SEQUOIA: A DEEP ROOT IN LINUX'S FILESYSTEM LAYER

Duzés Florian

CVE-2021-33909 — 21 novembre 2024

SOMMAIRE

Aperçu général

Vulnérabilité : size_t -> int

- Création d'un répertoire avec un chemin > 1Go
- Monter ce répertoire dans un espace utilisateur
- Suppression dudit répertoire
- Écriture d'un chaine de caractère hors limite
- Exploitation et accés root

Systèmes testés : Ubuntu 20.04, Ubuntu 20.10, Ubuntu 21.04, Debian 11 et Fedora 36

- Introduite en Juillet 2014
  - Linux 3.16 - 058504ed "fs/seq_file : fallback to vmalloc allocation"
- Découverte en Juin 2021
- Patch en Juillet 2021
  - Correction apporté par : Red Hat Product Security

Idée de l'attaque

GESTION MÉMOIRE DES FICHIERS VIRTUELS


```
168 ssize_t seq_read_iter(struct kiocb *iocb, struct iov_iter
↪  *iter){
170         struct seq_file *m = iocb->ki_filp->private_data;
205         /* grab buffer if we didn't have one */
206         if (!m->buf) {
207                 m->buf = seq_buf_alloc(m->size = PAGE_SIZE);
...
210         }
...
220         // get a non-empty record in the buffer
...
223         while (1) {
...
227                 err = m->op->show(m, p);
...
236                 if (!seq_has_overflowed(m)) // got it
237                         goto Fill;
238                 // need a bigger buffer
...
240                 kvfree(m->buf);
242                 m->buf = seq_buf_alloc(m->size <<= 1);
246         }
```

GESTION MÉMOIRE DES FICHIERS VIRTUELS


```
135 static int show_mountinfo(struct seq_file *m, struct vfsmount *mnt)
136 {
...
150                 seq_dentry(m, mnt->mnt_root, " \t\n\\");
```

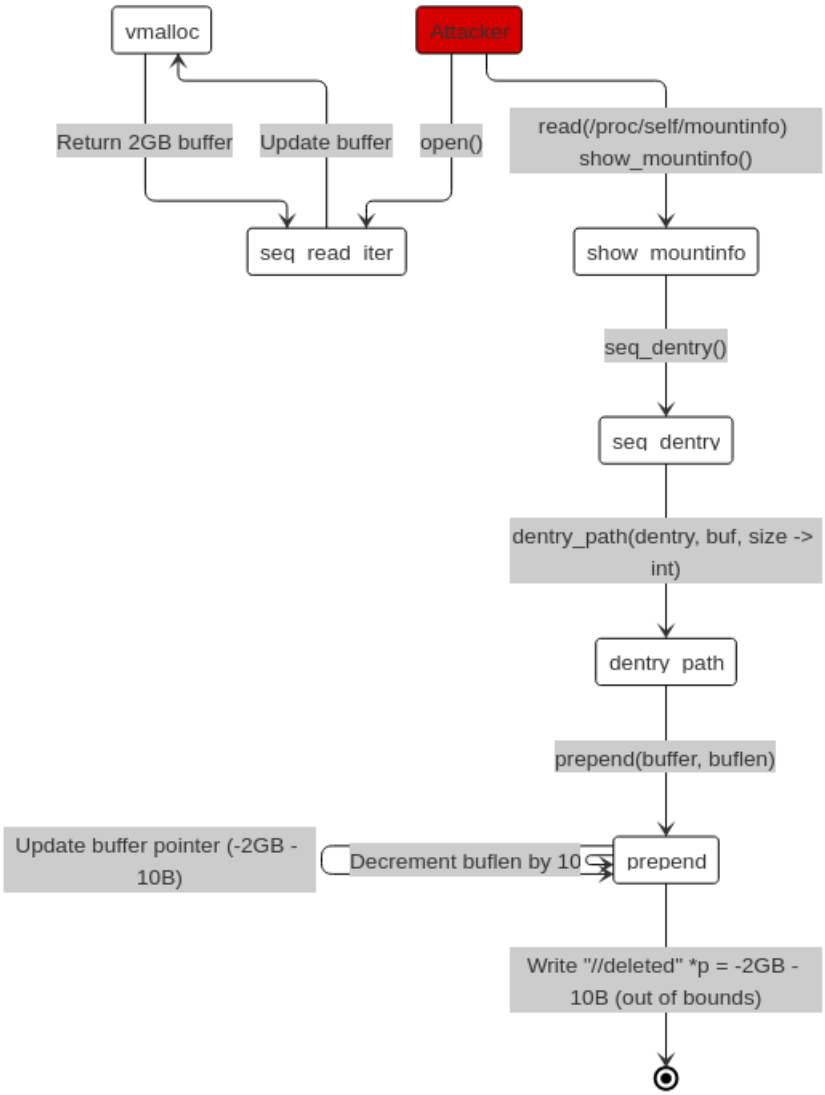
```
523 int seq_dentry(struct seq_file *m, struct dentry *dentry, const
↪   char *esc)
524 {
525         char *buf;
526         size_t size = seq_get_buf(m, &buf);
...
529         if (size) {
530                 char *p = dentry_path(dentry, buf, size);
```

GESTION MÉMOIRE DES FICHIERS VIRTUELS


```
380 char *dentry_path(struct dentry *dentry, char *buf, int buflen)
381 {
382         char *p = NULL;
...
385         if (d_unlinked(dentry)) {
386                 p = buf + buflen;
387                 if (prepend(&p, &buflen, "//deleted", 10) != 0)
```

```
 11 static int prepend(char **buffer, int *buflen, const char *str, int
↪  namelen)
 12 {
 13         *buflen -= namelen;
 14         if (*buflen < 0)
 15                 return -ENAMETOOLONG;
 16         *buffer -= namelen;
 17         memcpy(*buffer, str, namelen);
```

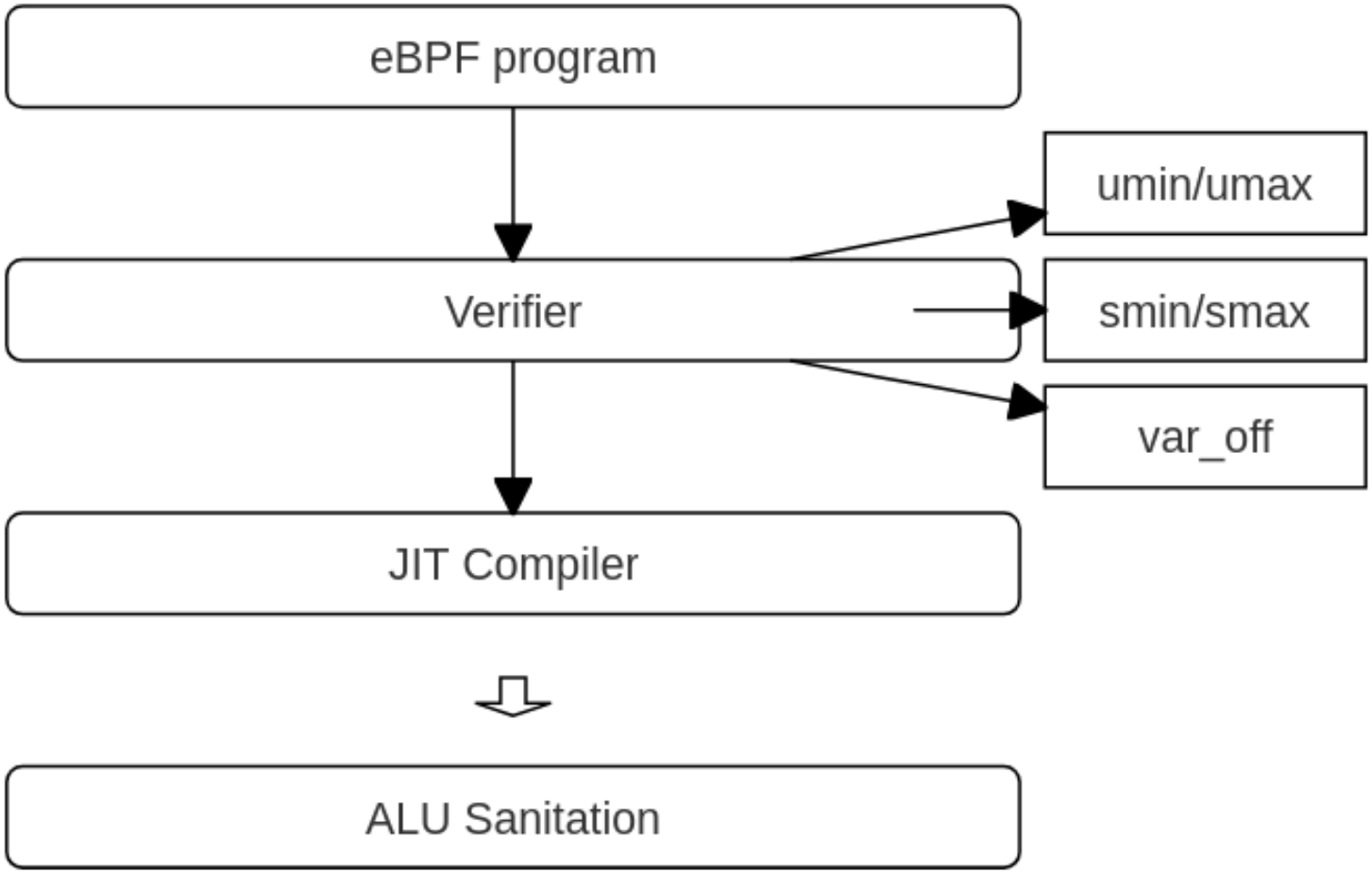
GRAPHIQUE DES OPÉRATIONS



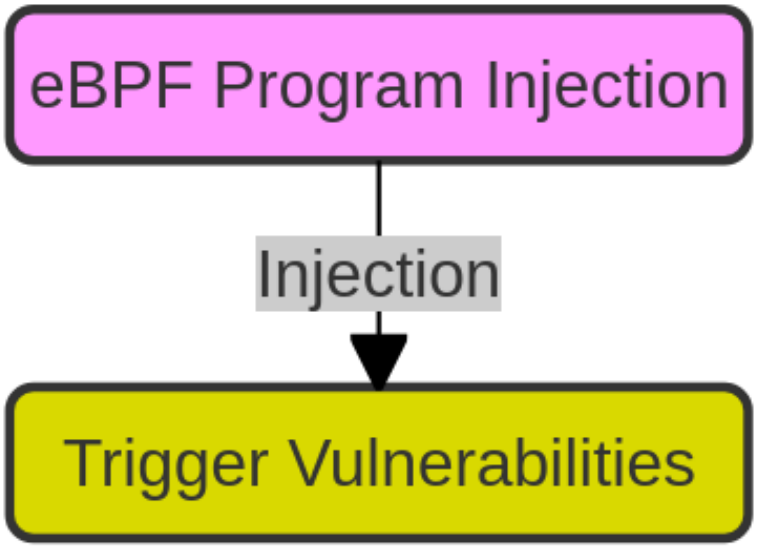
CVE-2020-8835 : Linux Kernel Privilege Escalation via Improper eBPF Program Verification

eBPF - Introduction

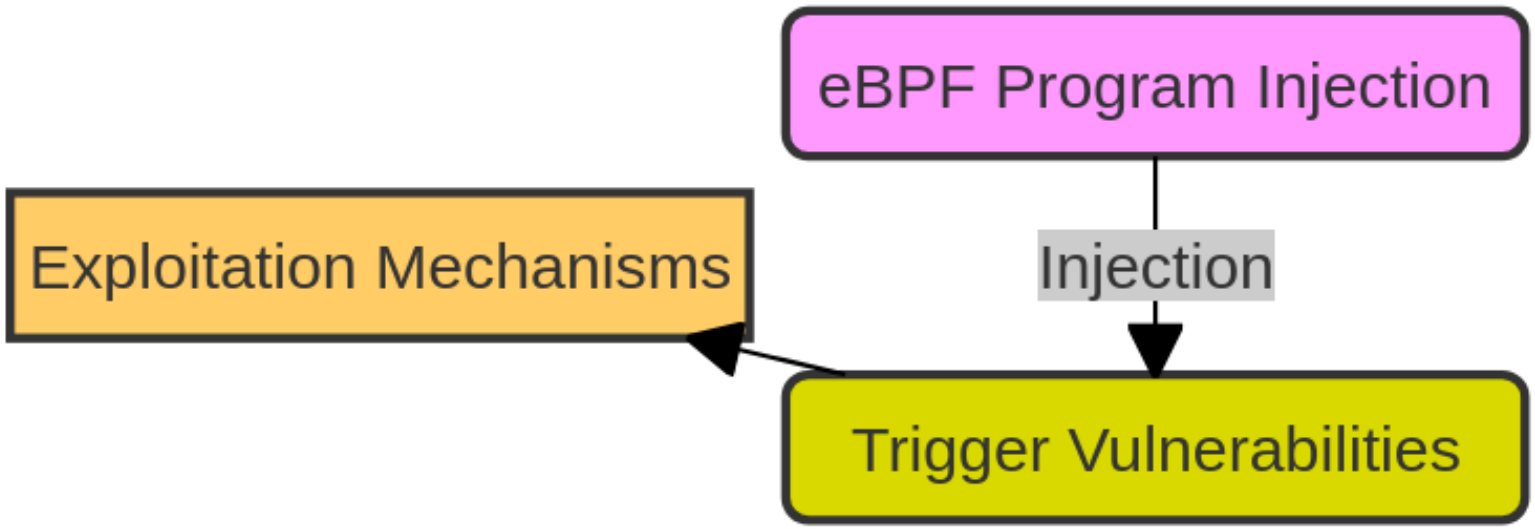
COMMENT ÇA MARCHE

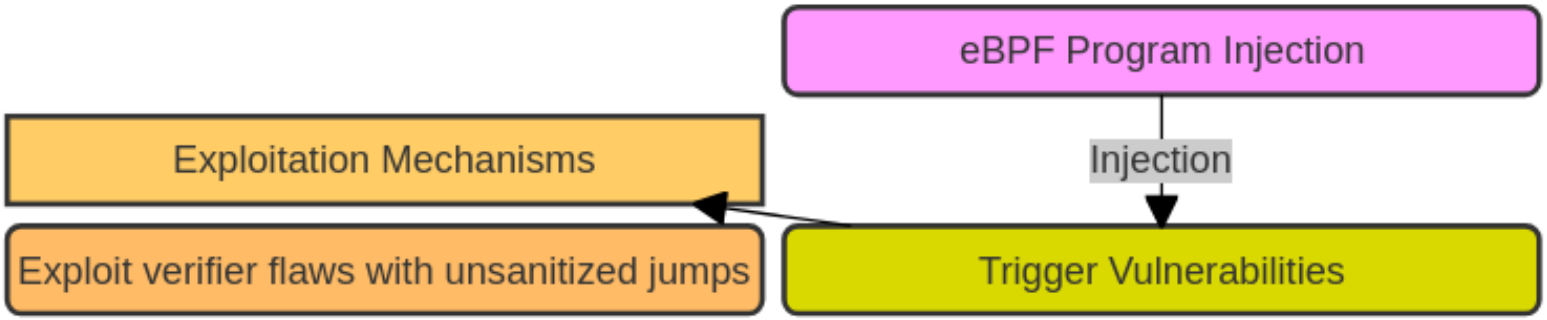


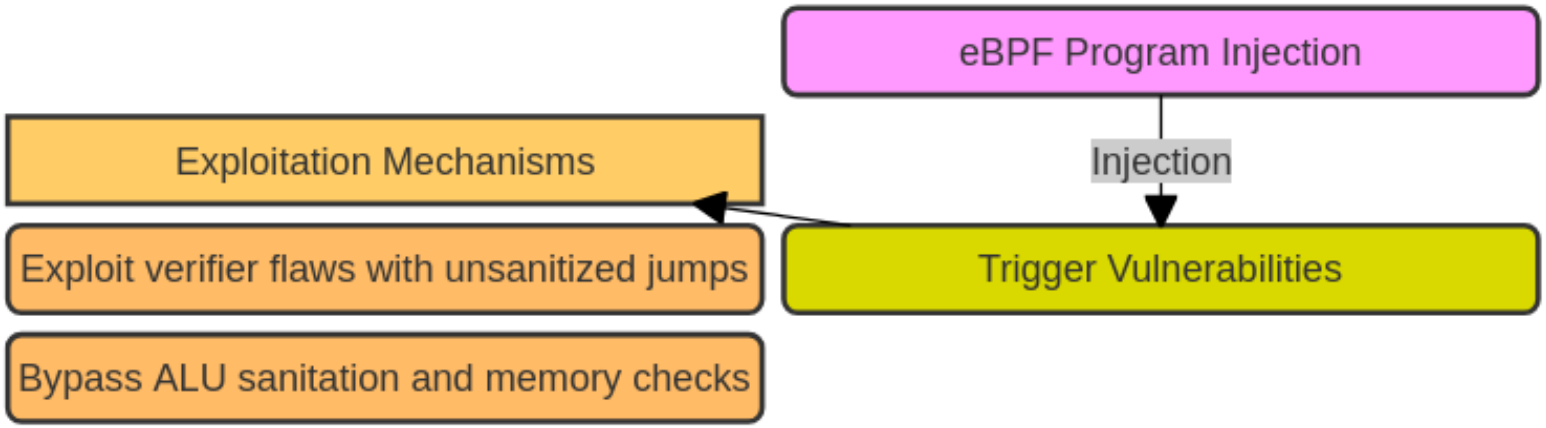

eBPF Program Injection

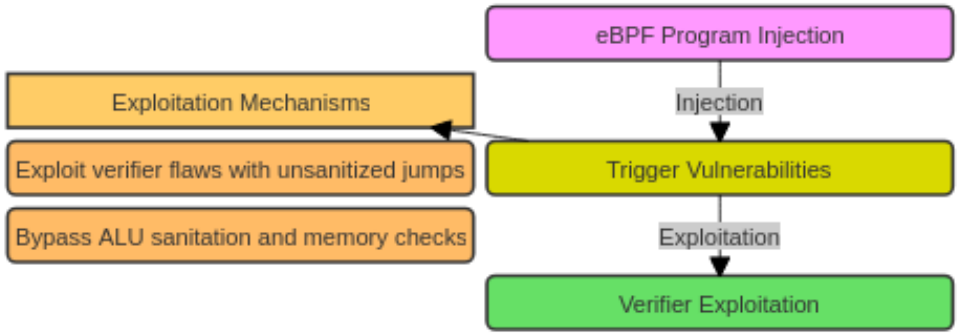

eBPF Program Injection
Injection
Trigger Vulnerabilities

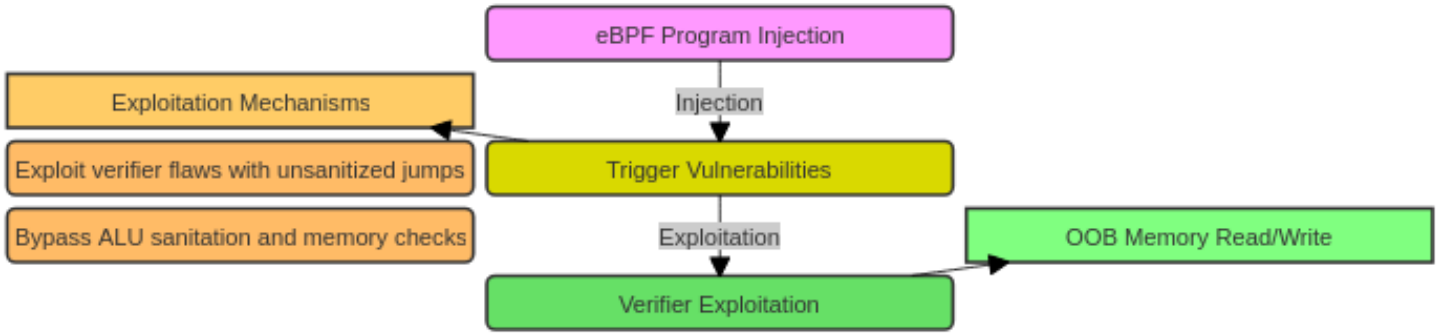
L'ATTAQUE

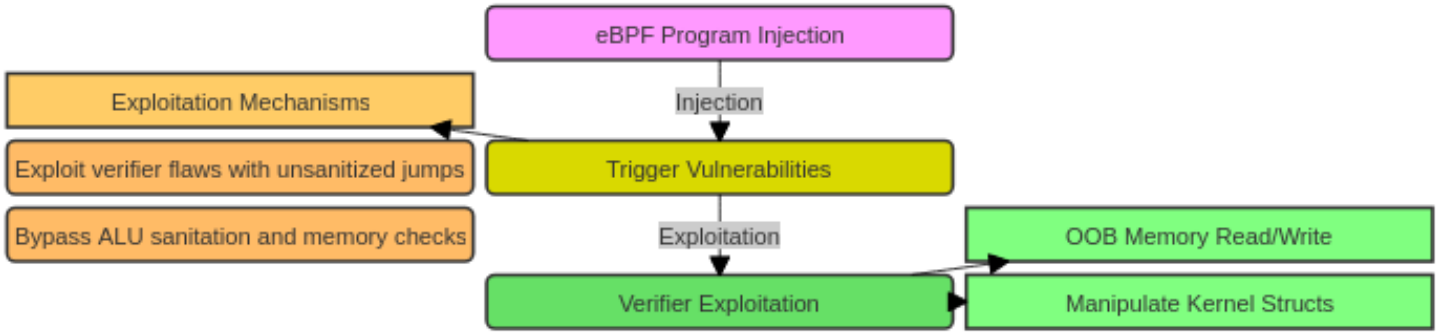

eBPF Program Injection
Injection
Exploitation Mechanisms
Trigger Vulnerabilities

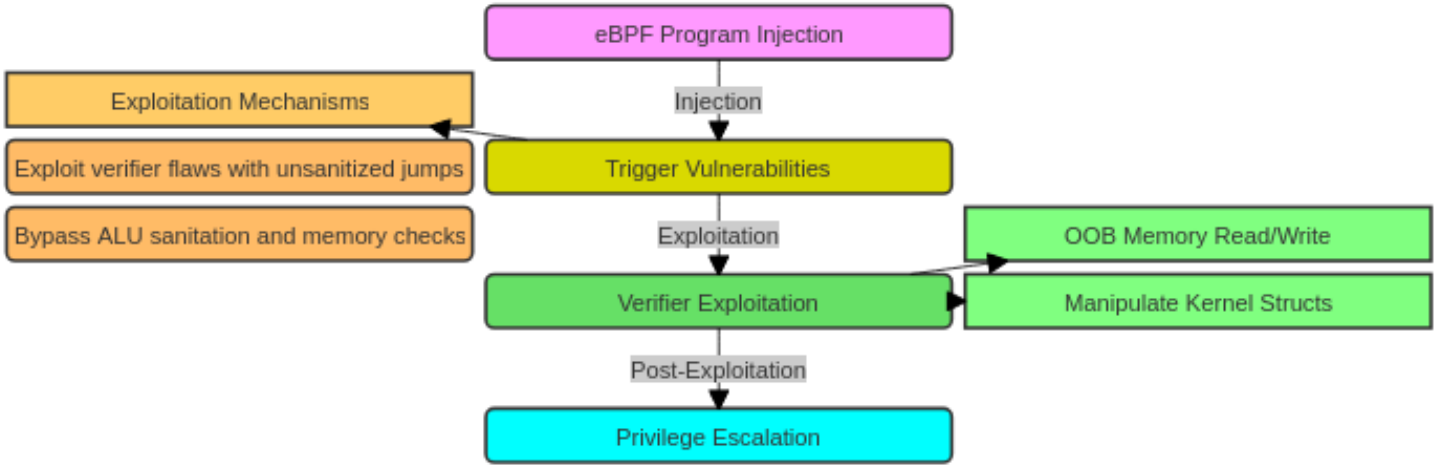

eBPF Program Injection
Injection
Exploitation Mechanisms
Exploit verifier flaws with unsanitized jumps
Trigger Vulnerabilities

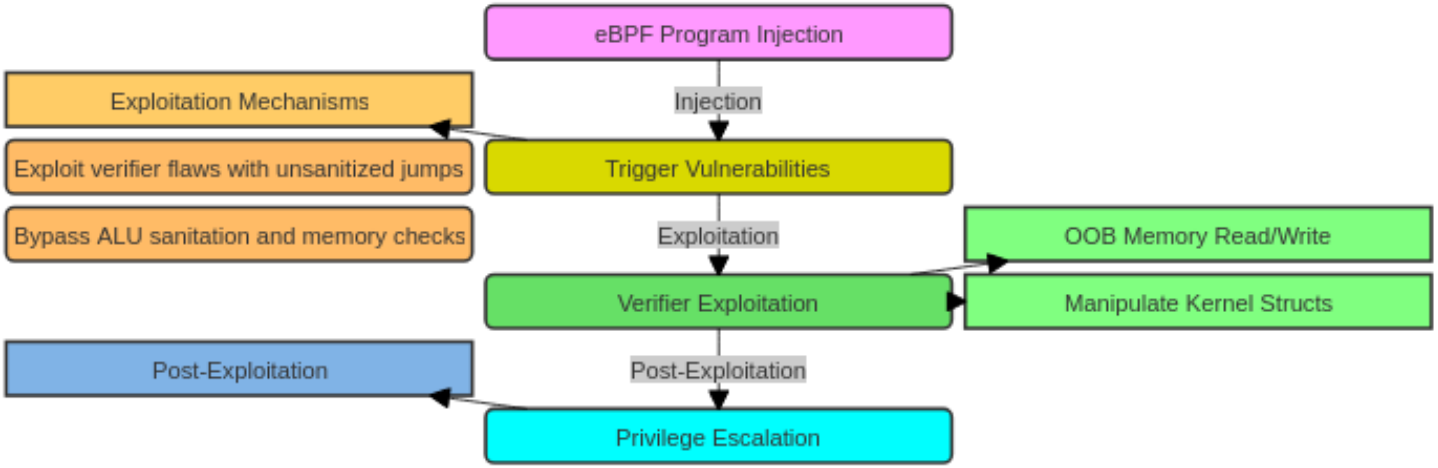

eBPF Program Injection
Injection
Exploitation Mechanisms
Exploit verifier flaws with unsanitized jumps
Bypass ALU sanitation and memory checks
Trigger Vulnerabilities

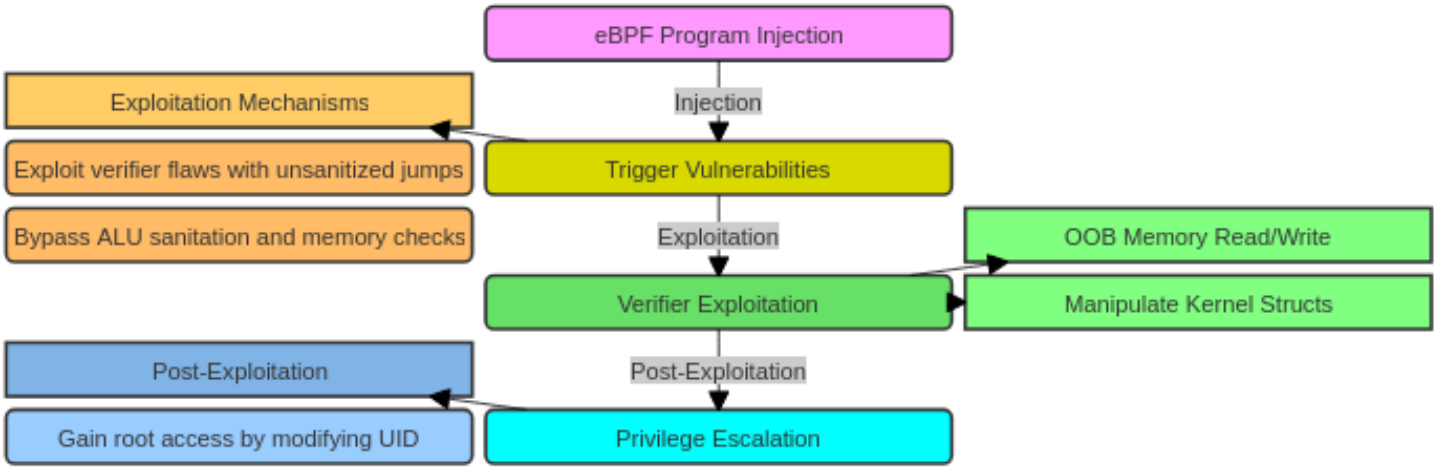

eBPF Program Injection
Injection
Exploitation Mechanisms
Exploit verifier flaws with unsanitized jumps
Bypass ALU sanitation and memory checks
Trigger Vulnerabilities
Exploitation
Verifier Exploitation


eBPF Program Injection
Injection
Exploitation Mechanisms
Exploit verifier flaws with unsanitized jumps
Bypass ALU sanitation and memory checks
Trigger Vulnerabilities
Exploitation
OOB Memory Read/Write
Verifier Exploitation

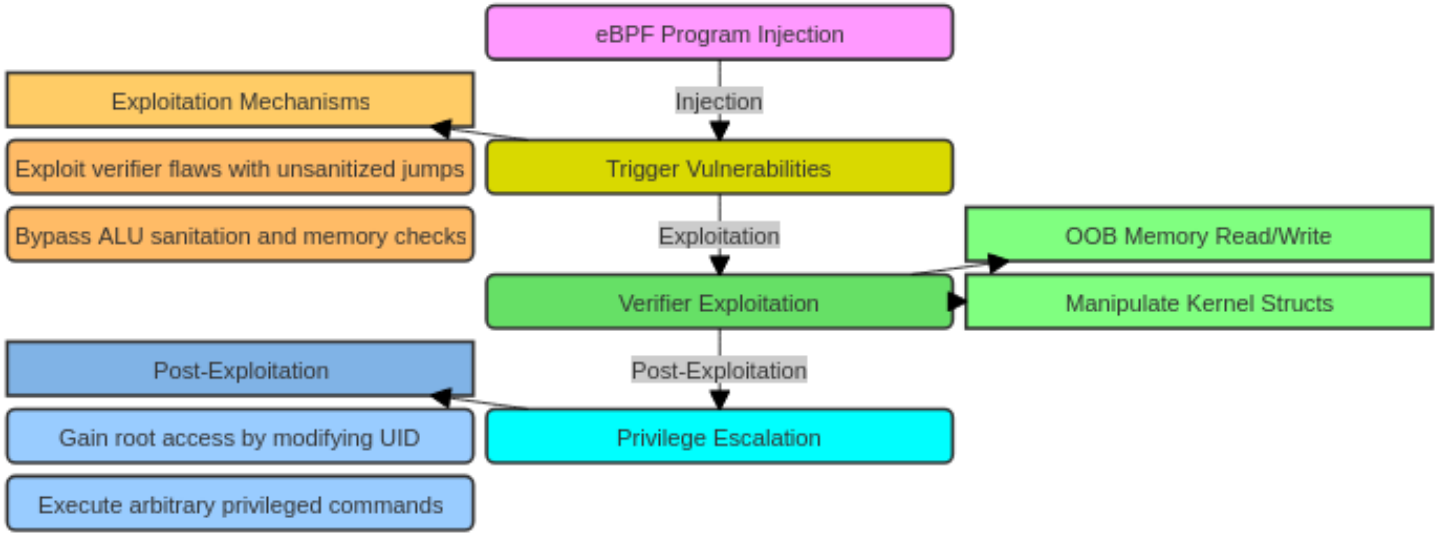

eBPF Program Injection
Injection
Exploitation Mechanisms
Exploit verifier flaws with unsanitized jumps
Bypass ALU sanitation and memory checks
Trigger Vulnerabilities
Exploitation
OOB Memory Read/Write
Verifier Exploitation
Manipulate Kernel Structs

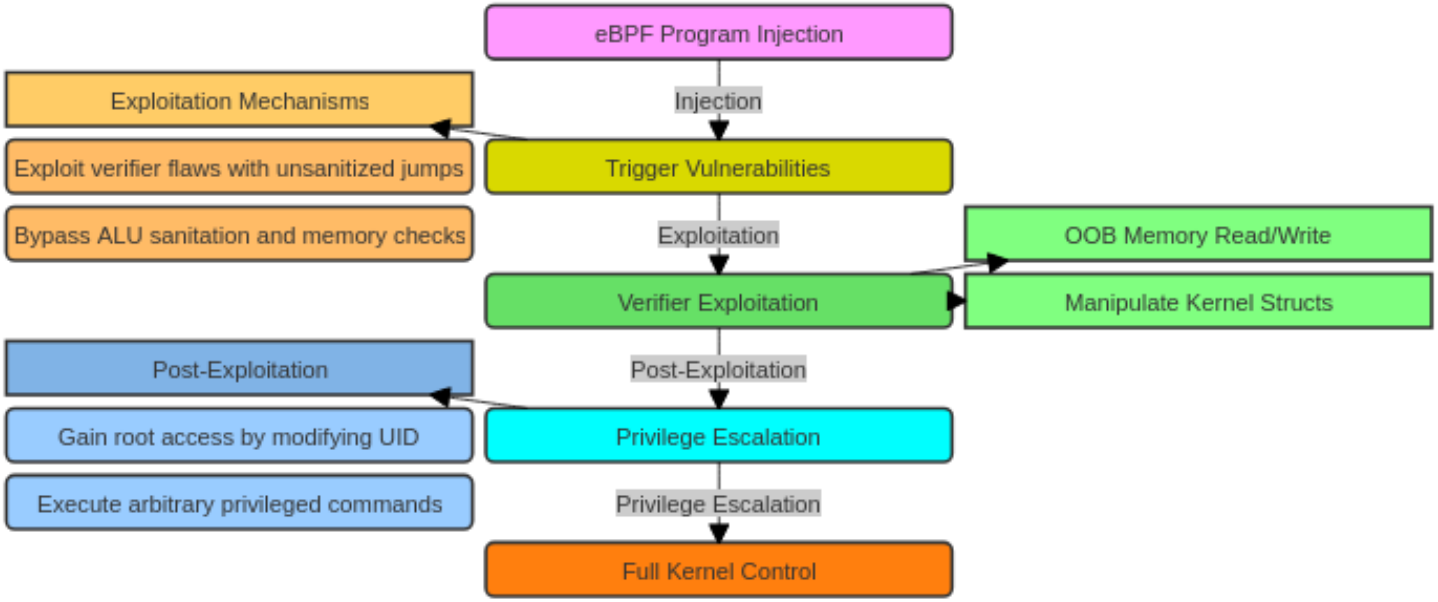

eBPF Program Injection
Injection
Exploitation Mechanisms
Exploit verifier flaws with unsanitized jumps
Bypass ALU sanitation and memory checks
Trigger Vulnerabilities
Exploitation
OOB Memory Read/Write
Verifier Exploitation
Manipulate Kernel Structs
Post-Exploitation
Privilege Escalation

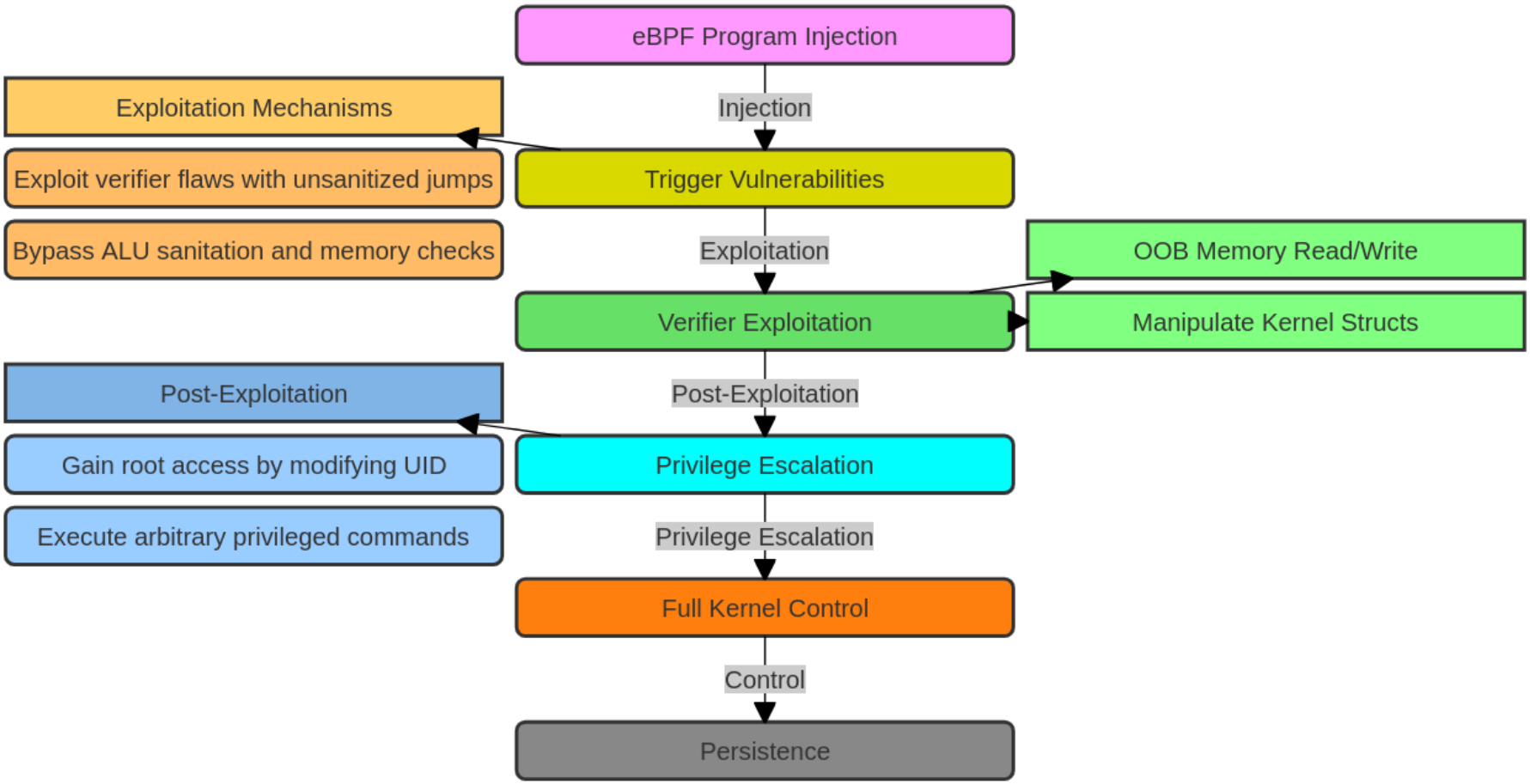

eBPF Program Injection
Injection
Exploitation Mechanisms
Exploit verifier flaws with unsanitized jumps
Bypass ALU sanitation and memory checks
Trigger Vulnerabilities
Exploitation
OOB Memory Read/Write
Verifier Exploitation
Manipulate Kernel Structs
Post-Exploitation
Post-Exploitation
Privilege Escalation


eBPF Program Injection
Injection
Exploitation Mechanisms
Exploit verifier flaws with unsanitized jumps
Bypass ALU sanitation and memory checks
Trigger Vulnerabilities
Exploitation
OOB Memory Read/Write
Verifier Exploitation
Manipulate Kernel Structs
Post-Exploitation
Post-Exploitation
Gain root access by modifying UID
Privilege Escalation

L'ATTAQUE


eBPF Program Injection
Injection
Exploitation Mechanisms
Exploit verifier flaws with unsanitized jumps
Bypass ALU sanitation and memory checks
Trigger Vulnerabilities
Exploitation
OOB Memory Read/Write
Verifier Exploitation
Manipulate Kernel Structs
Post-Exploitation
Post-Exploitation
Gain root access by modifying UID
Execute arbitrary privileged commands
Privilege Escalation


eBPF Program Injection
Injection
Exploitation Mechanisms
Exploit verifier flaws with unsanitized jumps
Bypass ALU sanitation and memory checks
Trigger Vulnerabilities
Exploitation
OOB Memory Read/Write
Verifier Exploitation
Manipulate Kernel Structs
Post-Exploitation
Post-Exploitation
Gain root access by modifying UID
Execute arbitrary privileged commands
Privilege Escalation
Privilege Escalation
Full Kernel Control


eBPF Program Injection
Injection
Exploitation Mechanisms
Exploit verifier flaws with unsanitized jumps
Bypass ALU sanitation and memory checks
Trigger Vulnerabilities
Exploitation
OOB Memory Read/Write
Verifier Exploitation
Manipulate Kernel Structs
Post-Exploitation
Post-Exploitation
Gain root access by modifying UID
Execute arbitrary privileged commands
Privilege Escalation
Privilege Escalation
Full Kernel Control
Control
Persistence

Réunion stratégique

- **Étape 1 : Création et suppression de répertoires**
  mkdir() créer une structure de répertoires profondément imbriquée (environ 1 million de répertoires imbriqués), dont la **longueur totale du chemin dépasse 1 Go**.
  Cette structure est montée via un *bind-mount* dans un espace utilisateur non privilégié, puis supprimée avec *rmdir()*.

- **Étape 2 : Blocage d'un programme eBPF**
  Un thread est créé pour allouer avec vmalloc() un petit programme eBPF (via l'appel système *BPF_PROG_LOAD*). Ce thread est ensuite bloqué (via userfaultfd ou FUSE) après que le programme eBPF a été validé par le vérificateur eBPF du noyau, mais avant qu'il ne soit compilé en JIT.

- **Étape 3 : Exploitation du dépassement de mémoire**

  Nous ouvrons le fichier /proc/self/mountinfo dans notre espace utilisateur non privilégié. Nous lançons *read()* sur le chemin du répertoire, ce qui provoque l'écriture de la chaîne *//deleted* à un **décalage de -2 Go - 10 octets** sous le début du tampon alloué par *vmalloc()*.

- **Étape 4 : Corruption du programme eBPF**
  La chaîne *//deleted* est utilisée pour **écraser une instruction du programme eBPF validé**, annulant ainsi les vérifications de sécurité du vérificateur eBPF du noyau.

- **Étape 5 : Arbitrage des lectures/écritures**

  Cette écriture est transformée en lecture/écriture arbitraire de la mémoire du noyau en utilisant les techniques **btf** et **map_push_elem** décrites par Manfred Paul dans :
  - *CVE-2020-8835 - Linux Kernel Privilege Escalation via Improper eBPF Program Verification.*

- **Étape 6 : Escalade de privilèges**
  Nous utilisons cette lecture arbitraire pour localiser le tampon **modprobe_path[]** dans la mémoire du noyau. Ensuite, l'écriture arbitraire permet de remplacer son contenu (par défaut *"/sbin/modprobe"*) par un chemin pointant vers notre propre exécutable et ainsi d'obtenir les **privilèges administrateurs**.

Détail de l'attaque

RÉPERTOIRE MALICIEUX

Il faut un répertoire avec un chemin dépassant 1 Go :

-> Théoriquement cela nécessite de créer plus de 4 millions de répertoires imbriqués

-> En pratique, show_mountinfo() remplace chaque caractère '\\' dans le chemin par la chaîne de 4 octets "\134".

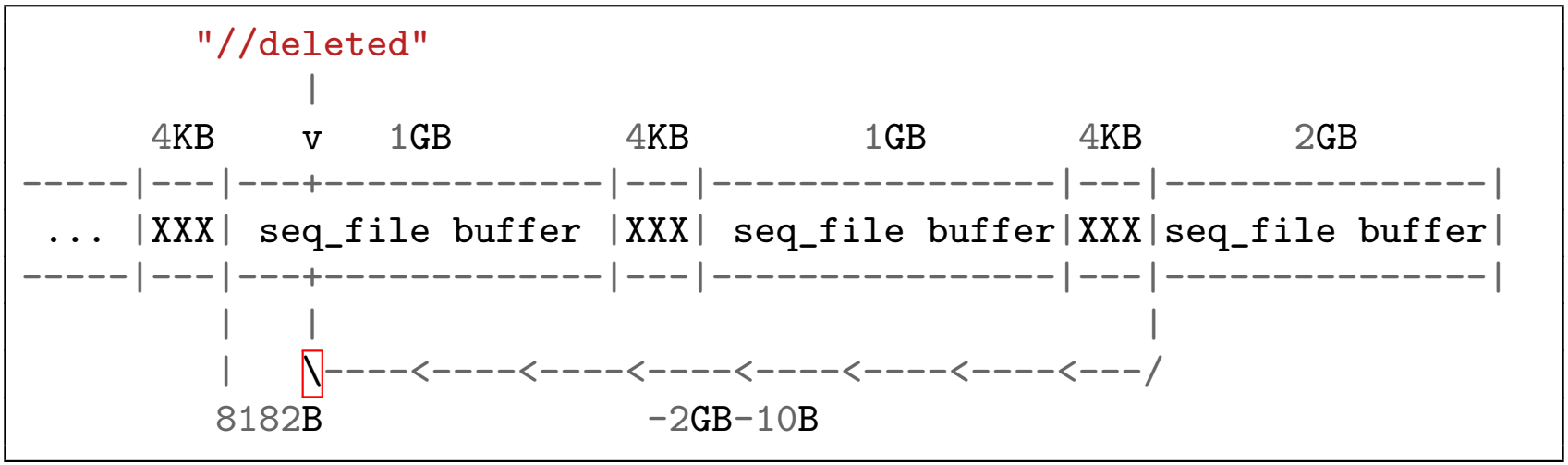
Cela réduit à 1 million de répertoires imbriqués la structure nécessaire.

REMPLISSAGE DE LA MÉMOIRE OU CARTOGRAPHIE DU TERRAIN

Remplissage des grandes zones libres vmalloc.

- Une partie du chemin très long, créé dans l'étape précédente (étape a), est montée via un bind-mount (MS_BIND) dans plusieurs espaces utilisateur non privilégiés.
- Des tampons sont alloués via des appels read() sur /proc/self/mountinfo (allocation de 768 Mo dans le crsh.c)

Allocations de deux tampons de 1Go et un tampon de 2Go.
Écriture hors-limite.



Calcul de la position de la chaîne : *(2*4Ko - 10)*

$2 * 4Ko = 2 * 4 * 1024 = 8192$

$8192 - 10 = 8182$

BOUCHER LES TROUS POUR FAIRE VENIR EBPF

Allocations via vmalloc() de nombreux messages NETLINK_USERSOCK.

Objectif, remplir la mémoire de petits objets inutile.

1. Lance 1024 threads -> charge un programme eBPF dans le noyau
2. Blocage du chargement avant l'allocation en mémoire

```
2076 static int bpf_prog_load(union bpf_attr *attr, union bpf_attr
↪  __user *uattr)
2077 {
....
2100          /* copy eBPF program license from user space */
2101          if (strncpy_from_user(license,
↪  u64_to_user_ptr(attr->license),
....
2161          /* plain bpf_prog allocation */
2162          prog = bpf_prog_alloc(bpf_prog_size(attr->insn_cnt),
↪  GFP_USER);
```

1. Libération du premier *seq_buffer*
2. Relâche des 1024 threads -> chargement en zone ciblé

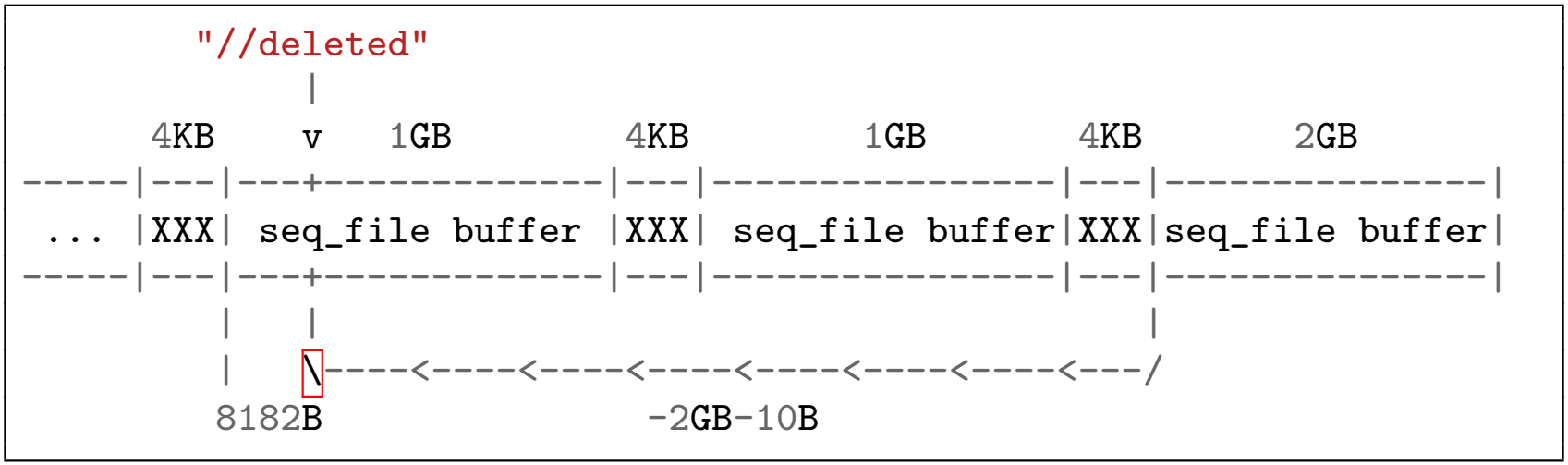
- Blocage d'un thread (ligne 12 795)

-> Après validation du verifieur eBPF

```
12640 int bpf_check(struct bpf_prog **prog, union bpf_attr *attr,
12641                union bpf_attr __user *uattr)
12642 {
.....
12795         print_verification_stats(env);
```

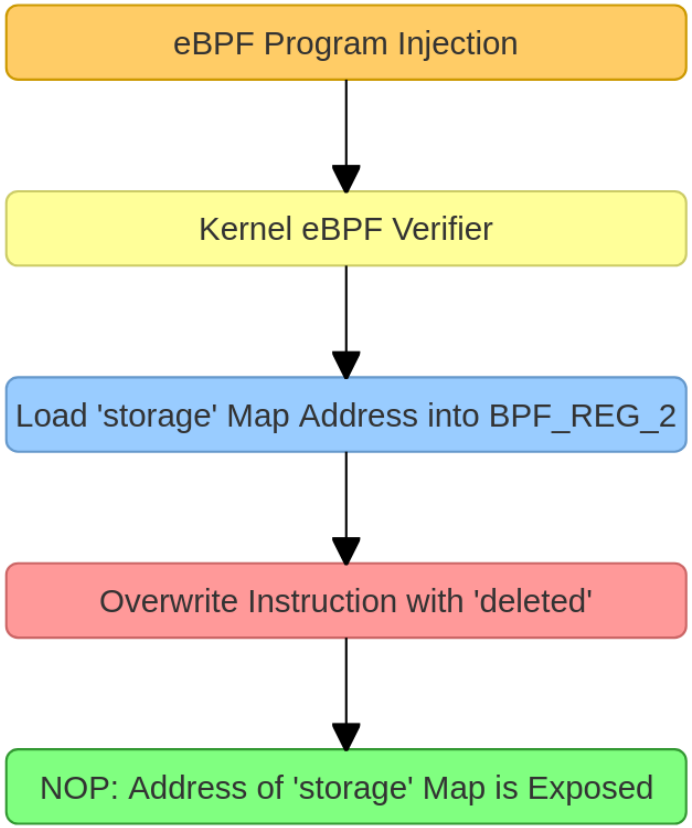
- Réécriture d'une instruction eBPF

-> Esquive des contrôles de sécurité



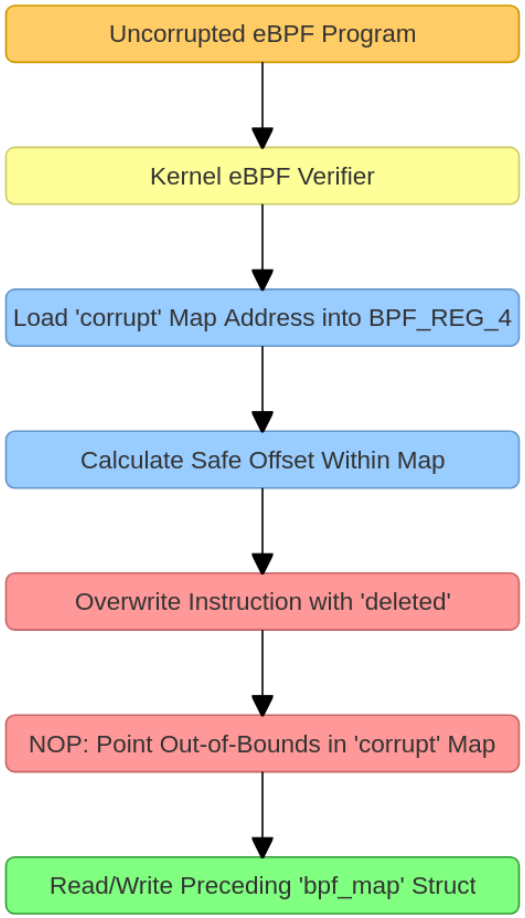
ASSAUT FINAL - 1


```
BPF_LD_IMM64_RAW(BPF_REG_2,
BPF_PSEUDO_MAP_VALUE,
storage)
BPF_MOV64_IMM(BPF_REG_2,
0)

BPF_LD_IMM64_RAW(BPF_REG_3,
BPF_PSEUDO_MAP_VALUE,
control)
BPF_STX_MEM(BPF_DW,
BPF_REG_3, BPF_REG_2, 0)
```

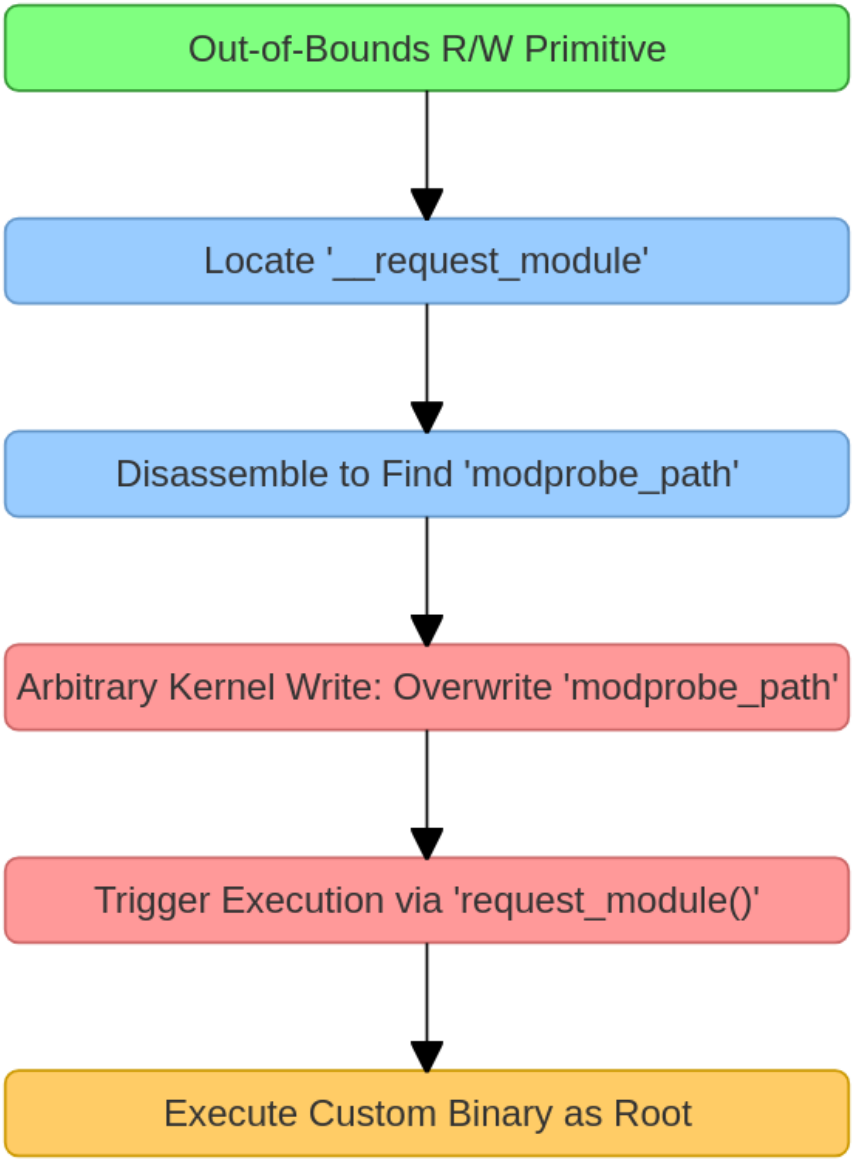


ASSAUT FINAL - 2


```
BPF_LD_IMM64_RAW(BPF_REG_4,
BPF_PSEUDO_MAP_VALUE,
corrupt)
BPF_ALU64_IMM(BPF_ADD,
BPF_REG_4, 3*64KB/2)
BPF_ALU64_IMM(BPF_SUB,
BPF_REG_4, 3*64KB/4)

BPF_LD_IMM64_RAW(BPF_REG_3,
BPF_PSEUDO_MAP_VALUE,
control)
BPF_LDX_MEM(BPF_H,
BPF_REG_7, BPF_REG_3, 0)
BPF_ALU64_REG(BPF_ADD,
BPF_REG_4, BPF_REG_7)
```



ASSAUT FINAL - 3



Solutions

Solutions qui permettent de prévenir l'attaque décrite ici :

- /proc/sys/kernel/unprivileged_userns_clone = 0
  - \+ Désactiver les espaces de noms utilisateurs non privilégiés
  - – L'outil FUSE permet toujours de monter un répertoire long (!)
- /proc/sys/kernel/unprivileged_b = 1
  - \+ Désactiver eBPF non privilégié
  - – on peut cibler d'autres objets (piles de threads)

Conclusion

1. https ://www.qualys.com/2021/07/20/cve-2021-33909/sequoia-local-privilege-escalation-linux.txt
2. https ://www.thezdi.com/blog/2020/4/8/cve-2020-8835-linux-kernel-privilege-escalation-via-improper-ebpf-program-verification

Duzés Florian

florian.duzes@u-bordeaux.fr

université de BORDEAUX