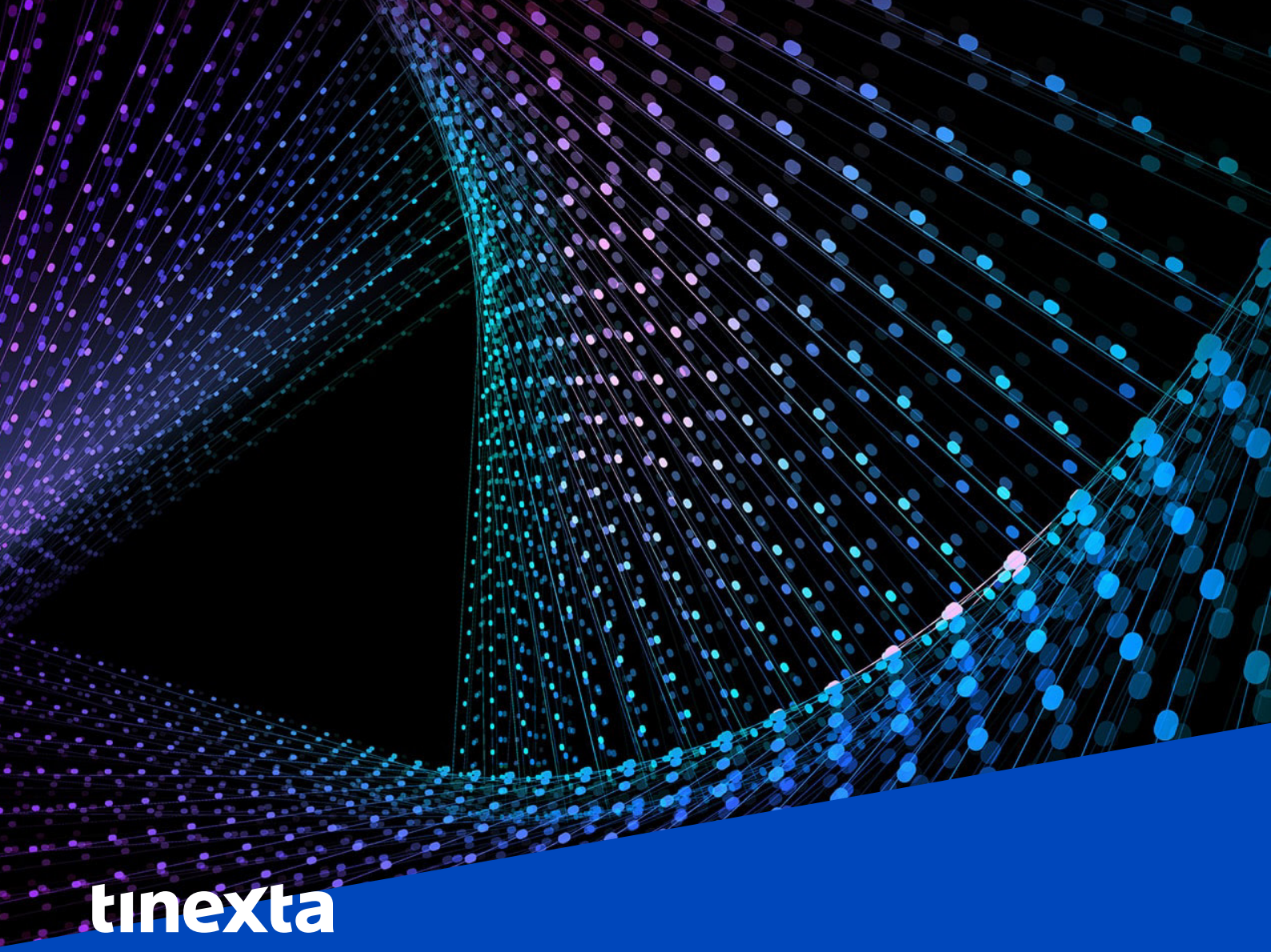


tinexta
defence

eBPF per la Digital Forensics & Incident Response:

osservabilità e azioni proattive a livello kernel

#TinextaDefenceBusiness

DFIR

Il Gruppo DFIR di Tinexta Defence è una Threat Response Unit specializzata in Digital Forensics & Incident Response che supporta imprese e pubbliche amministrazioni nella gestione di incidenti di sicurezza e nella produzione di evidenze digitali con valore probatorio.

L'attività del Gruppo integra competenze multidisciplinari e si articola in quattro aree principali:

- **Incident Response:** capacità di intervento rapido per contenere, eradicare e mitigare incidenti, riducendo l'impatto operativo;
- **Consulenze Tecniche d'Ufficio (CTU) e di Parte (CTP):** perizie informatiche conformi alle best practice di catena di custodia, a supporto del contesto giudiziario;
- **Forensic Readiness:** predisposizione preventiva di processi, tecnologie e standard per garantire che i dati raccolti siano accurati, integri e attestabili;
- **Ricerca e innovazione:** sperimentazione di tecnologie avanzate (eBPF, kernel telemetry, AI per anomaly detection, container forensics) per anticipare le minacce e sviluppare strumenti di nuova generazione.

In qualità di Threat Response Unit, il Gruppo DFIR non si limita alla fase di indagine post-evento, ma affianca i Security Operations Center (SOC) e le organizzazioni nella detection proattiva, nel threat hunting e nella gestione di crisi cyber.

La missione del Gruppo è elevare i livelli di sicurezza e resilienza delle infrastrutture critiche e dei sistemi informativi, coniugando rigore scientifico, innovazione tecnologica e capacità operativa a supporto della difesa digitale e del contesto giudiziario.

Summary

Abstract

La crescente complessità degli attacchi informatici richiede soluzioni capaci di garantire visibilità profonda e tempestiva a livello di sistema operativo, riducendo al minimo l'impatto prestazionale e preservando il valore forense dei dati raccolti.

In questo contesto, la tecnologia *eBPF (extended Berkeley Packet Filter)* si afferma come uno strumento innovativo per il campo della Digital Forensics & Incident Response (DFIR), consentendo l'esecuzione sicura di programmi personalizzati direttamente nel kernel Linux.

Il presente lavoro esplora le potenzialità di eBPF in ambito DFIR, con particolare attenzione a due direttrici principali:

- Osservabilità avanzata, attraverso il tracciamento di attività critiche come le sessioni SSH, i comandi eseguiti e i cambi di utenza, garantendo completezza e integrità della telemetria;
- Difesa proattiva, mediante meccanismi di controllo in grado di bloccare connessioni verso indirizzi IP malevoli, trasformando la raccolta di eventi in uno strumento di reazione immediata.

La soluzione proposta integra aspetti di *forensic readiness* (accuratezza temporale, integrità e formato log RFC 5424) con funzionalità di sicurezza operativa, ponendo le basi per un approccio DFIR più rapido, affidabile e integrabile con strumenti SOC e SIEM.

I risultati evidenziano come eBPF possa rappresentare una tecnologia strategica per la prossima generazione di strumenti di indagine e risposta, aprendo scenari di ricerca e sviluppo legati alla sicurezza di *container*, correlazione multi-evento e applicazioni di Machine Learning per rilevamento di anomalie a livello kernel.

Autori:

- Marco Tinti: Team Leader DFIR
- Riccardo Luzi: Forensics Analyst
- Andrea Mura: Malware Analyst

Introduzione

Un elemento cruciale in ambito DFIR *(Digital Forensics and Incident Response)* è la disponibilità di dati di qualità e in quantità sufficiente per condurre un'analisi efficace.

Maggiore è la visibilità sulle operazioni eseguite all'interno di un sistema, maggiore sarà la capacità di effettuare indagini accurate e complete. Allo stesso modo, la tempestività nella raccolta delle informazioni incide in maniera determinante sull'esito delle attività.

Nei sistemi basati su Linux esistono numerosi approcci e tecnologie per monitorare in modo puntuale e approfondito le attività, spesso attraverso codice che opera a livello del sistema operativo e sfrutta la posizione privilegiata del kernel per osservare e controllare l'intero ambiente.

Tra queste soluzioni, *eBPF (extended Berkeley Packet Filter)*[1] si distingue come strumento particolarmente potente per migliorare l'osservabilità di un sistema Linux. Si tratta di una tecnologia che consente di eseguire codice personalizzato in risposta a eventi specifici del kernel o dello spazio utente, senza modificare o ricompilare il kernel stesso. Il codice viene compilato in *bytecode* ed eseguito all'interno di una macchina virtuale integrata nel kernel, dove viene verificato da un *Verifier* che garantisce sicurezza e stabilità.

A differenza dei moduli kernel tradizionali:

- non accede direttamente alla memoria del kernel, riducendo il rischio di crash e vulnerabilità;
- può essere caricato e rimosso dinamicamente, senza riavvio del sistema;
- è *event-driven*, attivato da hook points predefiniti o personalizzati (es. *syscalls*, eventi di rete, *tracepoints*).

Grazie a queste caratteristiche, eBPF offre osservabilità avanzata, basso impatto prestazionale e la possibilità di agire in maniera proattiva, ad esempio bloccando connessioni o modificando il comportamento del sistema in risposta a eventi.

[1] https://ebpf.io/

Il sistema proposto ha l'obiettivo di mostrare un possibile utilizzo della tecnologia eBPF in questo campo, andando a collezionare informazioni riguardo le attività svolte tramite connessioni SSH verso un server Linux. Inoltre, mostra come tale tecnologia può essere utilizzata per proteggere proattivamente un sistema, proponendo una soluzione atta a bloccare le connessioni di rete dirette a uno specifico indirizzo IP presente in una lista configurata dall'utente.

Osservabilità su sistemi Linux

L'osservabilità è ormai un aspetto fondamentale sia per valutare le prestazioni e lo stato di un sistema, sia per implementare soluzioni di sicurezza efficaci.

Si intende per osservabilità la capacità di comprendere lo stato interno di un sistema complesso basandosi sui suoi output esterni. Migliorare questa capacità significa disporre di una visione più completa di ciò che accade, permettendo di individuare minacce in tempo reale e fornendo al contempo una solida base di conoscenza per analisi post-incidente. Il sistema operativo è il punto ideale per implementare strumenti di osservabilità, grazie alla sua posizione privilegiata che gli consente di monitorare e controllare l'intero ambiente.

In ambito Linux esistono diverse possibilità per estendere le capacità del kernel in base alle necessità operative.

Un primo approccio consiste nella modifica diretta del codice sorgente del kernel. Questa soluzione offre la possibilità di intervenire su qualsiasi parte del kernel per adattarlo a esigenze specifiche, ma richiede una conoscenza estremamente approfondita della sua *codebase*.

Ciò comporta inoltre limitazioni di portabilità tra diverse distribuzioni o versioni di Linux e, poiché ogni modifica deve essere approvata dalla community di sviluppo, può introdurre tempi di attesa che si misurano in mesi o persino anni.

In alternativa, è possibile utilizzare i moduli del kernel. Un modulo è un file oggetto contenente codice che estende le funzionalità del kernel a *runtime*, con la possibilità di essere caricato e rimosso in maniera dinamica *(on-demand)*. Molti driver di dispositivi hardware, ad esempio, sono implementati come moduli del kernel.

A differenza della modifica diretta del kernel, l'uso dei moduli consente di estenderne immediatamente le funzionalità, senza attendere l'approvazione della community e con minori problemi di portabilità.

Tuttavia, se il codice di un modulo non è sviluppato e revisionato con attenzione rispetto alla versione del kernel in cui deve operare, può introdurre vulnerabilità o causare crash del sistema. A questo punto entra in gioco eBPF, una tecnologia che permette di scrivere codice personalizzato da caricare e rimuovere dinamicamente, ampliando le capacità del kernel.

Pur condividendo con i moduli la possibilità di esecuzione a runtime, eBPF presenta una differenza fondamentale: i programmi vengono eseguiti all'interno della eBPF *Virtual Machine*, un ambiente isolato che opera su registri virtuali dedicati.

Questo approccio riduce drasticamente il rischio di introdurre vulnerabilità o instabilità, in quanto il codice eBPF non interagisce direttamente con la memoria del kernel.

Inoltre, il rischio di crash del kernel durante l'esecuzione viene ridotto grazie alla presenza del Verifier. Questo componente analizza il programma prima del caricamento, valutando tutti i possibili percorsi di esecuzione ed esaminando le istruzioni in ordine logico. Tra i principali controlli effettuati vi sono:

- **Controllo dell'accesso alla memoria:** garantisce che i programmi BPF accedano solo alla memoria a cui sono autorizzati ad accedere, assicurando che non superino limiti specifici (ad esempio, quando si accede a un array è necessario assicurarsi che non vi sia alcuna possibilità di accedere a un indice che superi i limiti di tale array);
- **Controllo dei puntatori prima della dereferenziazione:** una possibilità di causare il crash di un programma è dereferenziare un puntatore nullo (noto anche come *NULL*). Per evitare ciò, il Verifier richiede un controllo esplicito prima dell'effettivo accesso al valore puntato;
- **Esecuzione fino al completamento:** assicura che il programma eBPF venga eseguito fino al completamento e non consumi risorse indefinitamente, ponendo un limite al numero totale di istruzioni che elaborerà;
- **Istruzione non valide:** verifica che tutte le istruzioni in un programma siano istruzioni bytecode valide (ad esempio, *opcodes* noti);
- **Istruzioni non raggiungibili:** respinge programmi che contengono istruzioni irraggiungibili.

Caratteristiche principali di eBPF

I programmi eBPF sono detti “event-driven”: una volta caricati dinamicamente e associati a uno specifico evento, vengono eseguiti esclusivamente in corrispondenza di tale evento, senza necessità di riavviare il sistema.

In pratica, l’esecuzione avviene quando il kernel o un’applicazione raggiungono determinati hook points, che possono essere predefiniti (già presenti nel kernel) oppure personalizzati, a seconda del caso d’uso.

Hook points

Tra gli hook points predefiniti possiamo vedere chiamate di sistema, punti di ingresso e di uscita di funzioni (*fentry/fexit*), eventi di rete, kernel tracepoints e molti altri.

Se non ci sono hook points predefiniti che soddisfano un particolare caso d’uso, è possibile ancorare programmi eBPF più o meno ovunque nel kernel o in applicazioni dello spazio utente utilizzando rispettivamente kernel probes (*kprobe*) e user probes (*uprobe*).

Andiamo ad approfondire quelli utilizzati nella dimostrazione presente in questo articolo: **tracepoints**, **raw tracepoints** e **lsm**.

I **tracepoints** sono punti contrassegnati nel codice del kernel e rimangono stabili tra le diverse versioni del kernel, anche se una versione meno recente potrebbe non avere l'insieme completo di tracepoints aggiunti in una più recente. Ogni tracepoint ha un formato che descrive i campi che vengono tracciati da esso e, quindi, quali informazioni possiamo recuperare dal tracepoint.

Di conseguenza, le informazioni ricavate dal formato possono essere usate per creare una struttura dati che verrà associata a un set di argomenti grezzi quando utilizziamo un programma di tipo tracepoint.

Per ottenere prestazioni migliori è possibile utilizzare i **raw tracepoints**: hook points uguali ai tracepoint che permettono però di accedere agli argomenti grezzi senza dover creare la struttura dati ed eseguire l'associazione.

Oltre a raccogliere informazioni dalle funzioni del kernel, i programmi eBPF possono anche influenzare come questo si comporta in risposta a specifici eventi a cui il programma è associato.

Questo è possibile grazie all'**interfaccia LSM**, che fornisce una serie di hook in grado di attivarsi appena prima che il kernel agisca su una struttura dati e su path critici legati alla sicurezza del sistema. La funzione chiamata da uno di questi hook può decidere se consentire o meno l'esecuzione dell'azione impedendo, ad esempio, di accedere a uno specifico file, modificare i permessi di una cartella o contattare un indirizzo IP. È importante notare che BPF LSM viene supportato a partire dalla versione 5.7 del kernel, e che è necessario abilitare specifiche capacità del kernel per poter utilizzare questo tipo di hook.

Ciclo di vita

Una volta identificati gli hook di interesse, è possibile realizzare un programma eBPF utilizzando una delle *toolchain* esistenti che ne permettono l'implementazione in linguaggi come C e Rust, per poi agevolarne la conversione in bytecode, ovvero il formato atteso dal kernel.

Il programma eBPF viene poi caricato nel kernel utilizzando la chiamata di sistema **bpf**, usando tipicamente una delle molteplici librerie eBPF disponibili che permettono di sviluppare la parte del sistema residente nello spazio utente utilizzando linguaggi come C, Rust, Python e Go.

Una volta caricato, il programma eBPF viene sottoposto ai controlli effettuati dal Verifier descritto nel capitolo precedente. Se i controlli vengono superati, il **Just-in-Time Compiler** converte il bytecode generico del programma nel set di istruzioni specifico della macchina, rendendolo efficiente come se fosse codice del kernel nativamente compilato o codice caricato come un modulo del kernel.

Mappe e strutture dati

Per rendere i dati raccolti utilizzabili sia da altri programmi eBPF che da applicazioni nello spazio utente, viene utilizzato il concetto di **eBPF Maps**. Queste mappe sono strutture dati utilizzate per il salvataggio e la condivisione di dati nell'ecosistema eBPF e possono essere *hash tables, arrays, ring buffers*, strutture *LRU (Least Recently Used)* e molte altre.

Per interagire con le mappe e con specifiche strutture dati del kernel, i programmi eBPF devono utilizzare delle funzioni dedicate dette **helper functions**. Queste costituiscono un'API (Application Programming Interface) stabile offerta dal kernel che evita eventuali problemi di compatibilità tra programmi e versioni diverse del kernel.

Sicurezza

Date le capacità e potenzialità dei programmi eBPF, l'aspetto della sicurezza è di fondamentale importanza.

Per questo motivo vengono applicati diversi livelli di sicurezza:

- Generalmente, un programma eBPF può essere caricato nel kernel solo se il processo che lo sta caricando è in esecuzione in modalità privilegiata (*root*) o se la *capability CAP_BPF* è abilitata. In alternativa, è possibile abilitare **unprivileged eBPF** per permettere a processi non privilegiati di caricare dei programmi eBPF, che vengono comunque sottoposti a restrizioni in termini di funzionalità e accesso al kernel;
- Ogni programma eBPF caricato deve superare i controlli eseguiti dal Verifier;
- Superate le verifiche, il programma viene sottoposto a un processo di hardening che include la protezione della memoria utilizzata dal programma impostata in modalità read-only e l'oscuramento delle costanti presenti nel codice;
- Mentre è in esecuzione, un programma eBPF non può accedere direttamente ad aree di memoria arbitrarie del kernel. L'accesso a dati e strutture fuori dal contesto del programma è consentito solo tramite helper functions, garantendo così che accessi ed eventuali modifiche siano consistenti e sicure.

Tracciamento delle attività svolte via SSH

Come esempio pratico di una potenziale applicazione di programmi eBPF in campo DFIR viene presentato un sistema per tracciare connessioni SSH su un server Linux.

L'obiettivo è tracciare le attività svolte tramite connessioni SSH dal momento della connessione iniziale fino alla disconnessione, ponendo attenzione ai comandi eseguiti dagli utenti connessi.

Vengono inoltre correlate alcune informazioni per continuare a tracciare le attività con indirizzi e utenze originali anche in caso di cambio di utenza effettuato con comandi dedicati o tramite connessione SSH in *localhost*.

Infine, viene utilizzato un approccio di blacklisting per bloccare connessioni a specifici indirizzi IP, se questi sono presenti in una lista configurata dall'utente (es. una lista di IP notoriamente legati a server C2 malevoli).

Tracciamento di sessioni SSH

Quando un client SSH si collega a un server, viene invocata la chiamata di sistema **getpeername** da un processo **sshd** per recuperare le informazioni riguardanti l'indirizzo del peer che si sta collegando a uno specifico *socket*.

Utilizzando un tracepoint su questa chiamata di sistema è possibile recuperare indirizzo IP e porta del client collegato.

Dopo una serie di operazioni, un clone del processo **sshd** crea una shell invocando la syscall **execve** per eseguire **/bin/bash**. Da questo punto in poi, ogni comando eseguito utilizzando questa shell avrà come PPID (Parent PID) il PID del processo che ha creato la shell.

Utilizzando il tracepoint **sched_process_exec**, attivato ogni volta che un programma viene eseguito, è possibile collezionare informazioni riguardo il comando eseguito, il percorso del file eseguito e gli eventuali argomenti passati ad esso.

Inoltre, utilizzando delle helper functions di eBPF è possibile ottenere informazioni riguardo **PID**, **Parent PID** e **UID** del processo che ha attivato il programma eBPF.

Cambio di utenza tramite comando

Nel caso di un cambio di utenza effettuato tramite il comando **su** vengono creati diversi processi in successione prima della creazione della nuova shell di login, facendo perdere il riferimento al PID del processo che ha precedentemente creato la shell iniziale.

Questo porta alla perdita di correlazione tra i nuovi comandi eseguiti e le informazioni della sessione SSH relativi a indirizzo IP sorgente, porta sorgente e utenza inizialmente utilizzata per la connessione.

Per ovviare a questo problema viene utilizzato il tracepoint **sched_process_fork** per tenere traccia delle relazioni tra processo padre e processo figlio alla creazione di ogni nuovo processo, così da poter correlare eventuali nuove shell a quella originale. La catena degli eventi di fork viene tracciata con una mappa LRU per limitare il consumo di memoria, evitando overflow in ambienti ad alto carico.

In questo modo, anche in caso di cambio di utenza tramite il comando **su**, è possibile risalire all'utenza utilizzata inizialmente, insieme all'IP e alla porta del client connesso.

Cambio di utenza tramite SSH locale

Nel caso di un cambio di utenza effettuato avviando una nuova sessione SSH verso *localhost* (es. *utente2@localhost*), non è sufficiente tracciare la creazione di processi per poter risalire alle informazioni della sessione originale.

Per la gestione della nuova sessione viene creato un nuovo processo **sshd** che non è nella catena dei figli del processo che ha creato la shell iniziale.

Per poter risalire ai dati di interesse viene utilizzato un tracepoint sulla chiamata di sistema **getsockname**. Quando l'utente avvia la nuova sessione in locale, un processo **ssh** utilizza tale chiamata di sistema per recuperare informazioni riguardanti il proprio indirizzo IP e porta.

D'altra parte, il nuovo processo **sshd** utilizzerà **getpeername** per recuperare le informazioni di indirizzo IP e porta utilizzati dal client che si è connesso.

Grazie a queste informazioni è dunque possibile ottenere il PID del processo **ssh**, e da lì, ripercorrere la catena di processi per risalire alle informazioni originali della sessione.

In questo modo, anche in caso di cambio di utenza tramite una nuova connessione SSH, è possibile risalire all'utenza utilizzata inizialmente, insieme all'IP e alla porta del client connesso.

Blocco di indirizzi IP

Il sistema proposto non si limita a collezionare passivamente informazioni di interesse, ma permette di bloccare in modo proattivo tentativi di connessione verso specifici indirizzi IP configurati dall'utente.

Utilizzando una mappa dedicata, una lista di indirizzi IP configurata dall'utente viene passata dall'applicazione nello spazio utente al programma eBPF.

Il programma eBPF utilizza un hook fornito dall'interfaccia LSM che viene attivato durante il processamento di una invocazione della chiamata di sistema **connect**.

Dagli argomenti della **connect** è possibile risalire all'indirizzo IP a cui il processo chiamante sta cercando di connettersi. Se questo indirizzo IP è presente nella lista configurata, il programma eBPF altera il valore di ritorno della **connect** bloccandone così l'esecuzione, e indica all'applicazione nello spazio utente di generare un alert per indicare una possibile attività sospetta.

La porzione di codice che segue mostra un semplice esempio: utilizzando il codice di ritorno **–EPERM**, l'hook LSM di tipo **socket_connect** permette di bloccare la connessione verso uno specifico indirizzo IP di destinazione configurato dall'utente, e allo stesso tempo tracciare tale l'azione con un messaggio di debug a livello kernel tramite la funzione **bpf_printk**.

```
SEC("lsm/socket_connect")
int BPF_PROG(block_connection, struct socket *sock, struct
sockaddr *addr, int addrlen) {
    struct sockaddr_in *addr_in = (struct sockaddr_in *)addr;
    __u32 dst_ip = addr_in->sin_addr.s_addr;
    if (dst_ip == <IP_DA_BLOCCARE>) {
        bpf_printk("Connessione verso %d bloccata\n", dst_ip);
        return -EPERM;
    }
    return 0;
}
```

Logging e validità forense dei dati

La telemetria prodotta dal sistema è progettata per avere valore probatorio e operatività Security Operations Center (SOC), tenendo conto di tre principi cardine: accuratezza temporale, integrità e provenienza attestabile.

Per ogni comando eseguito via SSH viene registrato un evento con:

- **Timestamp duale:** tempo monotono derivato da **bpf_ktime_get_ns()** per garantire l'ordinamento sequenziale degli eventi e tempo *wall-clock* in formato UTC per correlazioni esterne;
- **Boot ID:** identificativo univoco del sistema per distinguere i riavvii, recuperato tramite lettura di */proc/sys/kernel/random/boot_id* dallo spazio utente e passato al kernel via mappe eBPF;
- **Sequenza per CPU:** contatore incrementale per ordinare gli eventi per core, essenziale in ambienti multicore;
- **Contatori eventi persi:** informazioni su eventuali perdite di dati dovute a *buffer overflows*, recuperate tramite **bpf_ringbuf_query()**, per garantire trasparenza sulla completezza del log;
- **Username e UID originali:** username e UID del processo **sshd** che ha creato la prima shell;
- **Username e UID correnti:** username e UID del processo che ha eseguito il comando corrente;
- **Indirizzo sorgente:** indirizzo IP e porta del client inizialmente connesso;
- **PID e Parent PID:** identificatori del processo e del padre;
- **Comando eseguito:** il comando completo con percorso del file binario;
- **Argomenti:** parametri passati al comando.

I tentativi di connessione verso indirizzi IP presenti nella blacklist configurata generano log con lo stesso set informativo, arricchiti da una flag che evidenzia l'attività sospetta.

Le informazioni sono salvate in un file di log dedicato in formato *syslog* versione RFC 5424, che supporta dati strutturati per consentirne l'integrazione con sistemi SIEM come ELK o Splunk. Questo formato permette query avanzate su campi come UID originale o indirizzo IP, facilitando analisi forensi e correlazioni in tempo reale.

Conclusioni e sviluppi futuri

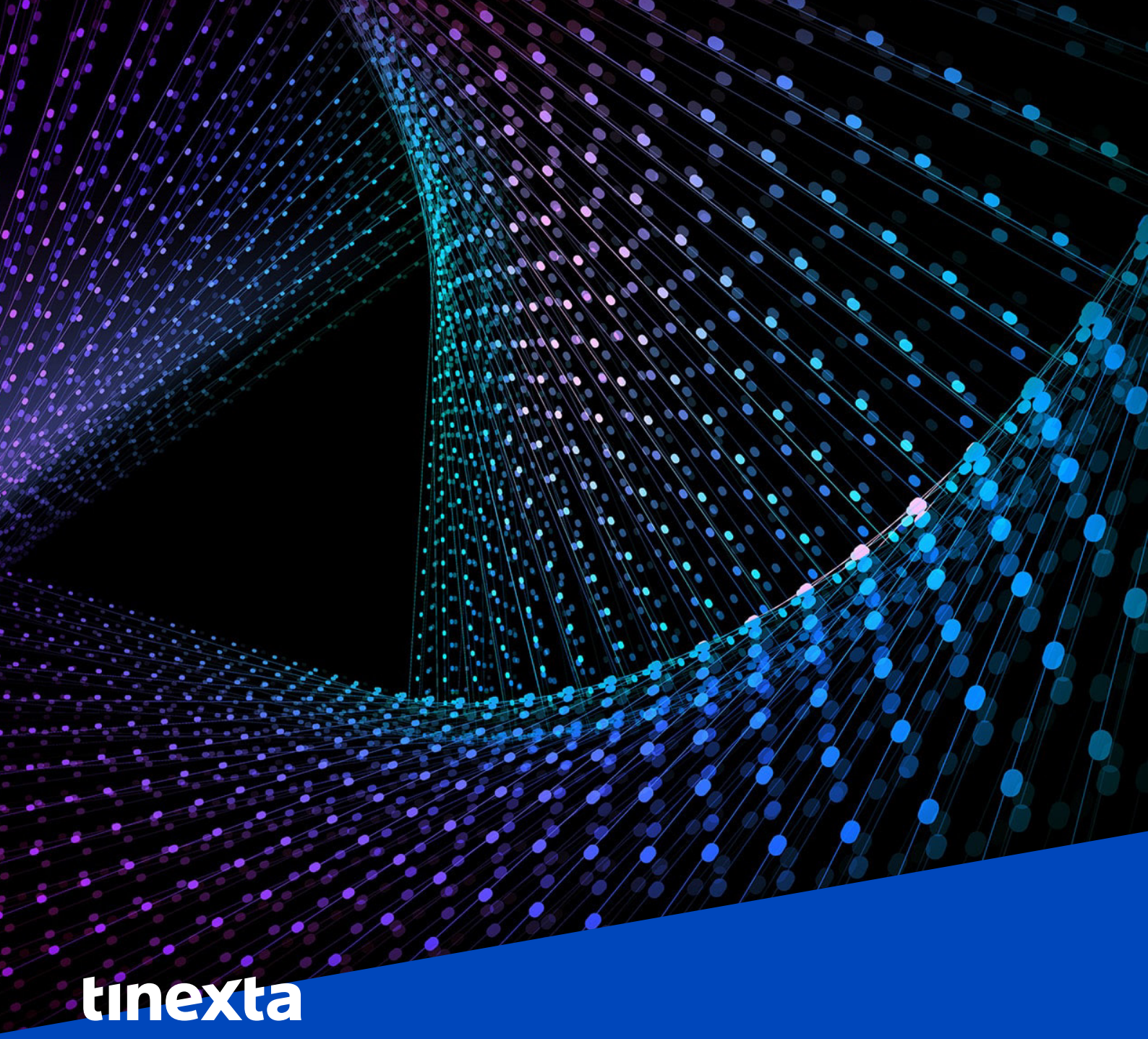
La tecnologia eBPF si dimostra molto valida e promettente in campo difensivo dei sistemi e investigativo degli incidenti. Grazie alla grande libertà fornita all'utente nello sviluppo di programmi eBPF è possibile andare incontro a numerosi casi d'uso di vitale importanza per la protezione dei sistemi Linux. È comunque importante tenere conto del fatto che alcune funzionalità utili di questa tecnologia sono disponibili solo in versioni più recenti del kernel Linux, e che per utilizzarla serve abilitare delle capacità del kernel che potrebbero non essere compatibili con i criteri di sicurezza di alcune realtà.

Alcuni sviluppi futuri del sistema descritto sono:

- **Azioni proattive verso processi sospetti:** compiere ulteriori azioni proattive verso processi ritenuti sospetti, come il blocco delle connessioni sopra citato;
- **Visibilità dei container:** visto il crescente e ampio utilizzo di container e microservizi, aggiungere informazioni riguardo i container all'interno dei quali vengono eseguiti i comandi;
- **Correlazione di altri eventi:** raccogliere informazioni riguardo altri tipi di eventi che sono legati in qualche modo ai comandi eseguiti dagli utenti connessi;
- **Regole per alert:** aggiungere un sistema di regole per generare alert in determinate situazioni.

L'integrazione futura con tecniche di Machine Learning per anomaly detection basata su metriche eBPF rappresenta un'area di ricerca promettente. Progetti open source come Falco (CNCF)[2] e ricerche accademiche recenti suggeriscono che l'applicazione di ML a telemetria kernel-level può migliorare significativamente il rapporto signal-to-noise, con alcuni studi che riportano riduzioni dei falsi positivi nell'ordine del 30-40% in contesti specifici. È importante notare che il sistema presenta limitazioni su kernel precedenti alla versione 5.10, dove alcune funzionalità come raw tracepoints e ring buffers potrebbero non essere disponibili, richiedendo implementazioni alternative con maggiore overhead.

[2] https://falco.org/



tinexta
defence

Defence Tech | Next |
Donexit | Foramil | Innodesi

Via Giacomo Peroni, 452 – 00131 Roma
tel. 06.45752720 – info@defencetech.it
www.tinextadefence.it

#TinextaDefenceBusiness