Sécurité des SI - Devoir 1

Gwennan Eliezer, Léo Marché et Syméon Smith

16 décembre 2022

Table des matières

1 Présentation de la vulnérabilité

La vulnérabilité choisie est la CVE-2022-23222.

1.1 Service compromis

Cette vulnérabilité se situe dans la bibliothèque eBPF installée dans le noyau Linux. Cette bibliothèque permet aux utilisateurs d'exécuter du code dans le noyau pour filtrer les paquets arrivant sur les interfaces réseaux d'un ordinateur, aussi bien client que serveur. Elle est notamment utilisée dans libpcap, la bibliothèque sur laquelle des logiciels comme Wireshark se basent dans des environnements Linux.

1.2 Description de la vulnérabilité et de l'attaque

Pour toute question relative aux fonctions, structures, syscalls, ... utilisés, se référer au Glossaire en fin de document.

1.2.1 Description de la vulnérabilité

Pour assurer la sécurité pendant l'exécution de code dans eBPF, le noyau Linux implémente un vérificateur, qui s'assure que l'exécution du code eBPF respecte certaines règles :

- Vérification des pointeurs (les registres ne contiennent pas des pointeurs pouvant pointer vers des endroits interdits en mémoire)
- Contrôle des lectures/écritures sur le stack (comme celui-ci n'est pas initialisé à 0, on veut éviter la fuite de donnée due à des lectures sans écritures)
- Interdiction d'écriture de pointeurs sur le stack.
- ...

Pour ce faire, le vérificateur prédit le résultat des opérations sur les registres et s'assure qu'elles respectent toutes certaines règles. La vulnérabilité se situe dans le fait qu'il y a toutefois un certain type d'opérations que le vérificateur ne traque pas : l'arithmétique de pointeur sur des valeurs de registres de type `PTR_TO_<SMTHING>_OR_NULL` (par exemple `PTR_TO_MEM_OR_NULL`). Ce type de valeurs est utilisé dans eBPF pour des opérations dont les résultats sont incertains. Afin de s'assurer que les résultats de telles opérations soient soit `NULL`, soit un pointeur `PTR_TO_<SMTHING>` sain, le vérificateur fournit une fonction qui permet de vérifier le type de ces pointeurs et de vérifier leur sûreté si ce sont des pointeurs de type `PTR_TO_<SMTHING>`. Si cette fonction n'est pas lancée sur le pointeur ou que le pointeur n'est pas sûr, le pointeur est inutilisable.

Afin de comprendre la faille, il faut aussi savoir que lorsqu'un registre, nommons le $R1$, est copié dans un autre registre, disons $R2$, son type est lui aussi copié. Si une vérification est appliquée à $R1$ et que $R2$ n'est pas modifié, alors $R2$ est considéré comme ayant le même type et la même valeur que $R1$.

Mais vu que le vérificateur ne traque pas l'arithmétique de pointeur sur les types ressemblant à `PTR_TO_<SMTHING>_OR_NULL`, si $R1$ était `PTR_TO_<SMTHING>_OR_NULL`, le vérificateur va considérer que $R2$ n'est pas modifiée même si on lui a appliqué entre temps des opérations arithmétiques. On peut donc facilement réussir à mettre `NULL` dans $R1$ avec le type `PTR_TO_<SMTHING>_OR_NULL`, le copier dans $R2$, le modifier pour y mettre une valeur arbitraire en utilisant de l'arithmétique de pointeurs, et lancer la vérification sur le registre $R1$ pour avoir le type `NULL`, faisant ainsi croire au vérificateur que le registre $R2$ est `NULL` et ainsi outrepasser la vérification des pointeurs d'eBPF sur $R2$.

Cela, combiné au fait qu'eBPF permet l'exécution de code avec des privilèges noyaux et qu'il y a des fonctions d'eBPF permettant d'écrire ou de lire dans la mémoire à partir d'adresses situées dans les registres, permet de faire des écritures et des lectures arbitraires dans l'espace mémoire du noyau, permettant de réécrire l'UID/GID d'un processus et donc de réussir une élévation de privilèges.

1.2.2 Description de l'attaque

Notre objectif est d'arriver à lire et écrire de manière arbitraire dans la mémoire. Pour cela, on utilise un processus en plusieurs étapes [2] :

1. Allocation de deux `bpf_map` contiguës :
   On demande à eBPF de nous donner deux `bpf_map` : A et B. À ce stade, on ne sait pas si elles sont contiguës ou pas, toutefois, on peut utiliser la fonction `bpf_ringbuf_output` ainsi que la vulnérabilité pour s'en assurer. Pour ce faire, on écrit une valeur aléatoire R dans A, puis on utilise la vulnérabilité et `bpf_ringbuf_output` pour faire une lecture hors limite de B. On essaye de lire là où la valeur R se trouverait si A était allouée juste après B. Si on retrouve la valeur aléatoire, on considère que c'est un succès, sinon c'est un échec. On recommence ensuite en inversant les rôles.

Finalement, si aucun des deux ne fonctionne, cela veut dire que A et B ne sont pas contiguës et on recommence tout jusqu'à ce que B et A soient contiguës.

2. Réécriture d'une `bpf_map` :
   Maintenant qu'on a deux `bpf_map` contiguës, A et B avec B placé juste après A, on peut utiliser la fonction `bpf_skb_load_bytes` pour réaliser une écriture hors limite sur la struct de B.
3. Lecture arbitraire en mémoire :
   Pour réussir à lire de manière arbitraire, on pourra réécrire l'attribut `btf` de B en utilisant une écriture hors limite et en mettant un pointeur arbitraire à la place. Ensuite, on utilisera le syscall `BPF_OBJ_GET_INFO_BY_FD` sur B. Cela ira lire le contenu du champ `btf` de B et le copiera dans un buffer, permettant de récupérer le contenu de la mémoire à notre adresse arbitraire.
4. Écriture arbitraire en mémoire :
   Pour réussir à écrire de manière arbitraire, on pourra réécrire l'attribut `ops` de B en utilisant une écriture hors limite. On commencera par le copier et par faire pointer `B->ops` vers notre copie. Puis en modifiant certaines fonctions et certains attributs de notre copie, on arrive à utiliser le syscall `BPF_MAP_UPDATE_ELEM` pour écrire arbitrairement dans la mémoire.
5. Élévation de privilèges :
   Soit on utilise une valeur standard (valable uniquement pour une version de noyau donnée), soit on utilise la lecture arbitraire pour trouver l'adresse d'`init_pid_ns`, le namespace dans lequel est notre processus (en utilisant les symboles `kstrtab` et `ksymtab` exportés par le noyau). Ensuite, on trouve l'adresse de la structure `task_struct` associée au pid de notre programme en cherchant dans `init_pid_ns`. Une fois notre `task_struct` trouvée, on utilise l'écriture arbitraire pour réécrire l'uid et le gid dans l'attribut `cred`. En mettant ces deux valeurs à 0, on récupère des privilèges root (root étant l'utilisateur ayant pour uid 0 et/ou le groupe ayant pour gid 0).

Le processus utilisé est illustré dans une Proof of Concept commentée par nos soins sur Github [3].

1.2.3 Type de machines concernées

Cette faille peut concerner les machines clientes et les machines serveurs, étant donné que eBPF est présent dans les noyaux des versions clientes et serveurs de distributions très connues (Ubuntu/Debian).

Une architecture typique de SI qui pourrait être impliqué dans l'exploitation de la faille serait un serveur web, sur lequel un attaquant aurait réussi à récupérer un accès utilisateur non privilégié (par exemple `www-data`, l'utilisateur par défaut pour Apache2, Nginx, etc.) en utilisant une faille web. En exploitant la faille eBPF, il pourra ensuite réaliser une élévation de privilège pour récupérer des droits administrateurs. Cela pourrait permettre d'accéder à des clefs privées, des fichiers sensibles de l'administrateur et potentiellement de compromettre plus que la machine par laquelle l'attaquant est entré.

2 Préconisations de sécurité

2.1 Empêcher l'exploitation

Pour empêcher l'exploitation de la vulnérabilité, il est possible de réaliser plusieurs choses :
- Mettre à jour la distribution et le noyau, permettant de limiter par défaut l'usage d'eBPF aux utilisateurs privilégiés.
- Utiliser un noyau sans eBPF.
- Désactiver manuellement eBPF pour les utilisateurs non privilégiés :
  `echo '1' >> /proc/sys/kernel/unprivileged_bpf_disabled`.

2.2 Limitation de l'impact

Pour limiter l'impact de la vulnérabilité, il faut passer en revue tous les programmes utilisant eBPF qui tournent sur une machine. Tous les programmes sûrs et vitaux doivent être transférés à un utilisateur privilégié et on doit ensuite désactiver eBPF pour les utilisateurs non privilégiés.

2.3 Bonnes pratiques pertinentes

Une bonne pratique pertinente est le principe du moindre privilège : l'immense majorité des personnes n'ont pas besoin d'exécuter eBPF en tant qu'utilisateur lambda. Cela devrait donc être désactivé par défaut.

3 Extrait de PSSI fictif

Toutes les machines exécutant un noyau Linux doivent être monitorées. Les machines sur lesquelles il est possible de réaliser des mises à jour régulières de noyaux doivent être maintenues à jour autant que possible via des canaux de sécurité. Pour les autres, la valeur de `/proc/sys/kernel/unprivileged_bpf_disabled` doit être mise à 1 et doit être monitorée. Un changement dans cette valeur doit alerter sur une possible intrusion sur la machine.

4 Démonstration

La démonstration est effectuée sur une machine virtuelle créée avec Vagrant, tournant sur Ubuntu 20.04 avec le noyau Linux `5.13.0-27-generic`. La faille est présente aussi sur d'autres versions du noyau. Le fichier `Vagrantfile` liste les commandes lancées à la création de la machine virtuelle, avec les étapes suivantes :

- installer les outils nécessaires à la compilation du C ainsi que la bibliothèque `libbpf` ;
- cloner et compiler le dépôt de l'*exploit* : celui utilisé est un Proof of Concept disponible sur Github [1] ;
- installer la bonne version du noyau ;
- mettre la machine dans la configuration vulnérable (cette configuration est celle par défaut sur certaines installations d'Ubuntu) en changeant la valeur de la variable située dans `/proc/sys/kernel/unprivileged_bpf_disabled` à 0 ;
- redémarrer la machine pour appliquer le changement de noyau.

Lorsque la machine est lancée (`vagrant up` puis `vagrant ssh` pour y accéder), un exécutable `exploit` se situe à la racine. Une fois lancé (`./exploit`), cet exécutable effectue automatiquement la procédure d'exploitation de la faille d'eBPF et donne accès à un shell en mode administrateur, sans restriction. Par exemple, en lançant la commande `id`, on voit que le processus courant possède l'uid 0, montrant bien que l'exploit a effectué une élévation de privilèges.

Glossaire

— `bpf_map` : format de stockage de données permettant le partage entre espace utilisateur et espace noyau. Cela permet (théoriquement) de sécuriser l'accès à des données situées en espace noyau.
— `btf` : attribut de `bpf_map` ; format de métadonnées encodant des informations de debug d'un `bpf_map`.
— `ops` : attribut de `bpf_map` ; structure contenant les fonctions effectuant des opérations sur un `bpf_map`.
— `bpf_skb_load_bytes` : fonction utilitaire de eBPF permettant de récupérer une partie d'un paquet, en indiquant la taille et l'offset de la mémoire à récupérer.
— syscall : appel système (fonction offerte par le noyau du système d'exploitation).
— `BPF_OBJ_GET_INFO_BY_FD` : syscall eBPF permettant de récupérer les informations d'un `bpf_map` à partir de son descripteur de fichier.
— `BPF_MAP_UPDATE_ELEM` : syscall eBPF permettant de créer ou mettre à jour un élément (paire clé/valeur) dans une `bpf_map`.
— namespace : fonctionnalité de Linux permettant d'isoler des ressources et des processus.
— `init_pid_ns` : structure permettant de connaître le namespace d'un processus.
— `kstrtab`, `ksymtab` : symboles du noyau Linux permettant de récupérer l'adresse de `init_pid_ns`.
— `task_struct` : structure contenant toutes les informations relatives à l'exécution d'un processus.
— `cred` : membre de `task_struct`, structure contenant les informations relatives à l'identification d'un processus (uid, gid, ...)

Références

[1] Matan Liber. Proof of Concept Exploit Code for CVE-2022-23222. https://github.com/PenteraIO/CVE-2022-23222-POC, Juin 2022.

[2] Matan Liber. The Good, Bad and Compromisable Aspects of Linux eBPF. https://pentera.io/blog/the-good-bad-and-compromisable-aspects-of-linux-ebpf/, Juin 2022.

[3] Léo Marché. Proof of Concept Commented Exploit Code for CVE-2022-23222. https://github.com/LeoMarche/CVE-2022-23222-POC, December 2022.