INTRODUZIONE eBPF

<https://sysdig.com/blog/sysdig-and-falco-now-powered-by-ebpf/> -> parte 1

<https://sysdig.com/blog/the-art-of-writing-ebpf-programs-a-primer/> -> parte 2

RISC:

Reduced Instruction Set Computer (in acronimo RISC), nell'elettronica digitale, indica un'idea di progettazione di architetture per microprocessori che predilige lo sviluppo di un'architettura semplice e lineare. Questa semplicità di progettazione permette di realizzare microprocessori in grado di eseguire il set di istruzioni in tempi minori rispetto a una architettura CISC.

CISC:

Complex Instruction Set Computer (CISC), in elettronica digitale, indica un'architettura per microprocessori formata da un set di istruzioni contenente istruzioni in grado di eseguire operazioni complesse come la lettura di un dato in memoria, la sua modifica e il suo salvataggio direttamente in memoria tramite una singola istruzione. Il termine è nato per distinguere l'altro paradigma dominante nei microprocessori, il paradigma Reduced Instruction Set Computer (RISC).

SYSTEM V APPLICATION BINARY INTERFACE

L' interfaccia binaria dell'applicazione System V è un insieme di specifiche che descrivono in dettaglio le convenzioni di chiamata , i formati dei file oggetto , i formati dei file eseguibili , la semantica dei collegamenti dinamici e molto altro per i sistemi conformi alla specifica X/Open Common Application Environment e alla definizione dell'interfaccia System V. . È oggi l'ABI standard utilizzato dai principali sistemi operativi Unix come Linux, i sistemi BSD e molti altri. Il formato eseguibile e collegabile (ELF) fa parte del sistema V ABI.

1. INTRODUZIONE

eBPF è una parte fondamentale del kernel Linux che fornisce strumenti di tracciamento e di sicurezza.

Extended Berkeley Packet Filter (eBPF) è una tecnologia moderna e potente utilizzata nel kernel Linux per un'ampia varietà di applicazioni, tra cui networking e tracing. Fondamentalmente, eBPF consente a un utente (in alcuni casi privilegiato) di inserire codice quasi generico nel kernel. Tale codice verrà quindi eseguito a un certo punto nel tempo, di solito dopo che si sono verificati determinati eventi di interesse nel kernel. Questi eventi in genere si riferiscono (ma non sono limitati) al networking e al tracing.

1. SCOPO

Come accennato in precedenza, l'obiettivo finale di eBPF è consentire l'esecuzione del codice utente nel kernel. Questo in teoria suona abbastanza simile a ciò che il kernel offre tradizionalmente agli utenti per estendere le sue funzionalità, ovvero moduli del kernel caricabili . I moduli del kernel caricabili sono costituiti da codice C compilato per uso generico caricato in fase di esecuzione all'interno del kernel tramite i comandi insmod/modprobe. Il codice di un modulo del kernel di solito si collega a vari sottosistemi del kernel in modo che venga richiamato automaticamente al verificarsi di determinati eventi. Ciò è stato utile per gli sviluppatori che desiderano implementare il supporto per nuovi dispositivi hardware o funzioni di tracciamento, ad esempio.

Questa analogia con eBPF non potrebbe essere più lontana dalla verità! Un grande vantaggio di eBPF è che, a differenza di un modulo del kernel, eseguirà solo codice ritenuto completamente sicuro da eseguire . Nello specifico, ciò significa che non porterà mai a un crash del kernel o all'instabilità del kernel. Questo è uno dei maggiori punti di forza per eBPF. È anche qualcosa attualmente difficile da ottenere con altre tecnologie senza rinunciare a una seria flessibilità.

1. SCRITTURA

Per raggiungere la sua garanzia di sicurezza, eBPF è essenzialmente implementato come una macchina virtuale di processo nel kernel. Questa macchina virtuale esegue programmi sicuri per conto dell'utente. eBPF espone all'utente un processore virtuale, con un set personalizzato di istruzioni simili a RISC che sono relativamente semplici da comprendere e gestire. Fornisce inoltre una serie di registri CPU virtuali e un'area di memoria dello stack. Di conseguenza, gli sviluppatori possono progettare e scrivere programmi in bytecode eBPF da eseguire in questo ambiente virtuale e passare i programmi alla macchina virtuale. Il framework eBPF inizierà a eseguire questi programmi solo dopo aver verificato che sono sicuri dal punto di vista dell'esecuzione. I controlli di verifica includono l'assicurarsi che un determinato programma eBPF non possa entrare in un ciclo infinito, non possa accedere a memoria non valida e altri invarianti di chiave.

È molto importante notare che uno sviluppatore non è necessariamente tenuto a creare da zero il bytecode eBPF quando scrive un nuovo programma. Gli sviluppatori eBPF hanno effettivamente implementato un back-end eBPF per LLVM (Low-Level Virtual Machine), il che significa che Clang può essere utilizzato per compilare un sottoinsieme di codice C standard in un file oggetto eBPF, che viene quindi caricato all'interno del kernel per la verifica e ulteriore utilizzo. Sebbene la traduzione da C a eBPF non sia priva di avvertimenti significativi, rende relativamente facile scrivere nuovo codice di strumentazione in un linguaggio di programmazione familiare come C. Inoltre, espande enormemente i casi d'uso di eBPF.

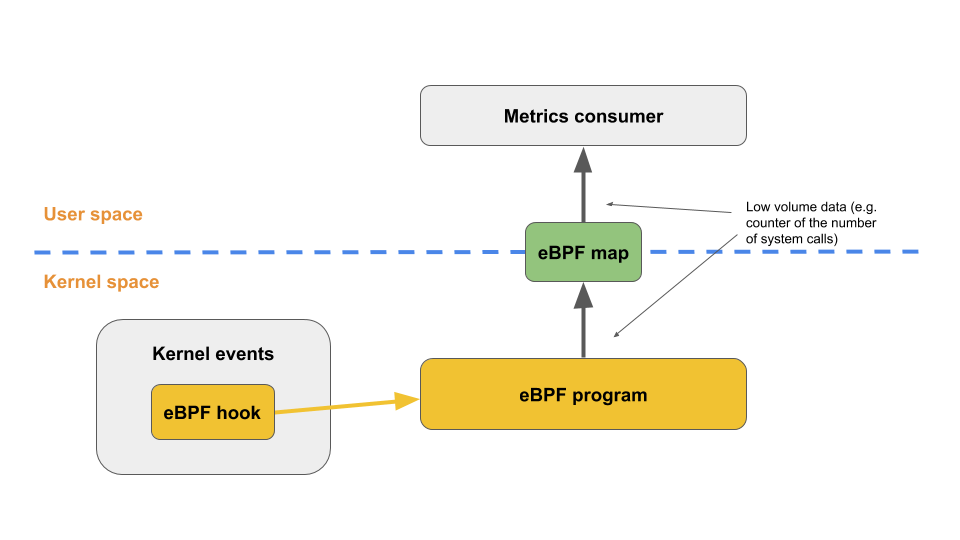
1. ARCHITETTURA

Ciò che rende eBPF molto interessante è che questi programmi possono essere eseguiti in vari punti di esecuzione nel kernel collegandoli a **tracepoint** statici , **kernel** dinamici e **probes** utente e molti altri punti di **hook**. Quando un programma eBPF collegato viene eseguito, riceve come input alcuni dati rilevanti provenienti dal kernel stesso.

Ad esempio, se collegato all'esecuzione di una chiamata di sistema tramite un tracepoint di chiamata di sistema, riceve gli argomenti della chiamata di sistema passati dal processo dello spazio utente che richiama la chiamata di sistema al kernel. Il programma può utilizzare questo input per mutare attivamente lo stato del sistema. Questo è tipico per i casi d'uso di rete per filtrare un pacchetto di rete, ad esempio. Oppure può calcolare passivamente un insieme di metriche, tipico per il tracciamento dei casi d'uso. In quest'ultimo caso, queste metriche possono quindi essere inviate allo spazio utente utilizzando strutture di dati chiamate "**mappe** eBPF". Le mappe eBPF sono strutture di dati chiave/valore più o meno generiche che sono condivise tra lo spazio utente e il kernel e consentono un flusso di dati a bassa velocità.

Nella stragrande maggioranza dei casi d'uso di tracciamento eBPF, i programmi eBPF vengono collegati a un punto di esecuzione molto occupato (in termini di frequenza) nel kernel (ad es. trasmissione di un pacchetto di rete o esecuzione di una chiamata di sistema) per calcolare statistiche numeriche come il numero di rete pacchetti trasmessi o numero di chiamate di sistema eseguite per tipo. Questi vengono posizionati in una mappa in modo che le applicazioni utente che guidano il processo di tracciamento possano recuperarli al proprio ritmo.

Ecco come appare l'architettura tipica di una pipeline di strumentazione eBPF:



Per garantire buone prestazioni sul lato kernel, il set di istruzioni RISC di un programma eBPF è abbastanza semplice da poter essere tradotto in modo relativamente semplice in codice macchina nativo tramite un passaggio JIT incorporato nel kernel . Ciò significa che subito dopo la verifica della sicurezza del programma, il runtime non subirà effettivamente l'overhead prestazionale dovuto all'esecuzione del bytecode eBPF tramite la macchina virtuale. Eseguirà semplicemente il codice macchina nativo, migliorando significativamente le prestazioni.

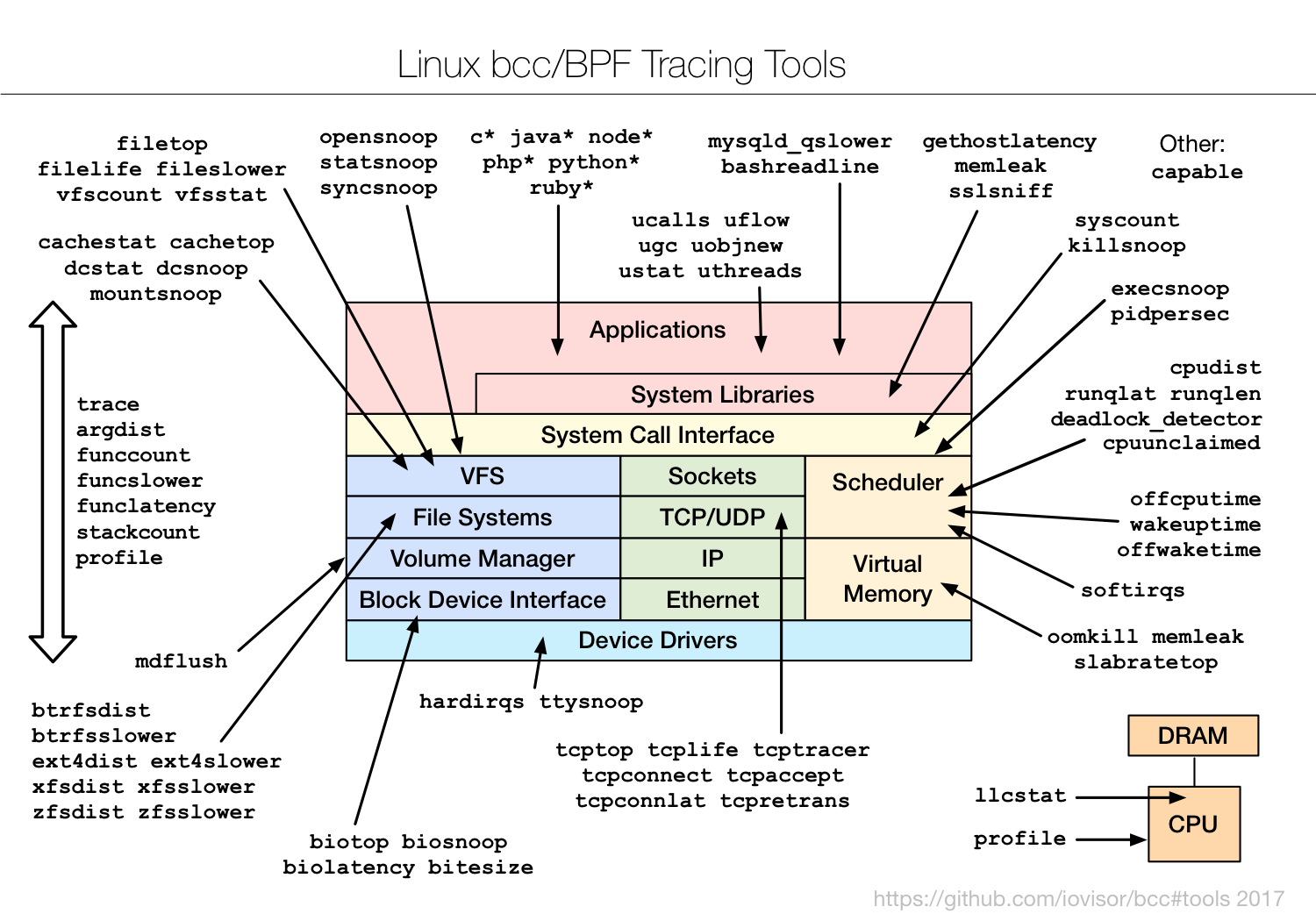
1. STRUMENTI

Considerando la popolarità di eBPF, vale la pena confrontare Sysdig con il resto degli strumenti nell'ecosistema eBPF. Un confronto completo non è possibile poiché i nuovi strumenti e framework basati su eBPF vengono introdotti a un ritmo molto rapido. Possiamo almeno confrontarlo con due dei progetti eBPF più grandi e popolari quando si tratta di tracciamento. Questi sono:

* bcc
* bpftrace (il più recente dei due)

**bcc** è essenzialmente un framework per scrivere programmi eBPF. Astrae una buona parte della complessità della gestione dell'eBPF stesso. In particolare, bcc semplifica la compilazione di un programma eBPF da C utilizzando LLVM, nonché i meccanismi effettivi di caricamento di un programma eBPF nel kernel e di collegamento al sottosistema interessato. Inoltre, bcc offre anche la possibilità di scrivere programmi eBPF in Python e Lua anziché in C. Ancora di più, bcc fornisce buoni collegamenti orientati agli oggetti quando si lavora con le mappe.

Per questo motivo, bcc è una buona scelta quando si scrivono programmi eBPF moderatamente complessi. Per impostazione predefinita, il framework fornisce dozzine di piccoli programmi eBPF che possono essere utilizzati per casi d'uso di risoluzione dei problemi una tantum. Questo è un diagramma completo di tali strumenti:



Uno dei problemi con bcc è che la scrittura di programmi eBPF può ancora essere notevolmente complicata nonostante i collegamenti Python e Lua. L'utente deve tenere a mente molte ipotesi sul modo in cui funzionano i programmi eBPF, in particolare i loro limiti. Per semplificare questa situazione è stato creato uno strumento più recente, bpftrace. bpftrace si trova sopra bcc. Invece di richiedere agli utenti di scrivere i propri programmi rispetto all'API bcc, offre una sintassi di livello superiore più espressiva. La sintassi è molto simile a quella del popolare framework di tracciamento DTrace . Ciò rende efficace eseguire il tracciamento avanzato tramite one-liner relativamente semplici .

1. COMPILAZIONE

L’attributo sopra le funzioni all’interno del file eseguito dal kernel è un attributo per il compilatore che si utilizza per indicare a LLVM di inserire il codice oggetto per la funzione in una sezione ELF ( Executable and Linkable Format ) separata, il cui nome nel file oggetto finale è inserito deltro l’attributo. Questo fa parte di un protocollo implicito tra lo sviluppatore che ha scritto la funzione e il loader eBPF che deve conoscere l'evento di sistema a cui deve allegare tale programma eBPF.

Dopo aver compilato il programma (con make solitamente) Clang e LLVM elaboreranno il codice sorgente ed emetteranno un singolo file oggetto contenente il programma eBPF.

La sezione ELF contiene il bytecode eBPF per la funzione scritta dall’utente.

1. eBPF LOADER

Il BPF loader esegue il programma.

Il file ELF contenente il programma eBPF viene passato come input al caricatore eBPF, che esegue le seguenti operazioni:

* Analizza le sezioni ELF e scegli quelle che iniziano con una determinata parola chiave. Ad esempio, una parola chiave è raw\_tracepoint . Ciò indica al caricatore che la sezione ELF contiene un programma eBPF che dovrà essere collegato a un evento kernel raw tracepoint. I raw tracepoint, quando disponibili, consentono di gran lunga le migliori prestazioni, a scapito della flessibilità, rispetto ad altri tipi di eventi del kernel a cui un programma eBPF può collegarsi (kprobes, uprobes, tracepoints, ...).
* L'altra parte del nome della sezione ELF verrà interpretata come il nome dell'evento. Questo identifica un raw tracepoint che viene chiamato ogni volta che viene eseguita una nuova chiamata di sistema. Si tratta essenzialmente di una stringa che il kernel riconosce direttamente come evento di sistema e può essere utilizzata per identificarla in modo univoco. L'elenco completo dei tracepoint supportati può essere facilmente esplorato eseguendo perf list sull'host.
* Una volta che l'evento del kernel è stato convalidato, il programma eBPF viene caricato nel kernel. Questo viene fatto tramite la chiamata di sistema bpf . Durante questa fase il kernel verifica che il programma sia sicuro da eseguire e, facoltativamente, lo tradurrà in codice macchina tramite il processo JIT. La **chiamata di sistema bpf** restituirà un descrittore di file che identifica il programma eBPF caricato o un errore.
* Infine, utilizzando la chiamata di sistema bpf, o la chiamata di sistema perf\_event\_open a seconda del tipo di evento del kernel , il caricatore eBPF istruirà il kernel ad allegare il programma eBPF appena caricato all'evento identificato al passaggio precedente.

L'intera faccenda è un po' complicata, ma è per lo più solo standard. Successivamente, il nostro programma verrà chiamato ogni volta che l'evento si attiva nel kernel.

1. eBPF VERIFIER

R0 è uno degli undici registri della macchina virtuale eBPF (R0-R10). Può succedere che si violi un requisito di un programma eBPF. Ogni programma eBPF deve sempre restituire un valore intero alla fine della sua esecuzione, e questo valore di ritorno deve essere memorizzato in R0. Il valore di ritorno è necessario perché la maggior parte delle volte il kernel utilizzerà effettivamente il valore di ritorno del programma e agirà in base al suo valore. Ad esempio, se un programma eBPF viene utilizzato per filtrare i pacchetti di rete, il valore restituito verrà interpretato come booleano per eliminare/accettare il pacchetto.

Il verificatore rileva errori simulando efficacemente ogni singolo ramo di esecuzione che il programma eBPF potrebbe assumere in fase di esecuzione. Tiene traccia del valore e del tipo dei registri per ogni ramo, assicurandosi che siano correttamente inizializzati se mai vengono letti.

Il verificatore eBPF tiene traccia di quale memoria punta ogni registro per ogni ramo che potrebbe eventualmente essere eseguito, e nega gli accessi che potrebbero essere potenzialmente non sicuri.

Il verificatore diventa sempre più intelligente con ogni nuova versione del kernel , semplificando la vita degli sviluppatori di programmi eBPF.

1. eBPF MEMORY ACCESS

Dobbiamo effettivamente accedere agli argomenti passati alla chiamata di sistema. Per fare questo, dobbiamo introdurre il concetto di "contesto". Ad ogni programma eBPF, all'avvio, viene passato un puntatore a un contesto nel registro R1. Il contesto è fondamentalmente una struttura che assume un significato diverso a seconda del tipo di evento specifico a cui alleghiamo il programma eBPF ed è gestito direttamente dalla macchina virtuale eBPF.

l'ABI System V impone il protocollo per lo scambio di argomenti durante l'invocazione di una chiamata di sistema tra utente e kernel e lo scambio avviene tramite i registri della CPU. In particolare, la convenzione è:

* Le applicazioni a livello utente usano come registri interi per passare la sequenza %rdi, %rsi, %rdx, %rcx, %r8 e %r9.
* L'interfaccia del kernel usa %rdi, %rsi, %rdx, %r10, %r8 e %r9.

Il verificatore eBPF di eseguire un accesso alla memoria non valido. La soluzione è dereferenziare correttamente la memoria potenzialmente non sicura usando un accesso controllato.

1. eBPF HELPERS

Un helper eBPF permette di eseguire un accesso controllato alla memoria.

Oltre all'ambiente di esecuzione virtuale standard, eBPF offre anche la possibilità di chiamare un insieme fisso di funzioni del kernel, chiamate helper eBPF. Gli helper eBPF eseguono alcune operazioni per conto del programma eBPF, in modo nativo. Queste funzioni sono implementate all'interno del kernel in C, e sono quindi codificate e fanno parte dell'ABI del kernel.

Ogni helper eBPF è identificato da un numero intero univoco che è “set in stone” nel kernel ABI tramite enum.

La convenzione di chiamata eBPF impone che gli argomenti a una funzione helper debbano essere passati utilizzando i registri R1-R5, in sequenza.

R10 è un registro speciale ed è inizializzato dalla macchina virtuale automaticamente al “frame pointer” del programma eBPF. Indica la parte superiore dello stack che il programma eBPF può utilizzare per memorizzare le variabili locali. Lo stack ha una dimensione limitata a 512 byte.

Lo stack che ci fornisce l'ambiente virtuale eBPF è di soli 512 byte. Riservare una variabile di dimensione maggiore su di esso causerebbe sicuramente una violazione dello stack. E, sicuramente si sovrascriverebbe altra memoria del kernel se il programma dovesse essere eseguito, quindi è un'operazione non sicura. Se il compilatore non dovesse rilevare questa cosa, il verificatore eBPF avrebbe rilevato tale condizione e avrebbe impedito il caricamento del programma.

1. Ebpf maps

Bisogna archiviare i dati in una posizione diversa, fuori dallo stack. L'ambiente virtuale eBPF non ci fornisce la possibilità di allocare memoria esterna o utilizzare variabili globali come faremmo in un normale programma utente/kernel C. Tuttavia, si ha la possibilità di utilizzare le mappe eBPF. Le mappe eBPF sono strutture di dati chiave/valore accessibili dal programma eBPF tramite un set aggiuntivo di helper e sono persistenti tra le invocazioni. Il kernel offre diversi tipi di mappe (tabelle hash, array e altro). In questo modo ogni invocazione del programma eBPF avrà un proprio slot della mappa utilizzabile per tutta la durata del programma. Poiché i programmi eBPF non vengono mai anticipati durante la loro esecuzione, archiviarli in una mappa per CPU è sicuro e non può mai portare a race condition o corrupted data.

La definizione delle mappe va in una sezione ELF separata (generalmente all’inizio del programma) in modo che il caricatore eBPF possa rilevarla correttamente e configurarla (questo accade anche con la chiamata di sistema bpf). Le mappe si possono complicare aggiungendo altri campi, ad esempio lo spazio per altri argomenti di chiamata di sistema o il pid del processo.

Al compilatore piace riorganizzare i rami d’esecuzione in un modo che il verificatore non capisce ancora alla prima esecuzione del programma.

"L'arte di scrivere programmi eBPF" in contrapposizione a "la scienza".