Cos’è eBPF?

* Spiegazione anagramma eBPF
* Dove e quando nasce:
  + Berkeley
  + BSD packet filter: macchina virtuale nel kernel per analizzare pacchetti (documento descrive come è fatta)
  + Documento: sistema di filtraggio pacchetti molto efficiente: pacchetti analizzati direttamente nel kernel
* Limiti di BPF:
  + Poca flessibilità a fronte di evoluzioni tecnologiche di hardware, protocolli di rete e architetture di rete
  + Non c’erano meccanismi di sicurezza per eseguire codice utente nel kernel
* Introduzione ad eBPF
  + Più registri a 64 bit (per andare incontro alle nuove architetture)
  + Programmi usati per scopi diversi, non solo per filtraggio pacchetti -> maggiore flessibilità
  + Aggiunta di meccanismi di sicurezza
  + Nuova chiamata di sistema per chiamare le funzioni del kernel più rapidamente (si vedrà in seguito)
  + Programmi hanno una memoria limitata
* Risultato: eBPF è una tecnologia che permette di eseguire programmi in maniera sandbox (isolati) in un contesto privilegiato, quale il kernel di un sistema operativo.

Pertanto, eBPF fornisce un approccio sicuro ed efficiente per estendere le funzionalità del kernel senza la necessità di modificare il codice sorgente del kernel o caricare i moduli del kernel scritti con le API native del kernel.

* Al giorno d’oggi costituisce il nome di una tecnologia, ma di fatto non è ancora diventato uno standard.
* Casi d’uso tra i più svariati nell’ambito dell’osservabilità di sistemi (uso risorse, chiamate di sistema,… per fare cyber security) e di performance, di ambienti cloud e container, oltre che a continuare a fare da filtraggio di pacchetti.
* Però è stata fondata l’eBPF foundation: istituire un comitato (Google, Meta, Netflix,…) per guidare la crescita di eBPF -> espandere conoscenza, definire requisiti, creare una community

Processo di esecuzione di un programma

Ci sono diversi modi per iniziare ad usare eBPF

* Usare dei tool da linea di comando
* Scrivere bytecode eBPF
* Scrivere programmi in un linguaggio di programmazione di alto-livello e sfruttare dei compilatori appositi (Clang-LLVM estesi per programmi eBPF) per generare il bytecode eBPF

Una volta che si ha il bytecode eBPF, questo va caricato nel kernel:

* Verifica: il programma deve essere safe per essere eseguito all’interno del kernel
  + Direct Acyclic Graph check -> programma termina
  + Analisi di profondità del Control Flow Graph -> controllo di tutte le istruzioni (no accesso ad aree di memoria proibite e complessità finita del programma)

Lo fa in modo smart

Programmi con istruzioni irraggiungibili o loop infiniti o accedono a memoria inaccessibile non passano il check.

No soluzione ad Halting problem (il verificatore usa dei particolari costrutti di eBPF per dire se un programma termina o meno)

* Hardening: aggiunge misure di sicurezza
  + Memoria del kernel tenuta dal programma eBPF è resa read-only
  + Tentativi di modificare il programma fanno crashare il kernel
  + Costanti rese non modificabili per evitare JIT spraying attacks (come JS injection)
* Compilazione JIT
  + Traduce il bytecode eBPF verificato in codice macchina nativo corrispondente all'architettura della CPU di destinazione che può essere eseguita direttamente da processore
  + Risultato: programma efficiente come se fosse del codice kernel compilato in modo nativo e caricato come un LKM
* Caricamento ed esecuzione
  + Il programma è caricato nel kernel con una chiamata di sistema e viene eseguito

Peculiarità di un programma eBPF

Un programma eBPF scritto con un linguaggio di programmazione di alto livello deve sfruttare dei costrutti sintattici per permettere al verificatore di svolgere il suo lavoro:

* Dire in quale punto del kernel attaccare il programma ed il tipo di programma -> hook point
* Usare delle strutture dati per scambiare dati tra utente e kernel -> mappe (quasi tutte sono chiave-valore, tranne un tipo che permette all’utente solo di leggere da una zona di dati comune)
* Usare delle funzioni per chiamare le funzioni nel kernel (non si possono chiamare direttamente per impedire la correlazione stretta tra funzioni eBPF e funzioni kernel che dipenderebbero dal particolare kernel)
  + Alcuni helper per tipo di programma si possono chiamare
  + Non più di 5 argomenti
  + Non introducono overhead -> i programmi eBPF chiamano direttamente la versione compilata dell’helper
* Inoltre helper e mappe si usano in combinazione per performare le “tail call”: dato che i programmi eBPF hanno una memoria limitata, se si vuole usare un programma che richiede più memoria si usano helper e mappe particolari che ad un certo punto cambiano il contesto d’esecuzione da un programma al successivo

La chiamata di sistema

* Serve per caricare il programma nel kernel.
* È un’interfaccia fornita da Linux per permettere agli utenti di usare le funzionalità di eBPF all’interno del kernel
  + Header contiene la definizione di macro, funzioni e struct che si usano con eBPF
  + Cmd: operazione da svolgere (fornisce un API per dire dove il programma si attaccherà nel kernel e quali helper potrà chiamare)
  + Attr serve per passare dati da lato utente a lato kernel
  + Size è la dimenzione di attr.
* Non va chiamata direttamente nel codice: usiamo gli helper che durante il processo di compilazione JIT chiameranno indirettamente quella syscall

Il problema della portabilità

* Definizione (A.N.)
* Problema: struct e campi cambiano tra versioni del kernel successive
* Soluzione temporanea: BCC
  + Toolkit per creare programmi eBPF
  + Per assicurarsi il layout della memoria del kernel è lo stesso previsto dal programma eBPF, quando l'applicazione viene eseguita dall'host, BCC chiama il suo compilatore incorporato, che consiste nella combinazione Clang-LLVM, inserisce le intestazioni nel kernel e esegue la compilazione al volo.
  + Problemi:
    - Distribuire il compilatore insieme al programma (pesante e richiede molte risorse per essere eseguito)
    - Difficile lavorare in modo agile perché gli errori si hanno solo in compilazione

BPF CO-RE

* Insieme di cose per far funzionare il concetto
  + BPF Type Format (BTF), un formato di metadati compatto ma sufficientemente espressivo che descrive le informazioni dei programmi C e viene utilizzato per migliorare le capacità del verificatore;

Ciò riduce la necessità di ricompilare i programmi diverse versioni del kernel, rendendo i programmi eBPF più portabili ed efficienti.

* + Un supporto per il compilatore, come Clang, un compilatore front-end per programmi C e C++, esteso con built-in che consentissero la cattura di campi o set, esistenza e dimensione, dimensione del tipo e rilocazione, valori enumerazione ed esistenza (accedere ad un campo con solo il nome).
  + Il kernel compatibile con CO-RE.

Servono le informazioni BTF del kernel: si trovano nel file vmlinux.h che contiene tutti i tipi usati nel codice sorgente del kernel in esecuzione sulla macchina

Dopo che abbiamo le 3 cose, un loader, chiamato libbpf, prende il file oggetto BPF generato dopo il file processo di compilazione del programma e innesca le fasi di caricamento e verifica;

consente la portabilità dei programmi eBPF risolvendoli e abbinandoli a tutti i tipi e i campi nel kernel e aggiorna necessari offset e altri dati riposizionabili.

* Libbpf: libreria per interagire con l’ecosistema eBPF nel kernel Linux -> implementazione BPF CO-RE
  + Permette agli utenti di preoccuparsi solo del codice, gestendo le relocations, verifica e caricamento nel kernel, creazione di mappe, hook points,…
  + Astrae molti concetti tramite la sua API per creare programmi eBPF
  + Richiede di conoscere solo le informazioni BTF del kernel e del programma

Risultato: programmi compilati una volta possono essere eseguiti su ogni kernel

* + Inoltre fornisce gli skeleton header files, un’interfaccia alternativa alle API di libbpf per gestire gli programmi eBPF dal lato utente (open, load, attach, tear down)

Linux: BumbleBee

* Progetto open source incentrato sulla semplificazione dell'esperienza dell'utente nella creazione di strumenti eBPF.
* Aiuta gli sviluppatori a creare, eseguire e distribuire programmi eBPF utilizzando il concetto di container tramite immagini Open Container Initiative (OCI).
* Basta installare una CLI su VM Ubuntu
* Scelte e creazione programma
* Limiti: solo 2 hook points e 2 mappe e metriche come output
* Usa dei costrutti per gestire in modo automatico il lato utente che non esistono se si scrivono programmi con libbpf

Linux: libbpf-bootstrap

* Progetto open source creato per setuppare più cose possibile per i novizi di eBPF
* Basta clonare un repository
* Processo
  + Scrittura programma eBPF
  + Compilazione: generazione file oggetto e skeleton file (un’interfaccia alternativa alle API di libbpf per gestire i programmi eBPF dal lato utente (open, load, attach, tear down))
  + Importare skeleton file nel programma lato utente per fare ciclo di vita del programma
  + Avviare programma

Windows: ebpf-for-windows

* eBPF ha raggiunto anche Windows con il progetto
* Sfrutta progetti open source esistenti che vengono eseguiti on top of Windows e aggiunge cose specifiche per l’ambiente Windows
* Obiettivo: fornire compatibilità per sistemi operativi diversi (ovviamente non per cose specifiche come Linux che ha molti hook points e helpers)
* Espone API libbpf per fornire codice sorgente compatibile per applicazioni che interagiscono con programmi eBPF
* Consentire agli sviluppatori di utilizzare le toolchain e le API eBPF su Windows.
* Basandosi sul lavoro di altri, questo progetto prende diversi progetti open source eBPF esistenti e aggiunge il "collante" per farli funzionare su Windows.
* Come mostrato nel diagramma, le toolchain eBPF esistenti come clang possono essere utilizzate per generare bytecode eBPF dal codice sorgente in varie lingue.
* Il bytecode risultante può quindi essere utilizzato da qualsiasi applicazione o manualmente tramite lo strumento da riga di comando netsh di Windows, che utilizzano entrambi una libreria condivisa che espone le API Libbpf, sebbene questo lavoro sia ancora in corso.
* La libreria invia il bytecode eBPF a un verificatore statico (il verificatore PREVAIL) che è ospitato in un processo protetto in modalità utente, che è un ambiente di sicurezza Windows che consente a un componente del kernel di considerare attendibile un demone in modalità utente firmato da una chiave che esso si fida.
* Se il bytecode supera tutti i controlli di sicurezza del verificatore, il bytecode può essere caricato nell'interprete uBPF in esecuzione in un contesto di esecuzione in modalità kernel di Windows o compilato dal compilatore just-in-time uBPF e avere il codice nativo caricato in modalità kernel contesto di esecuzione.
* I programmi eBPF installati nel contesto di esecuzione in modalità kernel possono collegarsi a vari hook per gestire eventi e chiamare varie API di supporto esposte dallo shim eBPF, che racchiude internamente le API pubbliche del kernel di Windows, consentendo l'uso di eBPF su versioni esistenti di Windows.
* Finora sono stati aggiunti due hook (XDP e socket bind) e, sebbene si tratti di hook specifici della rete, prevediamo che nel tempo verranno aggiunti molti altri hook e helper, non solo relativi alla rete.

Difficoltà nel setup dell’ambiente non indifferente: un ambiente adatto per lo sviluppo in Windows con eBPF è molto più difficile che in Linux (PUNTO CRITICO)

* Linux: clonare repo e assicurarsi che ci siano le informazioni BTF del kernel
* Windows: processo installazione VM complesso

Windows è progettato per garantire la sicurezza e la stabilità del sistema operativo.

* + O i driver sono firmati da Microsoft
  + O si usa una VM in modalità di test e si ha un certificato di test per firmare i driver e caricarli in modalità di test (non è certificato valido, ma va bene in test e non in produzione).
  + O si avvia la VM con un KD
* Primi 2 Requisiti progettati per proteggere la stabilità e la sicurezza del sistema operativo Windows.

Rischio di compromettere la sicurezza del sistema in produzione.

La firma digitale dei driver fornisce un meccanismo di controllo dell'integrità che aiuta a garantire che i driver siano affidabili e non compromettano il funzionamento del sistema operativo (evitare exploit -> sfruttare bug per creare comportamenti non previsti)

* KD per diversi motivi: Sviluppo e debugging + tracciamento e analisi + sicurezza + risoluzione problemi in tempo reale dei programmi eBPF in esecuzione nel kernel di Windows.

Windows: windows-ebpf-starter

* Progetto open source che vuole essere come libbpf-bootstrap ma per windows
* Richiede comunque il setup dell’ambiente
* Lato kernel funziona come spiegato da ebpf-for-windows
* Lato utente dovrebbe automatizzare il ciclo di vita di un programma eBPF senza sfruttare degli skeleton file, ma usando dei metodi per manipolare oggetti eBPF presenti all’interno dell’API di ebpf-for-windows.

Inoltre ci sono dei problemi con i vari “include” degli header file perché il progetto non riconosce quelli al di fuori della libreria ebpf-for-windows.