui la memoria venga riutilizzata, cosa molto probabile in un sistema moderatamente attivo, oppure dove è possibile rivelare informazioni contenute in zone di memoria ancora allocate. Questi termini rendono *init_on_free* una mitigazione costosa, utilizzabile in sistemi poco dinamici dove la privacy è un elemento di estrema importanza, come supportato incidentalmente da uno studio sulle modalità di ricerca privata di due browser popolari in ambiente Linux [8].

I meccanismi di difesa del kernel Linux sono ormai una parte consolidata del kernel, e di questi molti sono abilitati di default e gli altri rimangono comunque altamente configurabili. Il problema maggiore che si è riscontrato nel loro studio non è una mancanza di funzionalità, bensì una mancanza di documentazione. Linux è un progetto massivo, in rapida evoluzione e naturalmente la documentazione non può riflettere le ultime novità, in quanto c'è bisogno di tempo perché esse vengano accettate e utilizzate. Tuttavia, delle componenti ormai già consolidate la documentazione alterna sezioni dettagliate ad altre carenti, o a volte anche assenti, e le mitigazioni di sicurezza sono tra queste ultime. Molta della documentazione inerente alla sicurezza si concentra nel descrivere l'infrastruttura dei moduli di sicurezza LSM o tratta delle funzioni crittografiche del kernel, ma solo una pagina [10] accenna all'esistenza dei meccanismi di sicurezza descritti in questo lavoro e di altri, qui non trattati. Il progetto Kernel Self Protection Project (KSPP), dedicato all'introduzione e manutenzione di nuovi meccanismi di difesa del kernel Linux, presenta una wiki che cerca di classificare le difese del kernel e gli attacchi che esso può ricevere, ma le descrizioni sono ancora scarne e la documentazione offerta dista

dall'essere esaustiva.*

L'ultima fonte rimane, come sempre, la lettura del codice sorgente e dei suoi commenti, ma un problema ricorrente riguardo queste mitigazioni è quello di essere frammentate in più file e di avere più implementazioni per più casi d'uso simili, ma differenti. Ne sono un esempio l'inizializzazione forzata della memoria, che ha una sua implementazione per l'allocatore delle pagine e un'altra, più complessa, per gli allocatori della famiglia SLAB; o ancora la gestione dei canarini dello stack, che differisce per gli stack dei processi e per quelli degli interrupt handler.

Il documentare con ordine queste mitigazioni agevolerebbe sia il lavoro degli utenti che quello degli sviluppatori. Per i primi, l'avere un quadro generale dei costi e benefici di ogni funzionalità di sicurezza permetterebbe di adeguare la configurazione di un sistema Linux alle proprie esigenze con facilità. Al momento, solo le descrizioni di aiuto nei file *Kconfig* svolgono un ruolo descrittivo, ma non sempre le informazioni contenute sono sufficienti per giustificare una scelta. Per gli sviluppatori invece, una buona documentazione metterebbe in risalto i punti meno efficienti da migliorare e gli spazi per poter innovare.

È incoraggiante poter osservare che ci sono già dei tentativi individuali per cercare di mettere ordine nelle relazioni dei meccanismi del kernel. Un esempio notevole è quello di Alexander Popov e la sua *Mappa di difesa del kernel Linux* [16] che cerca di mettere in luce i diversi legami tra meccanismi di sicurezza, vulnerabilità e sistemi di debug offerti dal kernel Linux.

## 4.1 CONCLUSIONI

In questo lavoro si sono andati a studiare alcuni dei meccanismi di difesa che contribuiscono alla sicurezza di un kernel Linux e sono stati analizzati nel dettaglio due meccanismi legati a due aree chiave nella gestione della memoria: i canarini dello stack e l'inizializzazione forzata della memoria heap. In particolare, sono stati eseguiti degli esperimenti sulle prestazioni di due meccanismi di hardening poco studiati, *init_on_alloc* e *init_on_free*, e sono state fatte delle considerazioni sulla loro efficacia nel difendere un sistema. Quello che è emerso dallo studio di queste funzionalità è che molta della loro complessità deriva dall'integrazione del singolo meccanismo con il resto del kernel e dalla carenza di documentazione al riguardo, sulla quale c'è ampio spazio per migliorare.

* https://kernsec.org/wiki/index.php/Kernel_Self_Protection_Project

# A LA CHIAMATA DI SISTEMA MICROBENCHMARK

Questa è la definizione della chiamata a sistema addizionale *microbenchmark* dedicata alla misurazione dei tempi di allocazione e liberazione della memoria.

**Listing 4:** Codice di microbenchmark

```
1  SYSCALL_DEFINE3(microbenchmark, int, block_size,
2                  int, number_obj, int, alloc_flag){
3      int i;
4      ktime_t start, middle, end;
5      char **pointers;
6      unsigned long alloc_res, free_res;
7      pointers = kmalloc_array(number_obj, sizeof(char *), GFP_KERNEL);
8      start = ktime_get();
9      for(i=0; i<number_obj; i++){
10         pointers[i] = kmalloc(block_size, GFP_KERNEL);
11     }
12     middle = ktime_get();
13     for(i=0; i<number_obj; i++){
14         kfree(pointers[i]);
15     }
16     end = ktime_get();
17     kfree(pointers);
18     alloc_res = middle - start;
19     free_res = end - middle;
20     if(alloc_flag){
21         return alloc_res;
22     }
23     else{
24         return free_res;
25     }
26 }
```

La funzione prende come parametri d'ingresso tre numeri interi: *block_size* rappresenta la dimensione dell'oggetto da allocare; *number_obj* indica il numero di volte per cui va ripetuta questa operazione; e infine *alloc_flag* funge da variabile booleana per determinare il valore di ritorno d'interesse della funzione.

La firma di una chiamata di sistema potrebbe avere delle leggere differenze a seconda dell'architettura per la quale si sta compilando. Per ovviare a questo problema sono presenti delle macro nella forma di *DEFINE_SYSCALLN*, dove *N* è il numero di parametri della chiamata di sistema, che in fase di compilazione vengono sostituite con le firme opportune dal preprocessore.

Come prima cosa, la funzione predispone un array di puntatori nella memoria heap grande quanto il numero di oggetti da allocare, per poi successivamente andarlo a popolare nel primo ciclo for. Terminata l'allocazione, un secondo ciclo for libera tutta la memoria precedentemente allocata e, in seguito, viene rilasciato anche l'array di supporto. Il tempo di esecuzione di ogni ciclo for viene misurato utilizzando la funzione *ktime_get* che restituisce il numero di nanosecondi trascorsi dall'ultimo avvio del sistema. Va notato che *ktime_t*, il valore di ritorno della funzione *ktime_get*, è un alias per il tipo primitivo *unsigned long*. Considerato che una

chiamata di sistema restituisce sempre un valore di tipo *long*, è possibile ad ogni invocazione ritornare una sola delle due misure effettuate facendo attenzione in user space ad interpretare il valore di ritorno correttamente.

Infine, il parametro booleano *alloc_flag* viene interpretato per determinare quale delle due misure deve essere restituita al chiamante: il tempo di allocazione se il parametro è vero (i.e. non zero); altrimenti il tempo di liberazione.

Una volta compilato il kernel con la nuova chiamata di sistema addizionale è possibile scrivere programmi in user space che ne facciano uso. Il programma in 5 è stato utilizzato per la raccolta delle misure.

**Listing 5:** Programma per la raccolta dei dati

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  #define MICROBENCHMARK 449
6  #define BUFFER_SIZE 10000
7  #define ALLOC_FLAG 1
8
9  int main(int argc, char **argv){
10     unsigned long result;
11     int block = 512;
12     int number = 1000;
13
14     for(int i=0; i<BUFFER_SIZE; i++){
15         result = syscall(MICROBENCHMARK, block, number, ALLOC_FLAG);
16         printf("%f\n", (double)result/number);
17     }
18 }
```

Il programma invoca per numerose volte *microbenchmark*, identificata dal suo indice nella tabella delle chiamate di sistema (449 nello specifico), utilizzando la funzione di libreria *syscall*. Ogni risultato dell'invocazione viene diviso per il numero di oggetti che è stato richiesto di allocare, in modo da ottenere una stima per la singola allocazione (o liberazione) di un oggetto in memoria. I risultati vengono poi stampati a video, dove potranno poi essere eventualmente raccolti all'interno di un file di log.

Infine, tramite la macro *ALLOC_FLAG* è possibile controllare il tipo di misura restituito dalla chiamata di sistema, nelle modalità descritte nei paragrafi precedenti.

# BIBLIOGRAFIA


[1] *2022 CWE Top 25 Most Dangerous Software Weaknesses*. 2022. URL: https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html (visitato il 22/07/2022).

[2] *About Linux Kernel*. 2021. URL: https://www.kernel.org/linux.html (visitato il 07/05/2022).

[3] *Adding a New System Call*. URL: https://docs.kernel.org/process/adding-syscalls.html (visitato il 30/08/2022).

[4] Aleph1. *Smashing The Stack For Fun And Profit*. 1996. URL: http://phrack.org/issues/49/14.html#article (visitato il 10/08/2022).

[5] Jeff Bonwick. «The Slab Allocator: An Object-Caching Kernel». In: *USENIX Summer 1994 Technical Conference (USENIX Summer 1994 Technical Conference)*. Boston, MA: USENIX Association, giu. 1994. URL: https://www.usenix.org/conference/usenix-summer-1994-technical-conference/slab-allocator-object-caching-kernel.

[6] Crispan Cowan et al. «Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks.» In: *USENIX security symposium*. Vol. 98. San Antonio, TX. 1998, pp. 63–78.

[7] Thurston Dang, Petros Maniatis e David Wagner. «The Performance Cost of Shadow Stacks and Stack Canaries». eng. In: *Proceedings of the 10th ACM Symposium on information, computer and communications security*. ASIA CCS '15. ACM, 2015, pp. 555–566. ISBN: 9781450332453.

[8] Xosé Fernández-Fuentes, Tomás F Pena e José C Cabaleiro. «Digital forensic analysis methodology for private browsing: Firefox and Chrome on Linux as a case study». In: *Computers & Security* 115 (2022), p. 102626.

[9] Tais Ferreira et al. «An experimental comparison analysis of kernel-level memory allocators». eng. In: *Proceedings of the 30th Annual ACM Symposium on applied computing*. SAC '15. ACM, 2015, pp. 2054–2059. ISBN: 1450331963.

[10] *Kernel Self-Protection*. URL: https://www.kernel.org/doc/html/latest/security/self-protection.html (visitato il 07/09/2022).

[11] Brian W. Kernighan e Dennis Ritchie. «The C programming language». eng. In: 2nd ed. Prentice-Hall software series. Upper Saddle River, N.J: Prentice Hall PTR, 1988, pp. 185–189. ISBN: 9780133086232.

[12] Michael Kerrisk. *The Linux programming interface : a Linux and UNIX system programming handbook / Michael Kerrisk*. eng. San Francisco: No Starch Press, 2010. ISBN: 978-1-59327-220-3.

[13] Sebastian Krahmer. *x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique*. 2005.

[14] Robert Love. *Linux Kernel Development Third Edition*. eng. Pearson, 2010. ISBN: 978-0-672-32946-3.

[15] *OpenBSD Innovations*. URL: https://www.openbsd.org/innovations.html (visitato il 31/08/2022).

[16] Alexander Popov. *Linux Kernel Defence Map*. URL: https://github.com/a13xp0p0v/linux-kernel-defence-map (visitato il 07/09/2022).

[17] Alexander Potapenko. *mm: security: introduce init_on_alloc=1 and init_on_free=1 boot options*. 2019-06-06. URL: https://patchwork.kernel.org/project/linux-hardening/patch/20190606164845.179427-2-glider@google.com/ (visitato il 25/08/2022).

[18] Dennis M Ritchie. «The evolution of the Unix time-sharing system». In: *Symposium on Language Design and Programming Methodology*. Springer. 1979, pp. 25–35.

[19] Dennis Ritchie e Ken Thompson. «The UNIX time-sharing system». eng. In: *Communications of the ACM* 17.7 (1974), pp. 365–375. ISSN: 0001-0782.

[20] Richard Stallman. *The GNU Manifesto*. 1985. URL: https://www.gnu.org/gnu/manifesto.en.html (visitato il 03/05/2022).

[21] *The Kernel Address Sanitizer (KASAN)*. URL: https://www.kernel.org/doc/html/latest/dev-tools/kasan.html (visitato il 22/08/2022).

[22] *What is Free Software?* 2001. URL: https://www.gnu.org/philosophy/free-sw.en.html (visitato il 06/05/2022).

[23] Chris Wright et al. «Linux security modules: General security support for the linux kernel». In: *11th USENIX Security Symposium (USENIX Security 02)*. 2002.